A novice compendium to statistical GPU computing in R

Marco Y.S. Shum
mysshum@uwaterloo.ca
20170624

This is a short introductory compendium of notes on GPU (graphical processor unit) acceleration of statistical applications using R. The main purpose here is to expose the reader at a high level to the topic enough for him/her to explore it further. Except for the section involving C code, this guide assumes as a minimum that the reader is comfortable working with basic functionalities of R such as installing packages, loading a library, calling a function and performing basic algebra (such as using %*% for matrix multiplication).

This guide can be largely divided into two parts. The first part provides a high level introduction as well as a very brief survey of some applications of GPU computing in statistics. This is primarily meant to inspire the reader on ways the algorithms in their statistical procedures may be similarly (or otherwise creatively) implemented to leverage the advantages from a GPU. In the second part, a subset of GPU packages available on CRAN review. We introduce some common functionalities from the `gputools` package that provide GPU versions of useful functions found in the `base` package: these will be useful for users who wish to quickly prototype with functionalities such as `lm`, `dist` and `cor`. At a slightly lower level, we benchmark the performance of some packages on matrix multiplication and inversion. Finally, we touch upon the use of CUDA code (a flavour of the C language developed by NVIDIA to interface with GPUs.) The point here is to give those experienced with C the essential steps to write CUDA code and interface it with R so that more control over the code on the GPU can be had.

Connecting to a MFCF GPU machines

At the University of Waterloo, the MFCF group provides access to GPU computing resources. The server `gpu01` houses four NVIDIA Tesla K80 GPUs (information about this chipset may be found here,

`https://en.wikipedia.org/wiki/Nvidia_Tesla#Specifications_and_configurations`.

To log into this server, simply `ssh` into it using your Nexus credentials. Currently, one GPU machine running Linux is available to students and staff: it can be accessed on campus (e.g. on the Eduroam network) using the Nexus credentials by,

```
Shell> ssh nexusUserName@gpu01.student.math.uwaterloo.ca
```

where `Shell` is a Unix shell (Windows users may choose to use, for example, Putty to `ssh`.) When accessing from an off-campus location, a VPN client is necessary. Information about this resource can be found at, `https://uwaterloo.ca/math-faculty-computing-facility/services/specialty-research-linux-servers`.

GPU computing at Compute Canada

Compute Canada provides GPU computing resources for those who have an account with them. The machines are listed at the end of,

`https://www.computecanada.ca/research-portal/accessing-resources/available-resources/`

In particular, the Graham (GP3) machine is hosted at the University of Waterloo.

What is GPU computing?

GPU computing is a type of *high performance computing (HPC)* which employs a *graphical processing unit (GPU)* as the main hardware for performing computation, instead of the usual design of using *central processing units (CPU)*. Just as the CPU is a hardware component on a circuit board (the motherboard, in fact), a GPU is a chip that typically resides on a video card, the latter of which is also connected to the motherboard. Traditionally, video cards are found on personal computers that are used for graphically demanding tasks such as image or video processing and gaming applications.

Why GPU computing, instead of multicore CPU's?

Researchers found GPUs to be very suitable for tasks which can be parallelised (i.e. run simultaneously in one run of an application.) Compared to multicore CPU's and clusters of those, the parallel computing power obtained from the same budget of a GPU set up, when used properly, is typically greater. From a high level, the advantage over multicore CPU processors is due to the specialisation of the GPU chip design for high throughput: a GPU has a much larger number of cores than a CPU and so more parallel *threads* (roughly, unit of one task) may be performed on a GPU than on a CPU. However, proper usage of GPU typically involves careful planning: generally, the transfer of data from memory to CPU may be faster than to a GPU for such components that are introduced into the market at around the same time. Fortunately, there already exists `R` packages that implement GPU accelerated procedures for many common statistical usages. We give a general walkthrough of some of these packages below. However, many of these implementations are for specific functionalities: the reader may wish to move beyond this limitation by learning to write their own GPU code (two ways are to use the 'mid'-level GPU accelerated matrix operations benchmarked below or interface GPU-`C` code with `R` using the methods below for the more technically inclined.)

What statistical problems are good for GPU computing? A limited survey.

This small review serve as proof of concepts to inspire the reader into thinking about their problem from a parallelisation perspective. Some of these applications are in fact already implemented in `R` packages which we will either discuss or to which we will provide links in the 'Further resources' section.

A simple but prominent example is the parallelised sum of a map: to wit, for a finite index set, $i = 1, \ldots, M$, with maps $f_i : X \to \mathbb{R}$ and corresponding input x_i, the goal is to compute the sum,

$$\sum_{i=1}^{M} f_i(x_i).$$

There are two main places that the computation can be parallelised.

- Each $f_i(x_i)$ can be computed independently from one another, so can be done concurrently. For example, if one's hardware allows up to 1000 parallel tasks, then $M = 500000$ can ideally be done in 500 function calls of $f(x)$'s of each thread.
- The sum itself can be computed in parallel by, roughly, summing subsets of the summands at a time.

Monte Carlo methods with sampling such as the bootstrap [3] are readily parallelisable. Here, x_i is a bootstrap sample drawn from the original dataset and $f_i = f$ is the statistic in question. Importantly, note how this problem fits into the general paradigm of GPU computing: input data (i.e. the dataset) is loaded into the video card memory once, and each thread in the GPU simultaneously draws many bootstrap samples, x_i, and computes $f(x_i)$ before returning. This common paradigm is known as *single instruction, multiple data (SIMD)*: the sampled $\{x_i\}$ are independent of each other, and the same instruction (function f) is applied to each of them.

Many solutions that employ 'divide and conquer' strategies have GPU accelerated algorithms, and are beneficial to statistical applications as well.

For example, order statistics require sorting one's data set: sorting algorithms have been parallelised by researchers[12]. [14], in addition to providing a solution algorithm, also reviews previous research into GPU accelerated versions of radix, quick- and mergesort algorithms. [10] is a research collaboration involving NVIDIA researchers detailing implementations of sorting algorithms on GPU hardware.

Solutions involving iteration through pairs of data points have also been parallelised and GPU-accelerated. For example, [2] provides GPU algorithms for the Manhattan (l^1) distance between vectors of cDNA microarray data, as well as their Pearson's correlation. [8] also provides a method for parallelising the computation of Pearson's coefficient. Another example is the computation of the all-pairs shortest distance in a large graph (for example, [6] and [5].) All the examples exploit structures in the problem to reorganise the code from naïve for-loop constructs.

For example, consider Pearson's coefficient,

$$\frac{\mathrm{Cov}(x,y)}{\sqrt{\mathrm{Var}(x)\mathrm{Var}(y)}} := \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}.$$

Because the covariances are sample moments, such sums are readily parallelisable similarly to the Monte Carlo problems. As an extension, Spearman's coefficient is Pearson's coefficient on the ranks of the data,

$$\frac{\mathrm{Cov}(r_x, r_y)}{\sqrt{\mathrm{Var}(r_x)\mathrm{Var}(r_y)}}.$$

and so (at least naïvely) requires an additional step of ranking and sorting.

As a tangent on the use of ranking and sorting procedures, the computation of Kendall's coefficient between two data vectors, $x,\ y$, both of size m requires the classification of all $\begin{pmatrix} m \\ 2 \end{pmatrix}$ pairs $(x_i, y_i), (x_j, y_j)$ with $i \neq j$ into either concordant or discordant pairs. (A pair $\{x_i, y_i), (x_j, y_j)$ is said to be concordant if either $x_i < x_j$ and $y_i < y_j$ or $x_i > x_j$ and $y_i > y_j$, and discordant if either $x_i < x_j$ and $y_i > y_j$ or $x_i > x_j$ and $y_i < y_j$. A pair can be neither concordant or discordant, in which case the coefficient should be modified to take into account of such ties. See, for example, [1].) [9] investigate the parallelisation of the computation of Kendall's coefficient. Speed up is achieved largely due to different parallelisation of the underlying sorting procedure coupled with hardware specific improvements.

Work has also been done on incorporating GPU designs to optimisation problems: prominent statistical applications include model fitting and M-estimators (M stands for 'maximisation'.) Techniques to exploit of parallelisation may fall into two categories: parallelisation of evaluation of the functions and parallelisation of distributed algorithms. An example of the former is the evaluation of the score function of the form,

$$\sum_{i=1}^{m} \nabla_\theta \log L(x_i; \theta),$$

which is the gradient of the log-likelihood function $\log L$ over parameters θ given a dataset x. Once again, we see the SIMD-sum theme occurring: for a fixed θ (say, the previous step of a gradient descent), the evaluation of either the individual summands or the sum itself can be parallelised. An example of the latter is the parallelisation of swarm optimisation. In this framework, many candidate solutions are moved around the domain of the function in search of the optimum. At every step, all candidates report their function values: based on this collective information, the step of each candidate is adaptively determined in hopes that eventually this 'swarm' will all converge to a (global) optimum. An application to the particle swarm optimisation algorithm can be found in [15]. Generally, the sum of log-likelihoods are typically trivially parallelisable whereas the summands themselves require that the log-likelihood has a form that is amenable to parallelisation.

In Bayesian computations, there have been research into GPU accelerated sampling, broadly falling into the categories of Monte Carlo procedures with i.i.d sampling and Markov Chain sampling, MCMC. The common theme in the latter is to work with parametrisations and representations of the Bayesian network that allow either a large number of random variables to be sampled in parallel or to parallelise within a single sampling of one variable. An example of the former are mixture models whose latent variables and their corresponding observations may be sampled independently. An example of the latter is the use of computationally expensive univariate samplers in a Gibbs' sampling scheme. A common instance of this is the Metropolis-within-Gibbs scheme, where a Metropolis-Hastings MCMC is run for some or all of the univariate sampling from the conditional distributions. For large datasets, the evaluation of the posterior likelihood may be very expensive, but, as pointed out above, the product of an i.i.d sample likelihood can be parallelised via the log-likelihood sum computation. [11] provides the `cudaBayesreg` package for `R` for Bayesian hierarchical modelling, specialised to fMRI data. [13] provides an excellent exposition of GPU considerations in MCMC computations. [7] provides approximate Bayesian computation (ABC) methods in systems biology using GPU hardware: even though it is in `python`, the concept and methodology are nevertheless valuable.

GPU accelerated common statistical functions package

We single out the `R` package `gputools` as one of the more accessible packages for GPU computing because it wraps many common functionality of the base `R` package with a friendly syntax. The following are some examples directly from the manual [4] (load the library with `library(gputools)`.)

Linear models with `lm` on GPU:

```
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
```

```
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
anova(lm.D9 <- gpuLm(weight ~ group))
summary(lm.D90 <- gpuLm(weight ~ group - 1))# omitting intercept
summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical
```

Correlation coefficients with `cor` on GPU:

```
numAvars <- 5
numBvars <- 10
numSamples <- 30
A <- matrix(runif(numAvars*numSamples), numSamples, numAvars)
B <- matrix(runif(numBvars*numSamples), numSamples, numBvars)
gpuCor(A, B, method="pearson")
gpuCor(A, B, method="kendall")
A[3,2] <- NA
gpuCor(A, B, use="pairwise.complete.obs", method='pearson')
```

Distance calculation with `dist` on GPU:

```
numVectors <- 5
dimension <- 10
Vectors <- matrix(runif(numVectors*dimension), numVectors, dimension)
gpuDist(Vectors, "euclidean")
gpuDist(Vectors, "maximum")
gpuDist(Vectors, "manhattan")
gpuDist(Vectors, "minkowski", 4)
```

There are many more statistical procedures in `gputools` which may be called directly as those listed: interested readers may want to explore the list here.

Speed comparison of GPU matrix multiplication and inverse on R

For the user new to GPU computing, it can be intimidating to wade through the large amount of GPU (`R`) package documentation on the web. Many of them intersect on similar functionalities and their code implementation details are sometimes obscure in description. In particular, unless published, we found it difficult to ascertain how the code was GPU-accelerated, by how much and the reason for the speed up from the manual. Finally, it is realistically rare that the typical end user has the time or knowledge to fine-tune the packages themselves. Here, we run some benchmarks from an end-user perspective by replicating what is done in the `R` manual and scaling up the data size: no configurations are tuned beyond what is done by `install.packages`. Because matrix multiplication and inversion are fundamental operations in statistics (as well as in other fields that make use of `R`), we produce these benchmarks using large dense matrices.

The packages we compare are the `R base`, `gputools`, `gpuR` and `gmatrix`. All tests are run on a single NVIDIA Tesla P-100-PCIe GPU card. The package `microbenchmark` used to time the runs. The total time used includes the time used to create the matrices on the GPU memory. The data matrices are shared by all the tests. For a fixed m, we generate $A, B \in \mathbb{R}^{m \times m}$ by drawing their elements independently from $\mathrm{Unif}([0,1])$. $A^T A$ is also computed to obtain a positive definite matrix. All such data generation are done on CPU and not timed on GPU. The matrix multiplication operation is tested by computing AB for $m \in \{100, 1000, 2500, 5000, 7500, 10000\}$ and matrix inversion is tested by computing $(A^T A)^{-1}$ for $m \in \{100, 1000, 2500, 5000, 7500\}$. We measure the minimum, maximum and 25, 50, and 75-th quantiles of the base 10 logarithm of the number of seconds to perform the operation for the maximum number of simultaneous runs allowed by the memory of the GPU, capped at 100. We were unable to run the matrix inverse test for `gputools` at the time of writing, and so is omitted there.

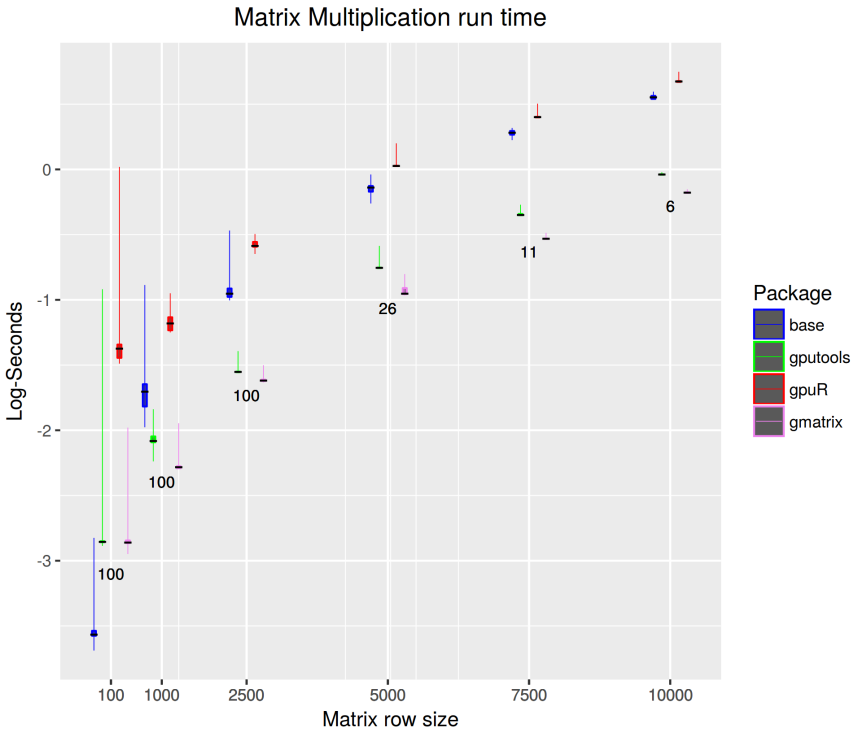

Figure 1: Base-10 logarithm of run times in seconds for matrix multiplication of two random square matrices. The box plots show the minimum, 25-th, 50-th, 75-th quantiles and the maximum $\log_{10}$ run time of a set number of independent benchmarking runs (displayed below each set of points for each matrix row size.)

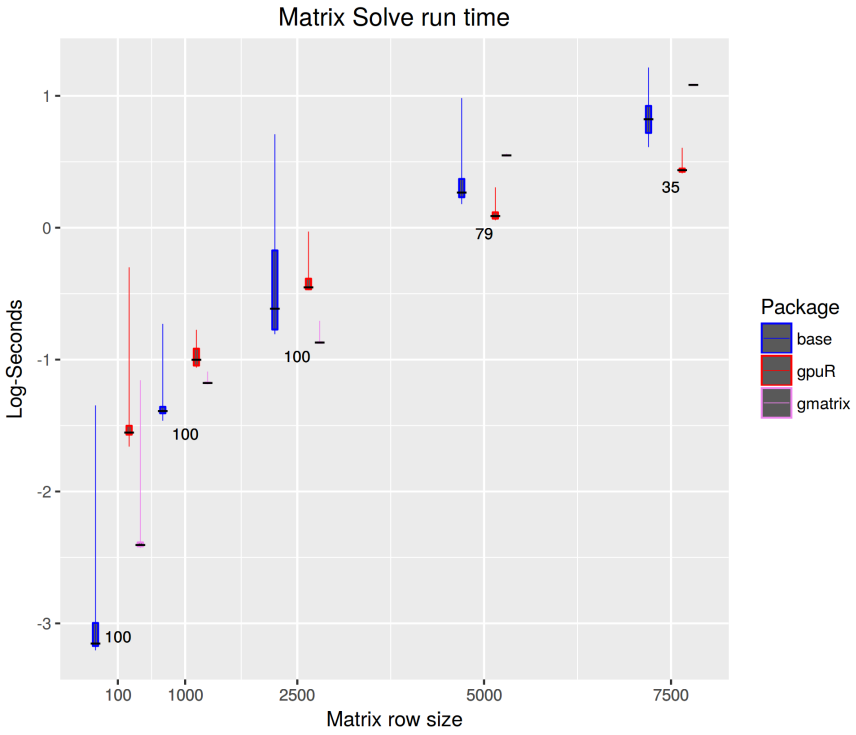

Figure 2: Base-10 logarithm of run times in seconds for matrix inversion of a random positive symmetric matrix. The box plots show the minimum, 25-th, 50-th, 75-th quantiles and the maximum $\log_{10}$ run time of a set number of independent benchmarking runs (displayed below each set of points for each matrix row size.)

We can see that, for our particular test cases, the `gmatrix` package grants (the most) significant speed up for both operations. At $m = 10000$, the median speed up between the base multiplication and the `gmatrix` multiplication is *strictly* more than $10^{\frac{1}{2}} \approx 3$ times or 300%. On the other hand, for $m = 2500$, the median of the speed up on the matrix inversion operation is somewhat negligible.

Interestingly, the package `GpuR` provides the most speed up for the matrix inverse experiment, while `gmatrix` seems to not scale well with the matrix size. We also note the larger 25-75% range of the `base` package run time versus both GPU packages. Particularly, the `GpuR` package seems to yield a speed up almost all the time, and its performance is possibly more robust against other factors that might influence the run time.

In general, we observe that computation on GPU is only suitable for large matrices whereas the `base` package on the CPU performs faster for smaller matrices. This is because any GPU procedure typically incurs extra run time to first load its data on to the GPU memory before its massive processing power can be leveraged.

Potential caveat: the `microbenchmark` package used here is agnostic to the GPU architecture. Using this package, the individual runs for a single package is run simultaneously on the GPU. It is yet unclear how this may actually affect the performance of certain packages, especially without knowing how the `R` code is implemented. `microbenchmark` was chosen as it is typically an accurate timer at the `C` level of any `R` code being run. In light of this, we treat the above benchmarks as conservative ones.

Note: as far as the author can tell, there are no other steps that are needed to set up packages tested on `R` outside of the instructions provided by the manual on CRAN and the steps prompted by `install.packages`. If any reader can point out any possible additional set up steps needed from an end-user perspective, the author would be happy to try them out and update this benchmarking.

`R` with CUDA: getting your hands dirty

Those experienced with `C` may opt to implement the algorithm in `C` directly. One reason for this is to have more control

over aspects of the execution (two prominent concerns being memory management and the number of parallel process to start.) Another reason may be that CRAN does not have a package suitable for one's application. Interestingly, to the best of the author's knowledge, there is no GPU accelerated versions of the popular `apply` function.

CUDA is an extension of `C` from NVIDIA to allow the user to interface with the GPU at the `C` level. We will not provide a tutorial on the CUDA language. Instead, we outline the salient points in an example that are different from the usual `C` language. Our discussion focuses on using a Linux operating system (as on many of the servers offered by MFCF as well as the BSD back end of MacOS.) The steps are analogous on Windows.

At a high level, `R` is able to load `.so` (`.dll` on Windows) library files using the `dyn.load` function, and `.C` function is used to call any functions compiled and linked in the library. We will therefore compile CUDA source code into these formats.

At the coding level, GPU coding with CUDA typically follows this general programming procedure.

- Create or store data on the CPU memory using `C`.
- Allocate on the GPU memory and copy data from CPU memory to it.
- Allocate any necessary memory and perform the parallel tasks on the GPU.
- Allocate on CPU memory and copy results of the parallel tasks from GPU memory to it.
- Process results using CPU as per usual `C` code.

We demonstrate this procedure with an example. Suppose we wish to approximate the integral,

$$I(a,b) := \int_a^b e^{-\frac{u^2}{2}} du = (b-a)E[e^{-\frac{U^2}{2}}],$$

where the expectation on the right is taken over $U \sim \mathrm{Unif}(a,b)$. Then, for any $a < b$, we can, for a large M, use the sample mean estimate on the left of the following that satisfies the law of large numbers,

$$\widehat{I}(a,b;M) := \frac{(b-a)}{M} \sum_{i=1}^{M} e^{-\frac{U_i^2}{2}} \xrightarrow{\text{a.s.}} (b-a)E[e^{-\frac{U^2}{2}}].$$

This sampling algorithm can be put into the above steps as follows.

- Create or store the random seed of the simulations on the CPU memory.
- Allocate GPU memory and write the seeds to it.
- Allocate memory on GPU for $\{e^{-\frac{U_i^2}{2}} : i = 1, \ldots, M\}$, and compute them.
- Allocate on CPU memory and copy results of the parallel tasks from GPU memory to it.
- Compute the mean of the results on CPU.

The `CUDA` and `R` code listings for this examples are in the appendix. The `CUDA` function `mcIntegrationGaussianDensity` is the main work horse of the steps of our procedure, and is called through `.C` in `R`. For example, to approximate $I(-4,4)$ using $\widehat{I}(-4,4;1024000)$, we perform the following `R` function call,

```
R> mcIntegrationCUDA(-4.0, 4.0, 1024000)
```

We will not discuss the memory management in `CUDA` here. We simply note that `CUDA` allows one to organise threads (in our case, one per Monte Carlo sampling) into *blocks*: such organisation is sometimes key to speed ups gained in GPU computing. In our example, we empirically found that dividing $M = 1024000$ Monte Carlo runs into 1000 blocks of 1024 threads yields fairly significant speed up against native `R` code. `microbenchmark` was used to time 100 independent runs of both algorithms:

As readily expected, the `CUDA/C` implementation out-paces the native `R` one with `sapply` very significantly. A single run of the CUDA code produced a value for $\widehat{I}(-4,4;1024000)$ of,

```
[1] 2.487866
```

Method	Min	Q25	Mean	Q50	Q75	Max
CUDA from R	6.177006 ms	6.292781 ms	13.03467 ms	6.308993 ms	6.327924 ms	649.695 ms
Native R	3.120825 s	3.1628 s	3.252945 s	3.240894 s	3.30436 s	4.12569 s

Figure 3: Statistics of run times computing $\widehat{I}(-4, 4, 1024000)$ between the custom CUDA implementation from R and native R using sapply. Each method was run independently for 100 times using microbenchmark.

while the native R code with sapply produced,

```
[1] 2.508225
```

Further resources

We have only reviewed a very small subset of functionalities of the packages herein are reviewed. The reader may therefore find their documentations and their originating academic publications to be useful for learning about the packages further.

- gputools:
  https://cran.r-project.org/web/packages/gputools/index.html (CRAN)
  https://academic.oup.com/bioinformatics/article/26/1/134/181997 (Academic source)
- gmatrix:
  https://cran.r-project.org/web/packages/gmatrix/index.html (CRAN)
- OpenCl
  https://cran.r-project.org/web/packages/OpenCL/index.html

The following resources may be useful for applications not covered in this document.

- *The CRAN HPC task list (contains a list of GPU supported packages, as well as a general list of non-GPU but HPC related packages for R):*
  https://cran.r-project.org/web/views/HighPerformanceComputing.html
- gpuR (similar to gmatrix)
  https://cran.r-project.org/web/packages/gpuR/index.html (CRAN)
  https://cran.r-project.org/web/packages/gpuR/vignettes/gpuR.pdf (Vignette)
- rpud package (removed from CRAN): similar toolset to gputools.
  http://www.r-tutor.com/content/download
- WideLM package (on MetaCRAN): uses GPU to fit many linear models in parallel to a single dataset.
  https://www.r-pkg.org/pkg/WideLM
- cudaBayesreg: a package for performing Bayesian hierarchical modelling (specialised to fMRI data.)
  https://cran.r-project.org/web/packages/cudaBayesreg/index.html
  https://www.researchgate.net/publication/265483496_cudaBayesreg_Bayesian_Computation_in_CUDA
- RCuda package: similar to OpenCl, provides a way to incorporate CUDA code in R.
  http://www.omegahat.net/RCUDA/
  http://www.omegahat.net/RCUDA/RCUDA.pdf
- An NVIDIA blog post on GPU computing with respect to R.
  https://devblogs.nvidia.com/parallelforall/accelerate-r-applications-cuda/
- Rth: a package based upon Thrust that provides functionalities accelerated by both GPU and multicore CPU.
  http://heather.cs.ucdavis.edu/$\sim$matloff/rth.html
  https://matloff.wordpress.com/2014/06/17/rth-a-flexible-parallel-computation-package-for-r/

Appendix: CUDA and R code for Monte Carlo integration example

In R:

```
#############################################################################################################
#
# Monte Carlo integration by calling custom CUDA library.
#
# M.Y.S. Shum
# 20170620
#
#############################################################################################################

library(microbenchmark)

# Load library
LIBRARY_PATH <- '../cuda/'
dyn.load(paste0(LIBRARY_PATH, 'McIntegration.so'))

# Conveient wrapper to call functions.
mcIntegrationCUDA <- function(a, b, num_mc, verbose=FALSE)
{
  #
  # Computes the integral
  #
   # I = int_a^b e**(-0.5 * u ** 2) du,
  #
  # using CUDA function from McIntegration.so.
  #

  num_threads_per_block <- 1024
  num_blocks <- ceiling(num_mc / num_threads_per_block)

  print(paste0('num_blocks: ', num_blocks))
  print(paste0('num_threads_per_block: ', num_threads_per_block))

  res <- .C("mcIntegrationGaussianDensity",
      as.double(a),
      as.double(b),
      as.integer(num_mc),
      as.integer(num_blocks),
      as.integer(num_threads_per_block),
      as.logical(verbose),
      result=double(length=1))

  return(res[[7]])
}

#
# Test runs
#
a <- -4.0
b <- 4.0
num_mc <- 1000 * 1024

# Do benchmark.
print(microbenchmark(mcIntegrationCUDA(a, b, num_mc)))
print(microbenchmark((b-a) * mean(sapply(1:num_mc, function(i) exp(-0.5 * runif(1,a,b) ** 2)))))

# Print a test run.
print((mcIntegrationCUDA(a,b,num_mc)))
print(((b-a) * mean(sapply(1:num_mc, function(i) exp(-0.5 * runif(1,a,b) ** 2)))))
```

In CUDA/C:

```
//////////////////////////////////////////////////////////////////////////////////////////////////////
//
// An example of Monte Carlo integration using CUDA.
//
// M.Y.S. Shum
// 20170620
//
// On MFCF machines, compile with,
//
// nvcc -G -L /usr/lib/R/lib/ -I /usr/share/R/include/ -I /usr/local/cuda-8.0/include -lR
//      --shared -o McIntegration.so -Xcompiler -fPIC /McIntegration.cu
//
//////////////////////////////////////////////////////////////////////////////////////////////////////

#include <unistd.h>
#include <stdio.h>
#include <math.h>

#include <R.h>

#include <curand.h>
#include <curand_kernel.h>

#define MY_PI 3.14159265358979323846264338327950288
```

```
__global__ void rand_init(unsigned int seed, curandState_t* states)
{
  //
  // Initialises the pseudo-random number generator for each thread. Threads will use the same seed,
  // but, by enumerating through the thread index, it will produce different set sequences of random
  // numbers.
  //
  unsigned int thread_ind = blockIdx.x * blockDim.x + threadIdx.x;
  curand_init(seed, thread_ind, 0, &states[thread_ind]);
}
__global__ void sampleAndTransform(curandState_t* states, double* output, const double a, const double b)
{

  //
  // Method to
  //
  // 1. sample u ~ Unif(a,b) and
  // 2. evaluate exp(-0.5 * u **2).
  //

  unsigned int thread_ind = blockIdx.x * blockDim.x + threadIdx.x;
  double u = (b - a) * curand_uniform(&states[blockIdx.x]) + a;
  output[thread_ind] = exp(-0.5f * pow(u,2));
}
extern "C" void mcIntegrationGaussianDensity(double* a, double* b,
                                             int* num_mc, int* num_blocks, int* num_threads_per_block,
                                             bool* verbose,
                                             double* result)
{

  //
  // Method to estimate the integral
  //
  // I = int_a^b e**(-0.5 * u ** 2) du.
  //
  // We note that I = (b-a) * Expectation(e**(-0.5 * u ** 2)) with u ~ Unif(a,b), and use
  // Monte Carlo sampling to estimate this.
  //


  // Initialise states in each parallel process.
  if(*verbose) { printf("Initialising randomness state on GPU.\n"); }
  curandState_t* states;
  cudaMalloc((void**) &states, *num_mc * sizeof(curandState_t));
  rand_init<<<*num_blocks, *num_threads_per_block >>>(time(0), states);
  cudaDeviceSynchronize();

  // Allocate memory for storing integrand.
  if(*verbose) { printf("Initialising GPU memory.\n"); }
  double cpu_integrands[*num_mc];
  double* gpu_integrands;
  cudaMalloc((void**) &gpu_integrands, *num_mc * sizeof(double));
  cudaDeviceSynchronize();

  // Compute integrands.
  if(*verbose) { printf("Computing integrands.\n"); }
  sampleAndTransform<<<*num_blocks, *num_threads_per_block >>>(states, gpu_integrands, *a, *b);

  // Write result from GPU to CPU memory.
  if(*verbose) { printf("Writing to CPU.\n");}
  cudaMemcpy(cpu_integrands, gpu_integrands, *num_mc * sizeof(double), cudaMemcpyDeviceToHost);

  // Take the average of the returned integrands and normalise.
  if(*verbose) { printf("Taking average on CPU\n"); }
  double expectation = 0.0;
  for (unsigned int i=0; i<*num_mc; ++i) { expectation += cpu_integrands[i]; }
  expectation /= *num_mc;

  // Compute integral.
  if(*verbose) { printf("Computing integral\n"); }
  double integral = ((*b-*a) * expectation);

  // Information
  if(*verbose) {printf("Integral= %f.\n", integral ); }
  if(*verbose) {printf("Normalised density? %f.\n", integral / (sqrt(2.0 * MY_PI) )); }

  // Write to result.
  *result = integral;

  // Clean up: free memory on GPU.
  cudaFree(states);
  cudaFree(gpu_integrands);
}
```

Bibliography

[1] Alan Agresti and Maria Kateri. *Categorical data analysis*. Springer, 2011.

[2] Dar-Jen Chang et al. "Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu". In: *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on*. IEEE. 2009, pp. 501–506.

[3] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[4] *Gputools on CRAN*. `https://cran.r-project.org/web/packages/gputools/gputools.pdf`.

[5] Pawan Harish and PJ Narayanan. "Accelerating large graph algorithms on the GPU using CUDA". In: *International Conference on High-Performance Computing*. Springer. 2007, pp. 197–208.

[6] Gary J Katz and Joseph T Kider Jr. "All-pairs shortest-paths for large graphs on the GPU". In: *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Eurographics Association. 2008, pp. 47–55.

[7] Juliane Liepe et al. "ABC-SysBioapproximate Bayesian computation in Python with GPU support". In: *Bioinformatics* 26.14 (2010), pp. 1797–1799.

[8] Yongchao Liu, Tony Pan, and Srinivas Aluru. "Parallel pairwise correlation computation on intel xeon phi clusters". In: *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*. IEEE. 2016, pp. 141–149.

[9] Yongchao Liu et al. "Parallelized Kendall's Tau Coefficient Computation via SIMD Vectorized Sorting On Many-Integrated-Core Processors". In: *arXiv preprint arXiv:1704.03767* (2017).

[10] Nadathur Satish, Mark Harris, and Michael Garland. "Designing efficient sorting algorithms for manycore GPUs". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–10.

[11] Adelino Ferreira da Silva. "cudaBayesreg: Bayesian computation in CUDA". In: *The R Journal* 2.2 (2010), pp. 48–55.

[12] Erik Sintorn and Ulf Assarsson. "Fast parallel GPU-sorting using a hybrid algorithm". In: *Journal of Parallel and Distributed Computing* 68.10 (2008), pp. 1381–1388.

[13] Marc A Suchard et al. "Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures". In: *Journal of computational and graphical statistics* 19.2 (2010), pp. 419–438.

[14] Xiaochun Ye et al. "High performance comparison-based sorting algorithm on many-core GPUs". In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–10.

[15] You Zhou and Ying Tan. "GPU-based parallel particle swarm optimization". In: *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE. 2009, pp. 1493–1500.