Random number generation and applications on GPUs in R?

Ruoyong Xu
University of Toronto

Patrick Brown
University of Toronto

Abstract

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. We introduce the R package **gpuRandom** that leverages the **gpuR** package, and can generate random numbers on GPU by utilizing the **clRNG** (OpenCL) library, the package enables reproducible research by setting random streams on GPU and can thus accelerate several types of simulation and modelling. **gpuRandom** is portable and flexible, developers can use its random number generation kernel for various other purposes and applications.

Keywords: GPU, **gpuRandom** 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 multi-core processors. Parallel computing is the solution to these drawbacks of R, 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`. One major way of parallel computing with R is using Graphics Processing Units (GPUs). GPUs can perform thousands of tests simultaneously, which makes them powerful at doing massive parallel computing and they are relatively cheap compared to multicore CPU's. Although there has been a few R packages developed that provide some GPU capability in the past years, they come with some restrictions. Packages such as **gputools**, **gpumatrix**, **cudaBayesreg**, **rpud** (available on github), are now no longer maintained, the popular **tensorflow** [**?**] package uses GPU via Python, which makes it hard to be incorporated in a new R package that uses GPU. All of these mentioned packages are restricted to R users with NVIDIA GPUs. **gpuR** [**?**] is the only R package with a convenient interface between R and GPUs and it is created for any R user with a GPU device. By utilizing the **ViennaCL** [**?**] library, it provides a bridge between R and non-proprietary GPUs through

an OpenCL (Open Computing Language) backend, which when combined with **Rcpp** [**?**] gives a building block for other R packages.

Random number generator (RNG) is critical in simulation-based statistical inference, there is not yet an R package that is able to generate random numbers on GPU. This is complicate to do on GPU because multiple streams will be needed for parallel generation of random numbers, and the stream states need to be able to be restored for reproducibility of simulation results. We introduce the **gpuRandom** R package 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, and describe and check the functions for generating uniform and Normal random numbers. In Section 3, we apply GPU-generated uniform random numbers in Monte Carlo simulation for Fishers exact test, we parallelise this Monte Carlo simulation task and implement it on GPU, then we provide two real data examples to demonstrate the function usage and test its R performance. Section 4 simulates simultaneously multiple Gaussian random surfaces by applying GPU-generated Normal random numbers and GPU-computed batches of Matérn covariance matrices, then we plot the generated random surfaces in the end. Finally, the paper concludes with a short summary and a discussion in Section 5.

2. Generating random numbers

Random numbers are the basis in statistical simulations methods, and are critical in many other fields. **?** 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 several general transformation methods for generating non-uniform random variables, for example, the most frequently used inverse transform method for generating a random variable X from some arbitrary distribution function F; The Box-Muller algorithm [**?**] for Gaussian random variable generation, and so on, which are all built on uniform random variables. Therefore, uniform random number generators (RNGs) are essential in simulating random numbers from all types of probability distributions. In the rest paper, "RNG" refers to uniform pseudo random number generator.

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 RNGs are actually stream objects of different forms. **gpuRandom** package uses the MRG31k3p RNG from the clRNG library, making it able to generate random numbers on GPUs. In what follows, we will illustrate how to create stream objects and how to use streams to generate (standard) Uniform and (standard) Gaussian random numbers.

2.1. Creating streams

In clRNG library, a stream object is a structure that contains three states (see the R output below): the current state of the stream, the initial state of the stream (or seed), and the

initial state of the current substream. The clRNG function `clrngMrg31k3pCreateStreams()` creates streams on the host computer 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 picks streams from the global memory and copies their current states only to its private memory. In **gpuRandom**, to avoid unnecessary data transfer between host and device, we are able to create streams both on the host computer and on the GPU device. `gpuRandom::CreateStreamsCpu()` is an R interface function of `clrngMrg31k3pCreateStreams()` that creates streams on the host as R matrices, so that users can easily view the streams. `gpuRandom::CreateStreamsGpu()` is adapted from `clrngMrg31k3pCreateStreams()` with the advantage of creating streams directly on the GPU device. From our experience, transferring a large-size matrix of streams from host to device by converting it to a '`vclMatrix`' object can take very long time, hence, `gpuRandom::CreateStreamsGpu()` is more efficient and more frequently used. Below, we show the usage of these two functions by generating 4 streams on the host and on the device respectively.

```
# Create streams on host
streams_on_CPU<-gpuRandom::CreateStreamsCpu(Nstreams=4)
t(streams_on_CPU)
##                  [,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
## substream.g1.1 12345  336690377  502033783  739421137
## substream.g1.2 12345  597094797 1322587635 1475938232
## substream.g1.3 12345 1245771585 1964121530  730262207
## substream.g2.1 12345   85196284 1949818481 1630192198
## substream.g2.2 12345  523477687 1607232546  324551134
## substream.g2.3 12345 2094976052 1462898381  795289868
```

`gpuRandom::CreateStreamsCpu()` has only one argument `Nstreams`, which is the number of streams to create. For the MRG31k3p RNG, each state of the stream object is comprised of six 31-bit integers, the state of the first stream (seed of the clRNG library) has a default value "12345". By setting a seed identical to the MRG31k3p default seed in `gpuRandom::CreateStreamsGpu()`, we can produce on device exactly the same streams as the streams created on host as follows,

```
# Create streams on GPU
seed <- c(12345, 12345, 12345, 12345, 12345, 12345)
streams1 <- gpuRandom::CreateStreamsGpu(seedR=seed, Nstreams=4, keepInitial=TRUE)
streams_from_GPU <- as.matrix(streams1)
streams2 = deepcopy(streams1)


t(streams_from_GPU)
##                  [,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
## substream.g1.1 12345  336690377  502033783  739421137
## substream.g1.2 12345  597094797 1322587635 1475938232
## substream.g1.3 12345 1245771585 1964121530  730262207
## substream.g2.1 12345   85196284 1949818481 1630192198
## substream.g2.2 12345  523477687 1607232546  324551134
## substream.g2.3 12345 2094976052 1462898381  795289868
```

The `gpuRandom::CreateStreamsGpu()` has three arguments:

- `seedR`: an R vector of length 6, which is the seed of streams. Default is set to a vector of six "12345".
- `Nstreams`: number of streams to create.
- `keepInitial`: logical specifying whether to keep the seed in generated streams. Default is set to TRUE.

Streams in **gpuRandom** are like `.Random.seed()` in R. By setting the RNG status with the `set.seed()` function in R, we are able to repeat the sequence of random seeds, which is an important quality criteria of RNGs. Similarly in **gpuRandom**, with the `seedR` argument, we are able to produce exactly the same sequence of streams as many times as we want.

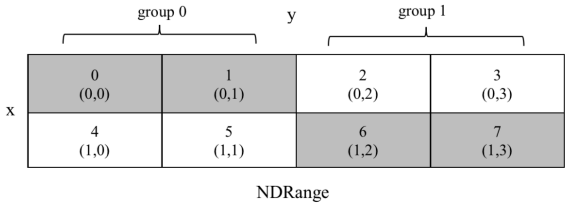
2.2. Using streams to generate i.i.d. $U(0,1)$ random numbers on GPU

The streams created on GPU are then used in work items that executes in parallel on GPU device, in **gpuRandom** each work item takes one distinct stream to generate random numbers, so number of streams should equal to the number of total work items in use. A part of the

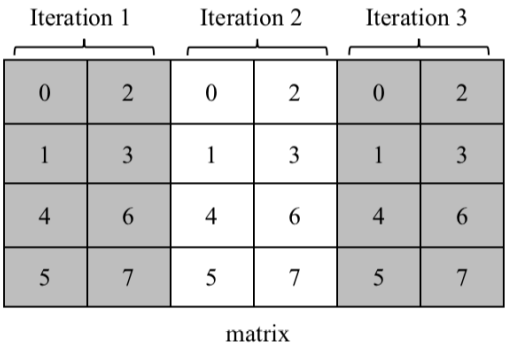
kernel (OpenCL functions for execution on the device) for generating uniform random numbers is shown below. Kernels are written in OpenCL C language, `__kernel` declares a function as a kernel, pointer kernel arguments must be declared with an address space qualifier, for example `__global` and `__local`. Pointers to the `streams` and to the matrix `out` on the global memory are passed to the kernel as arguments. `Nrow` and `Ncol` represent number of rows and columns of the matrix `out` respectively. `NpadCol` is the internal number of columns of `out`, `NpadStreams` is internal number of columns of the `streams` matrix (these variables are defined in macros not shown here). Each stream is taken to the private memory of each work item by the function `streamsToPrivate()`. Here we use a 2-dimensional index space with work-group size 1×2. `index` represents the position of the work item when mapping it to a one-dimensional vector. The function `clrngMrg31k3pNextState()` gives a uniform random integer named `temp` between 0 and 4294967295, 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 1/4294967295 in macro not shown here. 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 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);
}
```

The kernel code may not be straightforward enough in showing how streams are used in work items to generate random numbers, so here we give a small example to illustrate the matrix filling process. As shown in Figure 1, suppose we use a 2×4 NDRange (index space size) ((a) in Figure 1) and want to create a 4×6 matrix ((b) in Figure 1) of uniform random numbers using 8 streams. 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 work item index (each work item has a unique 2-dimensional global ID). Each stream object is passed to each work item. The number in a cell of matrix in (b) is the stream index, that indicates which stream does the random number generation for this matrix cell. For example, stream 0 which is allocated in work item $(0, 0)$ generates a random number in matrix $(0, 0)$ (row and column index of the matrix) 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 goes to matrix $(0, 4)$ and so on until it goes out of the matrix range in the next iteration.



(a)



(b)

Figure 1: name?

In the following code, we use the 4 streams created in Section 2.1 and 4 work items to generate a vector `sim_1` of 6 i.i.d. $U(0, 1)$ random numbers by the function `gpuRandom::runif()`. By default `Nglobal` is set as (64,8), and a matrix of `prod(Nglobal)` streams created on GPU is supplied. The output is an S4 object (a '`vclVector`' or a '`vclMatrix`'), to view the generated random numbers, we need to convert it to an R vector or matrix, in this way, the random numbers are moved from the global memory of device to the host computer.

```
sim_1 = gpuRandom::runif(n = 6, streams = streams1, Nglobal = c(1, 4),
    type = "double")
as.vector(sim_1)
## [1] 0.7353245 0.5180770 0.6142074 0.2319392 0.1100781 0.3619766
```

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

- `n`: a number specifying the length if to create a vector, or a vector of length 2 specifying the row and column number if to create a matrix.
- `streams`: streams used for random number generating.
- `Nglobal`: global index space that defines the total work-items that execute in parallel. Default is set as $(64, 8)$.
- `type`: "double" or "float" format of generated random numbers.

Next we show that using the same streams (`streams1` and `streams2`), `gpuRandom::runif()` returns a vector `sim_2` of same random numbers as `sim_1`.

```
sim_2 <- gpuRandom::runif(n = 6, streams = streams2, Nglobal = c(1, 4),
    type = "double")
as.vector(sim_2)
## [1] 0.7353245 0.5180770 0.6142074 0.2319392 0.1100781 0.3619766
```

Note that `streams2` is a `deepcopy` of `streams1` before `streams1` is used, reusing the `streams1` will produce a vector different from `sim_1` (result of `sim_3`), because each time a random number is generated, the current state of the stream is changed (advances by one position). See below the first six rows (current states) of `streams1` after it being used.

```
sim_3 = gpuRandom::runif(n = 6, streams = streams1, Nglobal = c(1, 4),
    type = "double")
as.vector(sim_3)
## [1] 0.6487742 0.1112075 0.3661944 0.5018562 0.1088229 0.3114331
t(as.matrix(streams1))[1:6, ]
##            [,1]       [,2]       [,3]       [,4]
## [1,] 1167281028  640923766  502033783  739421137
## [2,] 1918428443 1912157188 1322587635 1475938232
## [3,] 1858462085  286315187 1964121530  730262207
## [4,]  933585541 2119609883 1949818481 1630192198
## [5,] 1132031887  834429287 1607232546  324551134
## [6,]  465230163   47498872 1462898381  795289868
```

In order to reproduce the same sequence of random numbers, we can save the streams in a file on CPU, because GPU device does not keep memory for streams objects after each execution. Below is the R code in which we use the saved streams `streams_from_GPU.rds` to produce a vector `sim_saved`, which is also same as `sim_1`.

```
saveRDS(streams_from_GPU, "streams_from_GPU.rds")


# Load the streams object as streams_saved
streams_saved <- gpuR::vclMatrix(readRDS("streams_from_GPU.rds"))
sim_saved = gpuRandom::runif(n=6,streams=streams_saved,
                             Nglobal=c(1,4), type="double")
as.vector(sim_saved)
## [1] 0.7353245 0.5180770 0.6142074 0.2319392 0.1100781 0.3619766
```

We evaluate the distribution of 50000 GPU-generated uniform random numbers by making a density histogram (the left panel of Figure 2), plotting its empirical cumulative distribution and overlaying the theoretical cumulative distribution (the right panel of Figure 2). The two lines seem to overlap completely, suggesting the data generated by `gpuRandom::runif()` should come from a standard uniform distribution.

```
streams <- CreateStreamsGpu(Nstreams =256, keepInitial=1)
random_vector<-gpuRandom::runif(n=50000, streams=streams,
                                Nglobal=c(64,4), type="double")
random_vector<-as.vector(as.matrix(random_vector))
```

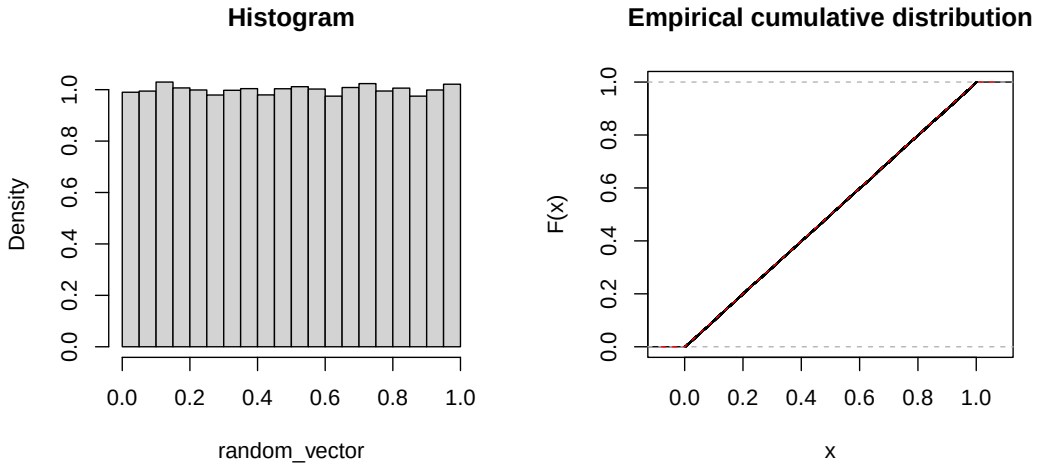


Figure 2: histogram and empirical cumulative distribution of the GPU-generated random numbers.

2.3. Using streams to generate i.i.d. $N(0,1)$ random numbers on GPU

We apply the Box-Muller transformation to $U(0,1)$ random numbers to generate Gaussian random numbers. Box-Muller algorithm (see algorithm 1) 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 derivation

basically uses transformation from Cartesian coordinates to polar coordinates to represent two independent, Normally distributed random variables. The Box-Muller algorithm turns out to be the best 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, but GPU is good at the high computational load of sine and cosine functions in the algorithm.

Algorithm 1: Box-Muller algorithm.

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

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

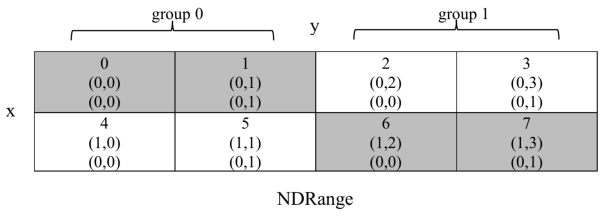
3, Take X and Y as two independent draws from $N(0,1)$;

Below is a fragment of the kernel function that generates standard Gaussian random numbers.

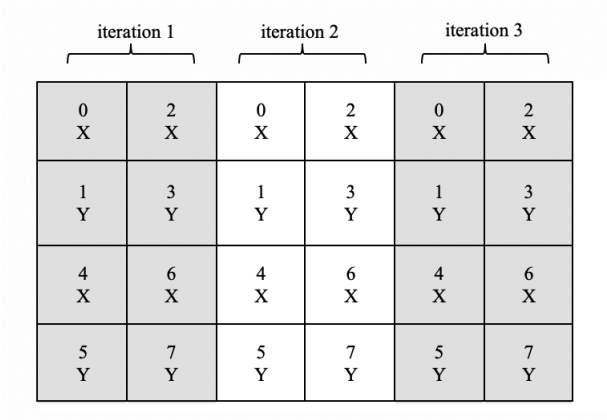
```
for(Drow=2*get_global_id(0),
    DrowStart=(Drow + get_local_id(1))* NpadCol;
    Drow < Nrow;
    Drow += DrowInc, DrowStart += DrowStartInc){

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);

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

     barrier(CLK_LOCAL_MEM_FENCE);

     if(get_local_id(1)) {
       out[Dentry] = part[0]*sinPart1;
     } else {
       out[Dentry] = part[0]*cosPart1;
     }
     barrier(CLK_LOCAL_MEM_FENCE);
   }//Dcol
   }//Drow
```

```
}
```

The kernel is adjusted from the kernel for generating uniform random numbers in 2.2, we only add the part that does the Box-Muller transformation. If developers want to use other Gaussian transform methods, this kernel code is easy to be modified and used in other R packages. In the code, `part0` and `part1` correspond to R and Θ in the formulas respectively. `barrier(CLK_LOCAL_MEM_FENCE)` ensures correct ordering of memory operations to local memory, so that $(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)$ are generated correctly in pairs, errors such as $(X_n, Y_{(n-1)})$ or $(X_{(n-1)}, Y_{(n)})$ will not occur. We illustrate the matrix filling process with the previous example: to create a 4×6 matrix ((b) in Figure 3) of Gaussian random numbers using a 2×4 NDRange ((a) in Figure 3) and 8 streams.



(a)



(b)

Figure 3: name???

As Gaussian numbers are generated in pairs using the Box-Muller method, work items are organized to be 1×2 work groups in the index space (a), the shading in the grids of (a) distinguishes work groups. Each work group generates a pair of Gaussian random numbers `(X,Y)` in each iteration, where the work item of '`get_local_id(1)`$= 0$' generates the `X` in a pair, and work item of '`get_local_id(1)`$= 1$' generates the `Y` in a pair. The numbers in the third row in each gird of (a) represents the local index of each work item. Cells of the matrix are filled from left to right, that is, in the first iteration of execution, the 8 work items generate random numbers in the leftmost two columns (8 cells) of 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. Therefore, each work item

generates one Gaussian random number in each iteration, the filling process is the same as the filling process of Figure 1.

We show the usage of `gpuRandom::rnorm()` by generating a 512×1024 matrix named `normal_matrix` of double presicion Gaussian random numbers, and assess Normality by making a histogram (left panel of Figure 4) and a Q-Q plot (right panel of Figure 4) of the generated random numbers. The points in the Q-Q plot in Figure 4 fall along the diagonal line, we have good evidence the data are from Gaussian distribution. We find doing a Q-Q plot can occupy a large amount of time in the process of producing an R markdown file, so as an incidental work, we make the function `gpuRandom::qqnorm()` that does the quantile computation for the normal distribution part (adapted from `stat::qnorm()`) on GPU, which speeds up the Q-Q plot greatly.

```
streams <- CreateStreamsGpu(Nstreams =512*128, keepInitial=1)
normal_matrix<-gpuRandom::rnorm(c(1024,512), streams=streams,
                                Nglobal=c(512,128), type="double")


avector<-as.vector(as.matrix(normal_matrix))


hist(avector,breaks=40, main = "histogram")
gpuRandom::qqnorm(avector, Nglobal=c(128,64), main="Q-Q Plot")
```

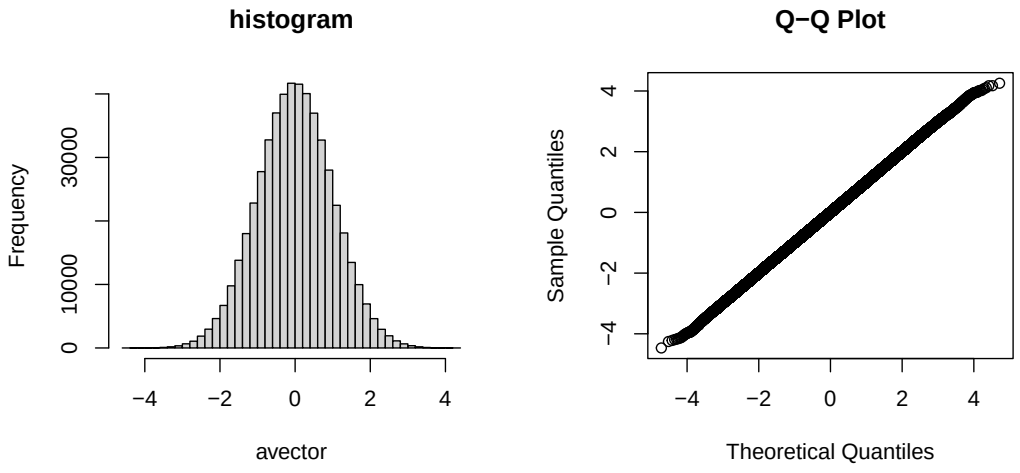


Figure 4: Histogram and Q-Q plot of the GPU-generated random numbers.

Let's compare the time of generating a large-size matrix of 100 million Gaussian random numbers of double precision using the `stats::rnorm()` and using the `gpuRandom::rnorm()`. Both process are not time-consuming, however, using `gpuRandom::rnorm()` with 512×128 work-items is 117.6 times (the number may fluctuate a bit from executions) faster than using the `stats::rnorm()`. The difference in elapsed time becomes larger when matrix size goes larger.

```
system.time(gpuRandom::rnorm(c(10000,10000), streams=streams,
                             Nglobal=c(512,128), type="double"))
##    user  system elapsed
##   0.060   0.003   0.062
system.time(matrix(stats::rnorm(10000^2),10000,10000))
##    user  system elapsed
##   7.023   0.966   7.985
```

3. Monte Carlo simulation for Fisher's exact test

One application of GPU-generated random numbers in **gpuRandom** is Monte Carlo simulation for Fisher's exact test. Fishers exact test is usually applied for analyzing 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 running 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. Given an observed table and a number of replicates B, the Monte Carlo simulation 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 (minus log-factorial of table) 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 $\frac{1+Counts}{B+1}$.

The Monte Carlo simulation solves the computational problem of Fisher's exact test but can still take long running time when B reaches millions in scale, thus we make the function `gpuRandom::fisher.sim()` that does the steps 1, 2 and 3 parallelly on GPU, and step 2 of saving the test statistics from the random tables as an option in `gpuRandom::fisher.sim()`, which largely saves processing time.

We show the usage and advantage of `gpuRandom::fisher.sim()` by computing the p-values for two real data 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 a normal GPU, the other is equipped with an excellent GPU and a normal CPU. The device information is included in the table 1. The R outputs for testing on computer 1 is shown in the following sections.

	CPU	GPU
Computer1	pthread-Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz	2 Radeon VII's : ROCm AMDGPU
Computer2	dontknow	GRID V100D-8C

Table 1: Device information for run-time compare.

These 2-way contingency tables (2 and 3) consists of selected data from the 2018 Natality public use file [**?**] from the Centers for Disease Control and Preventions National Center for Health Statistics, the 2018 natality data file may be downloaded at `https://www.cdc.gov/nchs/data_access/VitalStatsOnline.htm`. The row variable of the first table represents the twelve months: from January to December, and for the second 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.

3.1. Comparing running time: Month data example

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

```
#using CPU
stats::fisher.test(month,simulate.p.value = TRUE,B=1e6)$p.value
## [1] 0.4038036
system.time(stats::fisher.test(month,simulate.p.value = TRUE,B=1e6))
##    user  system elapsed
##  14.578   0.008  14.579
```

`stats::fisher.test()` takes about 9.8 seconds (time varies a little bit from executions) for one million simulations, has a p-value of 0.4035.

```
#using GPU
almost.1 <- 1 + 64 * .Machine$double.eps
observed_m= -gpuRandom::logfactSum(month, c(16,16))/almost.1
month_GPU<-vclMatrix(month,type="integer")
system.time(gpuRandom::fisher.sim(month_GPU, 1e6, streams=streams,
            type="double", returnStatistics=FALSE, Nglobal = c(128,128)))
##    user  system elapsed
##   0.322   0.003   0.325
```

`gpuRandom::fisher.sim()` takes about 2.3 seconds, compared to the elapsed time of `gpuRandom::fisher.si`
`gpuRandom::fisher.sim()` takes less than a quater of the time.

```
result_monthgpu <- gpuRandom::fisher.sim(month_GPU, 1e6, streams=streams,
                   type="double", returnStatistics=TRUE,  Nglobal = c(128,128))
result_monthgpu$simnum
## [1] 999424
result_monthgpu$counts
## [1] 403274
result_monthgpu$p.value
## [1] 0.403507
```

We requested one million simulations on GPU while the actual number of simulation is 999424, and got 403274 cases whose test statitsic is below the observed threshold, so the p-value from `gpuRandom::fisher.sim()` for this table is about 0.4035, which is close enough to the p-value from `stats::fisher.test()`: 0.4034.

3.2. Comparing running time: Week data example

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

We perform 10 million simulations for the “week” table.

```
#using CPU
fisher.test(week,simulate.p.value = TRUE,B=1e7)$p.value
## [1] 0.0001208
system.time(fisher.test(week,simulate.p.value = TRUE,B=1e7))
##    user  system elapsed
##  90.125   0.123  90.194
```

The "week" table has a much smaller p-value: around 0.000119, which should require a larger number of simulations to get a more accurate p-value.

```
#using GPU
observed_w= -gpuRandom::logfactSum(week, c(16,16))/almost.1
week_GPU<-gpuR::vclMatrix(week,type="integer")
system.time(gpuRandom::fisher.sim(week_GPU, 1e7, streams=streams,
            type="double", returnStatistics=FALSE,Nglobal = Nglobal))
##    user  system elapsed
##   1.893   0.012   1.905
```

`stats::fisher.test()` takes 60.4 seconds, while `fisher.sim()` takes about 10.8 seconds, the elapsed time is decreased to about 20% of the time taken on CPU.

```
result_weekgpu<-gpuRandom::fisher.sim(week_GPU, 1e7, streams=streams,
                type="double",returnStatistics=TRUE,Nglobal = Nglobal)
result_weekgpu$simnum
## [1] 9994240
result_weekgpu$counts
## [1] 1272
result_weekgpu$p.value
## [1] 0.0001273734
```

With about ten million simulations, we get 1272 cases and a p-value around 0.0001274. We summarized the results in Table 5.

data	B	Intel CPU	AMD GPU	NVIDIA GPU
Month	1e6	0.4038	0.403507	0.403507
Week	1e7	0.0001208	0.0001274	0.0001274

Table 4: Summary of p-values obtained on different devices.

data	B	Intel CPU	sencha CPU	AMD GPU	NVIDIA GPU
Month	1e6	9.74	14.58	2.28	0.31
Week	1e7	60.19	90.19	10.82	1.90

Table 5: Run-time compare of Fisher's test simulation on different devices, all numbers are in seconds.

Figure 5 visualizes the approximate sampling distributions of the test statistics from the two examples "Month" and "Week", by plotting a histogram for each of the example. The values of observed statistics are calculated by `gpuRandom::logfactSum()` and are marked with a blue line on each plot.

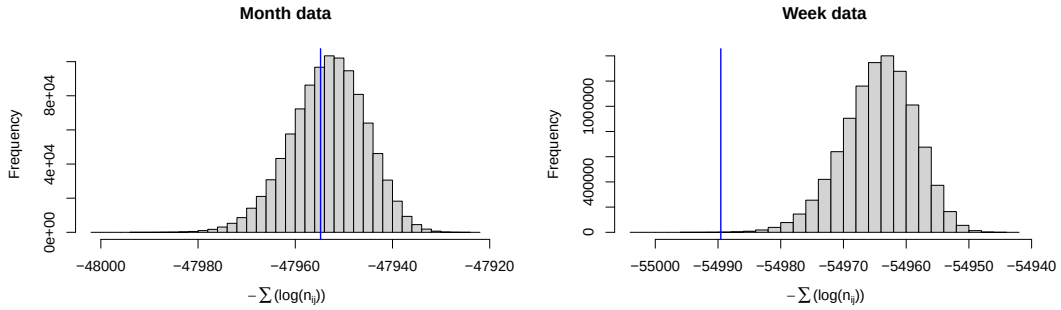


Figure 5: Frequency distribution for test statistics from month and week data tables.

4. Generating Gaussian random fields

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. One reason that Gaussian random fields play an important role is that the specification of their finite-dimensional distributions is simple [**?**], the expectation $\mu(x)$ and covariance function Σ completely determine the distribution of U, where

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

There exist many choices for the covariance function Σ, one common choice is the Matérn covariance function [**?**]. 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 $\kappa > m$. (For $\kappa = 0.5$, the Matérn covariance function reduces to the exponential covariance function $\sigma^2 * \exp(-d/\phi)$, when $\kappa \to \infty$, $\Sigma(d) \to \sigma^2 * \exp -(\|d\|/\phi)^2$, which is called the Gaussian covariance.) There are several alternative parameterisations of Matérn covariance functions appeared in literatures [see **?**]. In **gpuRandom**, we use the above form for computing Matérn covariance.

Due to the importance of Gaussian random field, there are a large amount of studies devoted to Gaussian random field generation techniques. **?** gives a comprehensive review on 7 popular methods for Gaussian random field generation, which are the turning bands method [**?**], spectral method [**??**], matrix decomposition method [**?**], Karhunen-Loéve expansion [**?**], moving average method [**??**], sequential simulation [**???**], and local average subdivision [**?**].

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. **gpuRandom** 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 (n) of locations 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 $\mathcal{O}(n^2)$ [**?**]. **gpuRandom** package takes advantage of GPU's parallel computing power to not only do 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, and introduces several GPU functions used in the simulation process.

4.1. Simulating Gaussian random fields with Matérn covariances

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 `gpuRandom::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 = gpuRandom::maternGpuParam(myParamsBatch, type=theType)
```

Step 3, compute Matérn covariance matrices using `gpuRandom::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
gpuRandom::maternBatch(maternCov, coordsGpu, paramsGpu,
                       Nglobal=c(128,64), Nlocal=c(16,4))
```

Step 4, the first argument `maternCov` in `gpuRandom::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)
gpuRandom::cholBatch(maternCov, diagMat, numbatchD=nrow(myParamsBatch),
                     Nglobal= c(512, 8),
                     Nlocal= c(32, 8),
                     NlocalCache=2000)
```

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

```
streamsGpu <- gpuRandom::CreateStreamsGpu(Nstreams=128*64, keepInitial=TRUE)


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

Step 6, use `gpuRandom::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.

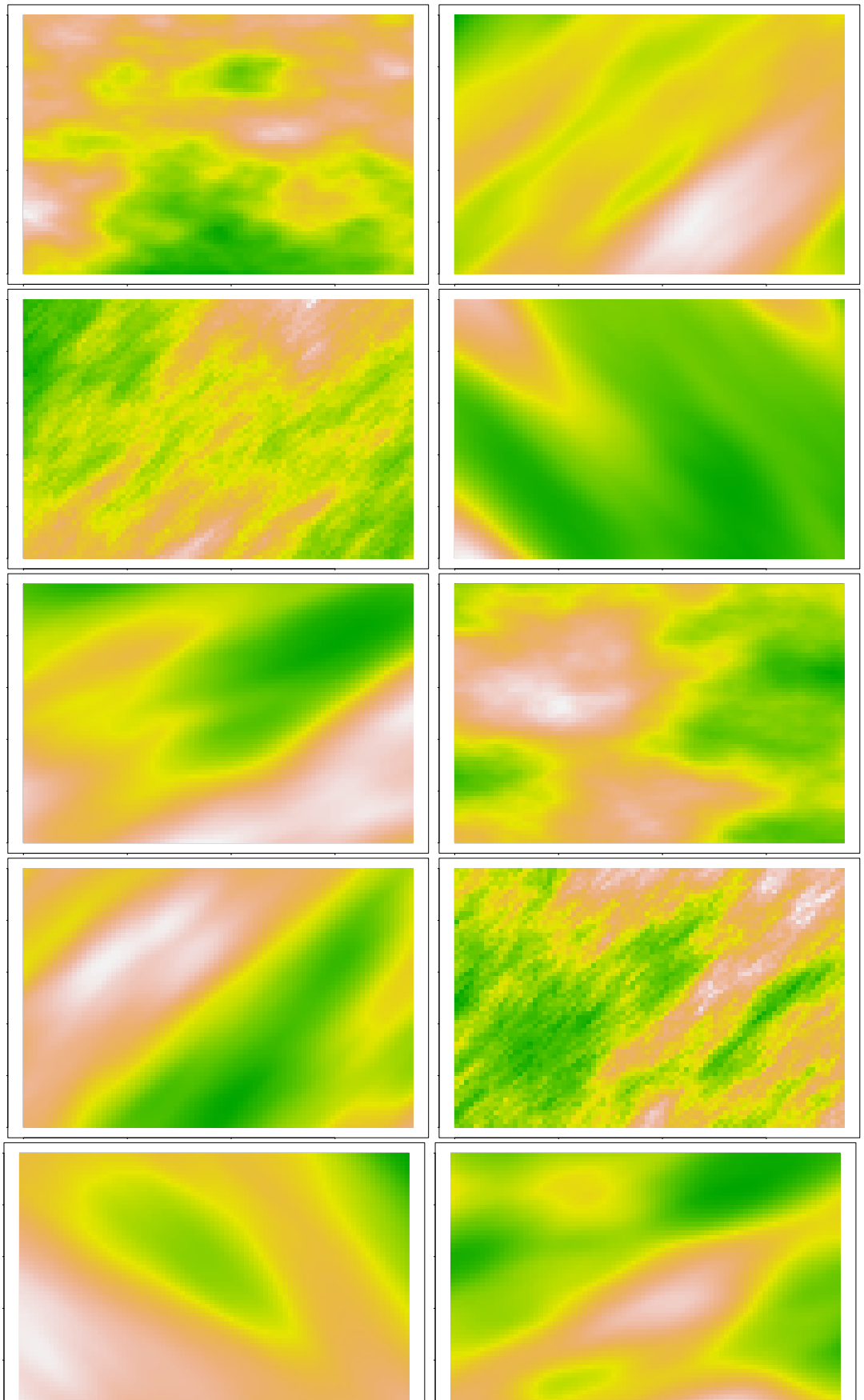
```
simMat = vclMatrix(0, nrow(zmatGpu), ncol(zmatGpu),
                   type = theType)

gpuRandom::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)
}
```



5. Discussion

The package **gpuRandom** 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 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, all of them uses parallel programming and avoids unnecessary data transfers between host and device. Comparison of performance between using **gpuRandom** and using classic R on CPU for some real data examples has demonstrated significant improvement in execution time.

By leveraging the **gpuR** package, **gpuRandom** 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. **gpuRandom** 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 further development.

One of the main difficulties in developing the package is bug fixes and uncertain software behaviors. Error messages are often vague and it is time-consuming to locate the error within the backend C program. The usual steps that we take to debug is commenting out some parts of codes and then recompile many times until we find the problem, or printing out flag messages in several places in the code to find out where the program stops at due to error. **gpuRandom** 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 **gpuRandom** 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:

Achim Zeileis
Journal of Statistical Software
and
Department of Statistics
University of Toronto
Universität Innsbruck
Universitätsstr. 15
6020 Innsbruck, Austria
E-mail: ruoyong.xu@mail.utoronto.ca