Random number generation and applications on GPU's 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 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. 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 [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 of R, 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`.

An important 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 Packages such as **gputools**, **gpumatrix**, **cudaBayesreg**, **rpud** (available on github), are now no longer maintained, the popular **tensorflow** [Allaire, Eddelbuettel, Golding, and Tang 2016] package uses GPU via Python, which makes it hard to be incorporated in a new R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** [Determan Jr.

2017] 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** [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** (not on cran, how to cite?) 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, however, it is not very clear how the streams are implemented (?).

There is not yet an R package that is able to generate random numbers on GPU, this is complicated to do 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 results, which is essential in many simulation applications. We introduce the R package **gpuRandom** 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 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 Fisher's exact test, we parallelise this Monte Carlo simulation 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

Uniform random number generators (RNGs) are essential in 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 transformation 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. **gpuRandom** package uses the MRG31k3p RNG from the clRNG, making it able to generate random numbers on GPUs. In what follows, we will illustrate how to create streams and how to use streams to generate (standard) uniform and (standard) Gaussian random numbers.

2.1. Creating streams

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). Each of these streams can also be partitioned into substreams with equally-spaced starting points [L'ecuyer *et al.* 2002], which are occasionally useful.

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). 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.

In clRNG library, the initial state of the first stream has a default value (12345). 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 is further partitioned into substreams of length $W = 2^{72}$. The state (and seed) of each stream is a vector of six 31-bit integers. This size of state is appropriate for having streams running in work items on GPU cards, for example, while providing a sufficient period length for most applications.

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. **gpuRandom** 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), creating streams on host makes it easy for us to view and check the streams behavior. `createStreamsGpu()` is adapted from the `clrngMrg31k3pCreateStreams()` and creates streams directly on a GPU device. Substreams are not created because currently we use one stream per work-item. In the following, we demonstrate the usage of these two functions by generating 4 streams on the host and on the device respectively.

```
# creating streams on CPU
library("gpuR")
library("gpuRandom")
myStreamsCpu <- createStreamsCpu(n=4, seed=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.
- `seed`: vector of length 6, recycled if shorter. Default is set to a vector of six "12345".

We can have streams on GPU by either converting the `myStreamsCpu` to a '`vclMatrix`' or producing streams directly on GPU using `createStreamsGpu()`. Setting a seed same as the MRG31k3p default seed in `createStreamsGpu()`, we can produce on device exactly the same streams as those created on host as follows,

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

`createStreamsGpu()` also has the two arguments `n` and `seed`, `seed` is by default a vector of six "12345". `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 `seed` argument, we make the sequence of streams repeatable.

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

The streams created sequentially on a GPU are then used in work-items that executes in parallel on a GPU device. In **gpuRandom** 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

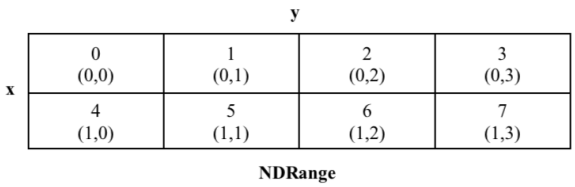
one-dimensional data structure. The function `clrngMrg31k3pNextState()` generates an uniform random integer called `temp` between 0 and 4294967295, the value of `temp` is then scaled to be in the interval $(0, 1)$ by multiplying it by a constant `mrg31k3p_NORM_cl`, which is defined to be 1/4294967295 in macro not shown here. At the end of generating random numbers, streams are transferred back to global memory through the function `streamsFromPrivate()`.

Listing 1: captiontext

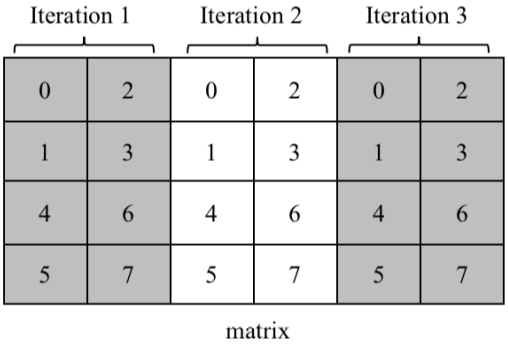
```
__kernel void mrg31k3pMatrix(
    __global int* streams,
    __global double* out){

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: name?

Now we use the streams created in section 2.1 to generate two vectors of double-precison i.i.d. $U(0, 1)$ random numbers with the function `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.

```
as.vector(gpuRandom::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(gpuRandom::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 `gpuRandom::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 if it uses the same streams in the same current state (here `myStreamsGpu1` and `myStreamsGpu2`), not streams starting from the same states. 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. Below shows the `myStreamsGpu1` after it being used.

```
t(matrix(as.matrix(myStreamsGpu1), nrow(myStreamsCpu), ncol(myStreamsCpu),
    dimnames = dimnames(myStreamsCpu)))
##                    [,1]       [,2]       [,3]       [,4]
## current.g1.1  878672095 2113333390  502033783  739421137
## current.g1.2  240667857  559530223 1322587635 1475938232
## current.g1.3  240667857 1309565828 1964121530  730262207
## current.g2.1  642281259 1335994581 1949818481 1630192198
## current.g2.2 1069151070   61444481 1607232546  324551134
## current.g2.3  809054265  197003928 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
```

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 a trivial example, in which we demonstrate saving 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"))
```

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

```
streams <- createStreamsGpu(n = 512)

random_u<-as.vector(runif(n=50000, streams=streams, Nglobal=c(64,8), type="double"))
```

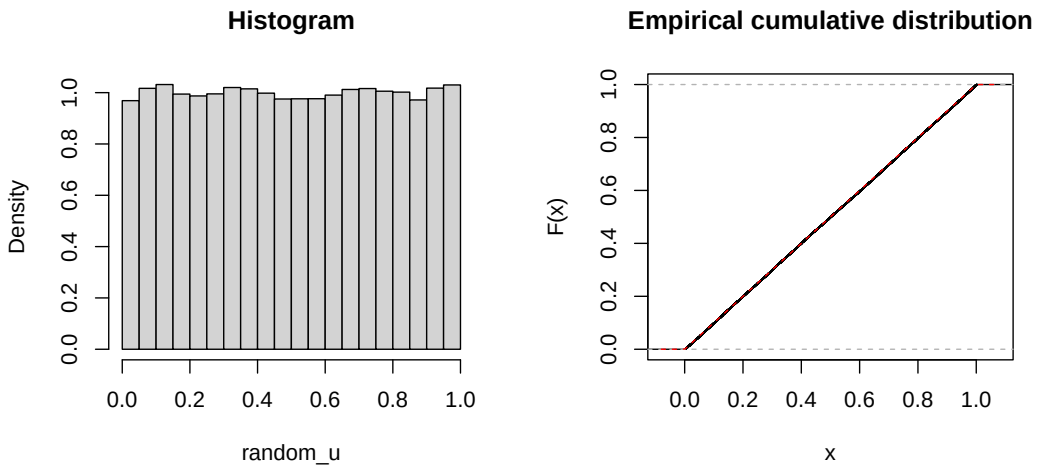


Figure 2: Histogram and empirical cumulative distribution of the GPU-generated random numbers.

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 standard Gaussian 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)`

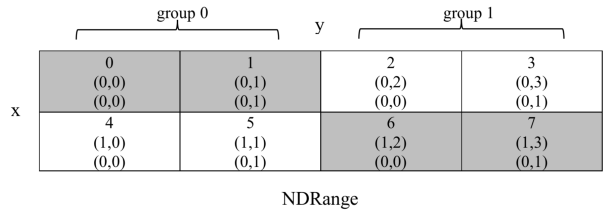
ensures correct ordering of memory operations to local memory, so that Gaussian random numbers $(X_1, Y_1), \ldots, (X_n, Y_n)$ are generated in pairs correctly, errors such as $(X_n, Y_{(n-1)})$ or $(X_{(n-1)}, Y_{(n)})$ are avoided.

Listing 2: name

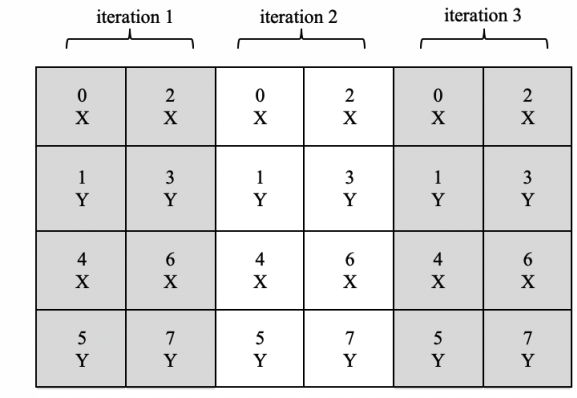
```
const int
index = get_global_id(0)*get_global_size(1) + get_global_id(1);
const int
DrowInc = 2*get_global_size(0), DrowStartInc = DrowInc * NpadCol;
int Drow, Dcol, DrowStart, Dentry;
uint temp;
uint g1[3], g2[3];
const int startvalue=index * NpadStreams;
local double part[2], cosPart1, sinPart1;

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 3 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 3: name???

In the following code, We create a 1024×512 matrix of single-precision floating-point Gaussian random numbers, and assess Normality by making a histogram and a Q-Q plot of the

generated random numbers in Figure 4. The points in the Q-Q plot shown in the right panel of Figure 4 fall along the diagonal line, we have good evidence the generated 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 adapted the `stat::qnorm()` to make the function `gpuRandom::qqnorm()` that does the quantile computation for the normal distribution part on a GPU device, which can speed up the Q-Q plot.

```
streams <- createStreamsGpu(n =512*128)
normal_matrix<-gpuRandom::rnorm(c(1024,512), streams=streams,
                                Nglobal=c(512,128), type="float")
avector<-as.vector(as.matrix(normal_matrix))


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

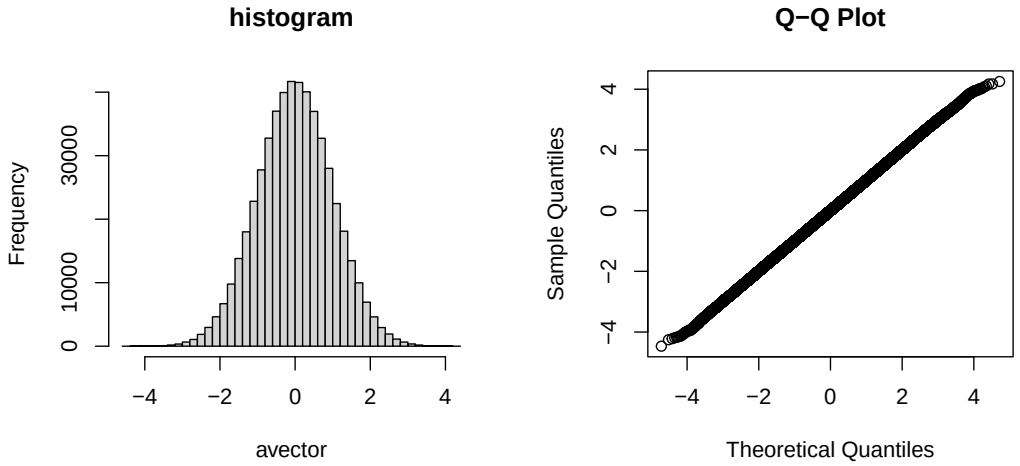


Figure 4: Histogram and Q-Q plot of the GPU-generated random numbers.

We generate a large-size matrix of 100 million double-precision Gaussian random numbers, and compare the run-time between using `stats::rnorm()` and `gpuRandom::rnorm()`. Both process are not time-consuming, however, using `gpuRandom::rnorm()` with 512×128 work-items is around 120 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
##   0.144   0.004   0.189
system.time(matrix(stats::rnorm(10000^2),10000,10000))
```

```
##    user  system elapsed
##   6.762   0.868   7.624
```

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. 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. 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 that have test statistics less or equal to the one from the observed table ($Counts$).
4. Estimate p-value using $\frac{1+Counts}{B+1}$.

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 `gpuRandom::logfactSum()`.

`stats::fisher.test()` can take a long running time when B reaches millions as the B random tables are constructed sequentially, one in each iteration. `gpuRandom::fisher.sim()` adapted the backend function `rcont2()` used by `stats::fisher.test()` for constructing random tables, so that it generates `prod(Nglobal)` random tables in parallel in each iteration (i.e., one random table each work-item per iteration), and the steps 2 and 3 are also done in parallel by work-items at each iteration on GPU. Step 2 of saving test statistics from the random tables is made optional, which can reduce the run-time.

We show the usage and advantage of `gpuRandom::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 table 1 and table 2 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. 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 1: 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

```
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(256,128))
result_month$simNum
## [1] 1015808
result_month$counts
## [1] 410644
result_month$p.value
## [1] 0.4042541
```

We requested one million simulations on GPU while the actual number of simulation is 1015808, 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 0.40375, which is close enough to the p-value from `stats::fisher.test()`: 0.40380.

```
#using GPU
system.time(fisher.sim(month_gpu, 1e6, streams=streams,
            type="double", returnStatistics=FALSE, Nglobal = c(256,128)))
##    user  system elapsed
##   0.720   0.001   0.721
```

`gpuRandom::fisher.sim()` takes about 0.32 seconds, **gpuRandom** accounts for 2.2% of the elapsed time on CPU.

```
#using CPU
stats::fisher.test(month,simulate.p.value = TRUE,B=1015808)$p.value
## [1] 0.4035335
system.time(stats::fisher.test(month,simulate.p.value = TRUE,B=1015808))
##    user  system elapsed
##  14.706   0.004  14.730
```

`stats::fisher.test()` takes about 9.8 seconds (time varies a little bit from executions) for 1015808 simulations, has a p-value of 0.4035.

3.2. Comparing running time: Week data example

Table 2: 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

```
observed_w= -gpuRandom::logfactSum(week, c(16,16))/almost.1
week_GPU<-gpuR::vclMatrix(week,type="integer")
result_weekgpu<-gpuRandom::fisher.sim(week_GPU, 1e7, streams=streams,
                type="double",returnStatistics=TRUE,Nglobal = c(256,128))
result_weekgpu$simNum
## [1] 10027008
result_weekgpu$counts
## [1] 1242
```

```
result_weekgpu$p.value
## [1] 0.0001239652
```

With about ten million simulations, we get 1223 cases and a p-value around 0.0001224705. We summarized the results in Table 3.

```
#using GPU
system.time(gpuRandom::fisher.sim(week_GPU, 1e7, streams=streams,
            type="double", returnStatistics=FALSE,Nglobal = c(256,128)))
##    user  system elapsed
##   4.585   0.001   4.582
```

`stats::fisher.test()` takes 90.1 seconds, while `fisher.sim()` takes about 1.90 seconds, the elapsed time is decreased to about 20% of the time taken on CPU.

```
#using CPU
fisher.test(week,simulate.p.value = TRUE,B=10010624)$p.value
## [1] 0.0001292627
system.time(fisher.test(week,simulate.p.value = TRUE,B=10010624))
##    user  system elapsed
##  90.434   0.196  90.561
```

The "week" table has a much smaller p-value: around 0.0001208, which should require a larger number of simulations to get a more accurate p-value.

3.3. A summary of the results

(Results obtained in first week of October)

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	9.74	2.28	14.58	0.32	month
10m	60.19	10.82	90.38	1.89	week

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.

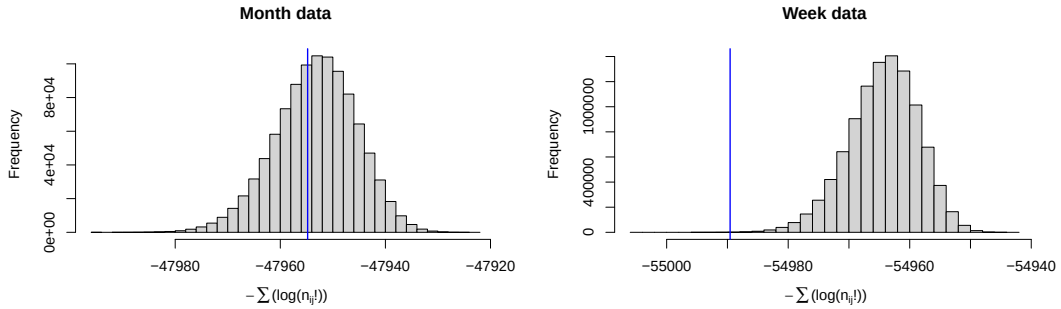


Figure 5: Frequency distribution for test statistics from month and week data tables.

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. 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 **gpuRandom**, 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, which are the turning bands method [Matheron 1973], spectral method [Mejía and Rodriguez-Iturbe 1974; Shinozuka and Jan 1972], matrix decomposition method [Davis 1987], Karhunen-Loéve expansion [Loeve 1978], moving average method [Journel 1974; Oliver 1995], sequential simulation [Johnson 1987; Gómez-Hernández and Journel 1993; Pebesma 2004], and local average subdivision [Fenton and Vanmarcke 1990]. 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. **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 of locations (n) is involved, the cost of Cholesky decomposition of the covariance matrix is $\mathcal{O}(n^3)$ [Liu *et al.* 2019], and the cost of matrix-vector multiplication $L * Z$ to generate the random field U is $\mathcal{O}(n^2)$ [Liu *et al.* 2019]. **gpuRandom** 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.

4.1. Simulating Gaussian random fields with Matérn covariances

```
library("gpuR")
library("gpuRandom")
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 `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(128, 8),
                     Nlocal= c(32, 8),
                     NlocalCache=1000)
##
## #pragma OPENCL EXTENSION cl_khr_fp64 : enable
##
##
## #define N 4800  //dimension of matrix
## #define colStart 0  //column to start at
## #define colEnd 4800
```

```
## //internal number of columns
## #define Npad 4864
## //internal columns for matrix holding diagonals
## #define NpadDiag 4864
## #define NstartA 0
## #define NstartD 0
## //elements in internal cache
## #define Ncache 1000
## //extra rows between stacked matrices
## #define NpadBetweenMatrices 23347200
## #define maxLocalItems 256
##
## __kernel void cholBatch(
##  __global double *A,
##  __global double *diag,
##  __local double*diagLocal,
##               int Nmatrix
## ){
##  const int localIndex = get_local_id(0)*get_local_size(1) + get_local_id(1);
##  const int localIndexIsZero = (localIndex==0);
##  const int NlocalTotal = get_local_size(0)*get_local_size(1);
##  local double toAddLocal[maxLocalItems];
##  int Dcol, DcolNpad;
##  int Drow, DrowBlock, Dk, Dmatrix, DmatrixBlock;
## int minDcolNcache;
##  double DL;
##  local double diagDcol;
##  int AHere, AHereDcol, AHereDrow, diagHere;
##  barrier(CLK_LOCAL_MEM_FENCE);
##
## for(Dmatrix = get_group_id(0);
##      Dmatrix < Nmatrix;
##      Dmatrix+= get_num_groups(0)){
##
##  diagHere = Dmatrix*NpadDiag + NstartD;
##  AHere =  Dmatrix*NpadBetweenMatrices + NstartA;
##
##  for(Dcol = colStart; Dcol < colEnd; Dcol++) {
##   DcolNpad = Dcol*Npad;
##   AHereDcol = AHere+DcolNpad;
##   DL = 0.0;
##   diagLocal[localIndex]=0.0;
##   toAddLocal[localIndex]=0.0;
##   minDcolNcache = min(Dcol, Ncache);
##   for(Dk = localIndex; Dk < minDcolNcache; Dk += NlocalTotal) {
##     DL = A[AHereDcol+Dk];
```

```
##     diagLocal[Dk] = diag[diagHere+Dk] * DL;
##     toAddLocal[localIndex] += diagLocal[Dk] * DL;
##   }// Dk
##   for(Dk=minDcolNcache+localIndex; Dk < Dcol; Dk += NlocalTotal) {
##     DL = A[AHereDcol+Dk];
##     toAddLocal[localIndex] += diag[diagHere+Dk] * DL * DL;
##   }// Dk
##
## // reduction on dimension 1
##   barrier(CLK_LOCAL_MEM_FENCE);
##   if( (get_local_id(1) == 0)){
##    for(Dk = 1; Dk < get_local_size(1); Dk++) {
##     toAddLocal[localIndex] +=  toAddLocal[localIndex + Dk];
##    }//for Dk
##   }//get_local_id(1) == 0
##   barrier(CLK_LOCAL_MEM_FENCE);
##
## // final reduction on dimension 0
##   if(localIndexIsZero ){
##    for(Dk = get_local_size(1); Dk < NlocalTotal; Dk+= get_local_size(1)) {
##     toAddLocal[localIndex] +=  toAddLocal[Dk];
##    } //Dk
##    diagDcol = A[AHereDcol+Dcol] - toAddLocal[localIndex];
##    diag[diagHere+Dcol] = diagDcol;
## #ifdef diagToOne
##    A[AHereDcol+Dcol] = 1.0;
## #endif
##   }  //localIndex==0 and Dmatrix < Nmatrix
##   barrier(CLK_LOCAL_MEM_FENCE);
##
## // off diagonals
##  for(DrowBlock = Dcol+1; DrowBlock < N; DrowBlock += get_local_size(0)) {
##   Drow = DrowBlock + get_local_id(0);
##   AHereDrow = AHere+Drow*Npad;
##   DL = 0.0;
##   if(Drow < N){
##   for(Dk = get_local_id(1); Dk < minDcolNcache; Dk+=get_local_size(1)) {
##    DL += A[AHereDrow+Dk] * diagLocal[Dk];
##   } // Dk
##   for(minDcolNcache + get_local_id(1); Dk < Dcol; Dk+=get_local_size(1)) {
##     DL += A[AHereDrow+Dk] * diag[diagHere+Dk] * A[AHereDcol+Dk];
##   }// Dk
##    }//Drow < N
##   toAddLocal[localIndex] = DL;
##
##   // local reduction
```

```
##   barrier(CLK_LOCAL_MEM_FENCE);
##   if( (get_local_id(1) == 0) & (Drow < N)){
##    DL = toAddLocal[localIndex];
##    for(Dk = 1; Dk < get_local_size(1); Dk++) {
##     DL +=  toAddLocal[localIndex + Dk];
##    }//Dk
##    A[AHereDrow+Dcol] = (A[AHereDrow+Dcol] - DL)/diagDcol;
##   }//get_local_id(1) == 0
##   barrier(CLK_GLOBAL_MEM_FENCE);
##
##  }//DrowBlock
## barrier(CLK_GLOBAL_MEM_FENCE);
## } // Dcol loop
##
## barrier(CLK_GLOBAL_MEM_FENCE);
## } // Dmatrix loop
##
## barrier(CLK_LOCAL_MEM_FENCE);
## }// kernel
```

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 <- createStreamsGpu(n=128*64)



zmatGpu = 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.

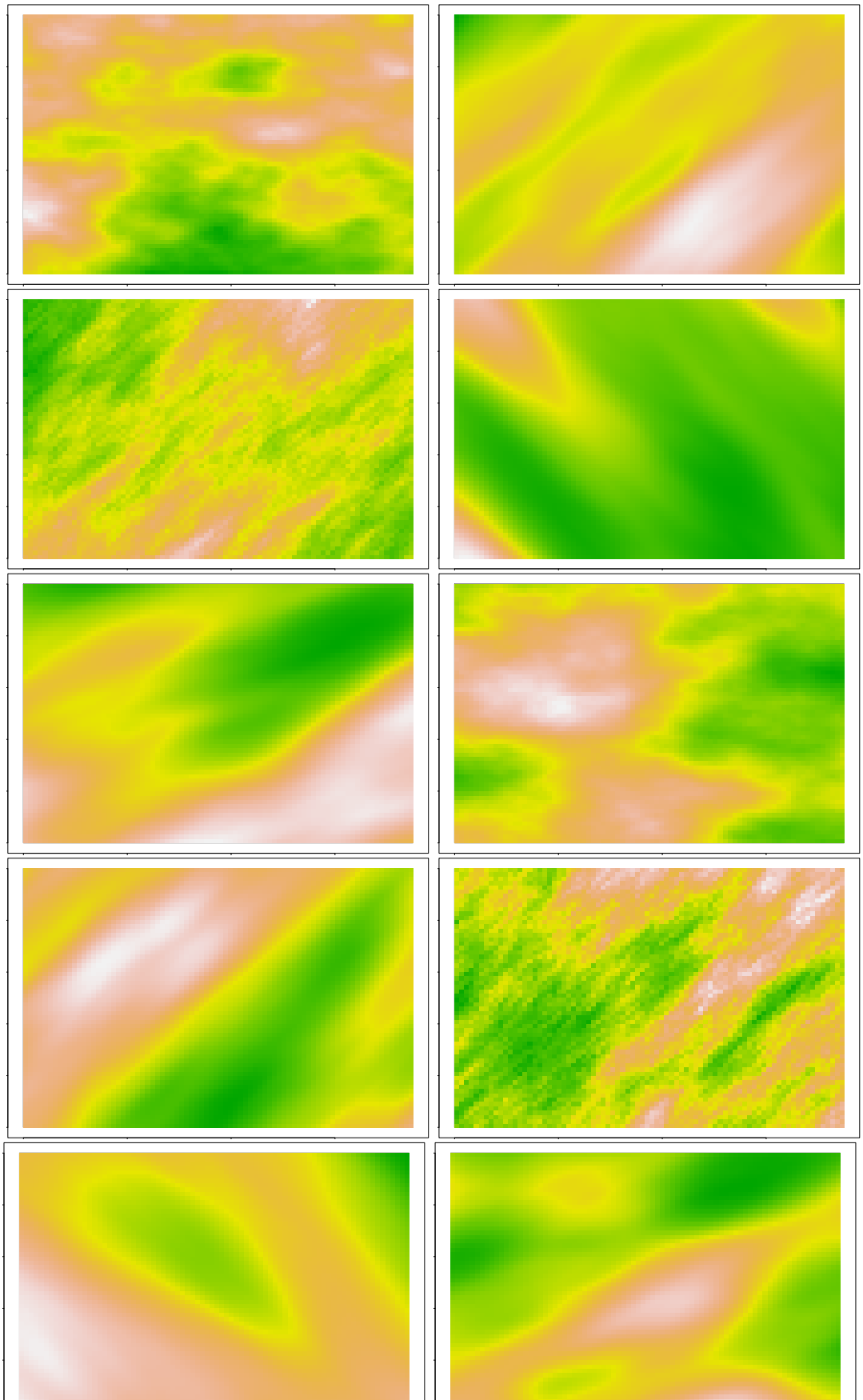
```
simMat = vclMatrix(0, nrow(zmatGpu), ncol(zmatGpu),
                   type = theType)

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 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 **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.

talk about using substreams in future?

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-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.

Davis MW (1987). "Production of conditional simulations via the LU triangular decomposition of the covariance matrix." *Mathematical geology*, **19**(2), 91–98.

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.

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/.

Fenton GA, Vanmarcke EH (1990). "Simulation of random fields via local average subdivision." *Journal of Engineering Mechanics*, **116**(8), 1733–1749.

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

Gómez-Hernández JJ, Journel AG (1993). "Joint sequential simulation of multigaussian fields." In *Geostatistics Troia'92*, pp. 85–94. Springer.

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.

Johnson ME (1987). *Multivariate statistical simulation: A guide to selecting and generating continuous multivariate distributions*, volume 192. John Wiley & Sons.

Journel AG (1974). "Geostatistics for conditional simulation of ore bodies." *Economic Geology*, **69**(5), 673–687.

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.

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

Loeve M (1978). *Probability theory. ii, volume 46 of. Graduate Texts in Mathematics*. 4 edition. Springer-Verlag New York.

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*.

Matheron G (1973). "The intrinsic random functions and their applications." *Advances in Applied Probability*, **5**(3), 439–468. doi:10.2307/1425829.

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

Mejía JM, Rodriguez-Iturbe I (1974). "On the synthesis of random field sampling from the spectrum: An application to the generation of hydrologic spatial processes." *Water Resources Research*, **10**, 705–711.

Oliver DS (1995). "Moving averages for Gaussian simulation in two and three dimensions." *Mathematical Geology*, **27**(8), 939–960.

Pebesma EJ (2004). "Multivariable geostatistics in S: the gstat package." *Computers & geosciences*, **30**(7), 683–691.

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.

Shinozuka M, Jan CM (1972). "Digital simulation of random processes and its applications." *Journal of Sound Vibration*, **25**(1), 111–128. doi:10.1016/0022-460X(72)90600-1.

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

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