Unleashing GPU Power Using R: The gmatrix Package

Nathan Morris
Case Western Reserve University

Abstract

In recent years, graphical processing units (GPUs) have become increasingly flexible in allowing the implementation of general purpose algorithms. Thus, given the massively parallel nature of GPUs and their excellent performance to power ratio, many large data centers are opting to install clusters with large numbers of NVIDIA®GPUs. Several very nice R projects such as **gputools** have made use of GPU power to speed up certain specific operations on the GPU. The **gmatrix** package follows a different path, giving users the tools to implement a broad range of algorithms in R for themselves. The package allows for easy management of the separate device and host memory spaces. Numerous numerical operations are implemented for these objects on the GPU. These operations include matrix multiplication, addition, subtraction, the Kronecker product, the outer product, comparison operators, logical operators, trigonometric functions, indexing, sorting, random number generation and many more. Currently, **gmatrix** can only be installed on Linux machines with an NVIDIA®GPU.

Keywords: GPU, CUDA, graphics processing unit, R.

1. Introduction

GPUs, which were originally designed to perform graphical rendering, are highly parallel devices designed to perform large numbers of simple operations in parallel. The large general market for GPUs has allowed manufacturers to push down prices while creating increasingly high performance devices which arguably draw relatively lower amounts of power than a similar central processing unit (CPU). Realizing the potential of such devices to be productive in fields outside of graphics, GPU manufacturers have increasingly attempted to create tools which will allow general purpose computing algorithms to be implemented on the GPU. One of the most influential of these tools is a programing model known as the "compute unified device architecture" (CUDA). This programming model extends the C language to allow programmers to create small functions known as kernels which run on the GPU. Unfortunately, implementing a large complex algorithm with CUDA is not always straightforward, and therefore numerous libraries implementing tasks such as linear algebra operations and random number generation have been created.

It is important to understand from the outset the limitations of GPUs. First, GPUs have their own memory space which is separate from the main memory of the computer. A GPU process does not directly access objects in the main memory. Objects in the main memory must first be moved to GPU memory. In the high performance computing setting, typically

the GPU memory will be significantly smaller than the main memory. Hence, it is important to be able to manage the GPU memory effectively. Second, there is often a high overhead to be paid for each GPU operation (i.e., kernel) that is launched. Hence, it is generally only worthwhile to perform an operation on the GPU if that operation is running quite slowly on the CPU. Third, GPUs are designed to run only relatively simple kernels. There is very little hope of actually getting R to run directly in a GPU kernel. Finally, GPUs typically have much higher single precision floating point capabilities than double precision floating point capabilities.

A number of R packages have already been produced which make use of GPU power. The packages **gputools** (J Buckner and Wilson 2013) and **cudaBayesreg** (da Silva 2011) implement a number of specific applications such as linear regression, support vector machines and Bayesian regression. While these packages may do very well at these specific applications, they are not intended to enable the user to create more general algorithms. The packages **magma** (Smith 2013), **HiPLARM** (Nash and Szeremi 2012) and **HiPLARb** (Szeremi 2012) provide access to matrix factorization and multiplication on the GPU. **HiPLARb** replaces some of the **base** linear algebra functionality in R, while **HiPLARM** replaces some of the functionality of the **Matrix** (Bates and Maechler 2013) package. Note that the **base** and **Matrix** packages provide only serial implementations of linear algebra routines, while **magma**, **HiPLARM** and **HiPLARb** provide much faster parallel / GPU versions of these algorithms. *However, these GPU based packages do not store the actual data on the GPU*, and any operations which use the GPU must involve the additional overhead of transferring data to and from the GPU for every operation. Also, these packages perform standard matrix operations such as matrix addition or element wise multiplication on the CPU instead of the GPU. Furthermore, these packages do *not* perform random number generation, indexing, sorting, or special functions (e.g., `log()` or `sin()`) on the GPU. Perhaps the package that motivated **gmatrix** most strongly was **rgpu** (Kempenaar and Dijkstra 2013), which allows objects to be transferred back and forth between the GPU and main memory. The primary innovation of our approach, however, has been to implement S4 classes which store an external pointer to data on the GPU. The `gmatrix` and `gvector` classes behave very similarly to standard R `matrix` and `vector` objects. Thus, many algorithms in R may be transferred transparently to the GPU with minimal effort. In this respect, **gmatrix** is perhaps somewhat comparable to the ®Matlab plug-in known as **Jacket**. Therefore, **gmatrix** differs from most previous GPU based R packages both in its generality and its approach to managing the separate GPU and host memory spaces. It should be noted that **gmatrix** does not perform matrix factorization which **HiPLARM** and **HiPLARb** excel at. The **rcula** (Morris 2013) package builds on the **gmatrix** package to do some simple matrix factorization.

2. The `gmatrix` and `gvector` class

The `gmatrix` and `gvector` S4 classes store matrix and vector objects on the GPU. Every `gmatrix` and `gvector` object has an associated type (i.e., `"double"`, `"single"`, `"integer"` or `"logical"`) which is stored in the `type` slot. Other slots for the class store the dimensions/length of the object, the names/rownames/colnames of the object and, most importantly, a pointer to the object in the GPU memory. An R `matrix` or `vector` may be moved to the GPU using the command `g()` and moved back to the main memory with the command `h()`. One may also use the numerous coercion functions such as `as.gmatrix()` and

`as.matrix()` to move data on and off the GPU. Also, numerous approaches to construction information on the GPU exist. In general, these function start with "`g`" and mimic the **base** R functions as displayed in Table 1. For example, `gmatrix()`, `g.rep()`, `gseq()` and `%to%` mimic respectively the R **base** functionality: `matrix()`, `rep()`, `seq()` and `:`. Consider, for example, the following lines of code, all of which produce the equivalent matrices, but which may differ in execution time.

```
R>  A <- g( matrix(1:100, 10, 10) )
R>  A <- gmatrix(1:100, 10, 10)
R>  A <- gmatrix(1 %to% 100, 10, 10)
```

It is crucial to understand that the `gmatrix` / `gvector` object only stores an external pointer. This has some advantages and some disadvantages. Perhaps the chief disadvantage is the unexpected behavior of a command such as `B <- A`. Such a command duplicates the pointer without duplicating the actual object on the GPU. Thus, changes to the object B (e.g., B[1,1] <- 1000) would change both objects `A` and `B`. One should instead use the command `B <- gdup(A)`. The `gdup()` function duplicates an object on the GPU and returns a pointer to the duplicated object. A similar problem occurs when modifying an object within a function call. For example:

```
R>  f <- function(x) { x[1,1] <- 10; return(TRUE)}
R>  f(A)
```

R creates a separate copy of x when it is modified. However, the separate copy of `x` still points to the same memory on the GPU as `A`, and so the original input copy is modified. This can again be easily fixed by modifying the function: `f <- function(x) {x <- gdup(x); x[1,1] <- 10; return(TRUE)}`. While the use of an external pointer does create some problems, it also opens up some interesting avenues for code optimization. For example, the command `A <- gmatrix(1 %to% 100, 10, 10)` performs an entirely unnecessary duplication of the input data because the input data cannot be used or modified elsewhere in the code. The command `A <- gmatrix(1 %to% 100, 10, 10, dup=FALSE)` uses the external pointer nature of the object to avoid this unneeded duplication. Because GPU memory is precious and often the data is quite large, the ability to explicitly avoid such duplications is something to be appreciated.

A related point is that, because R is only aware of the pointer, R does not know when the GPU memory is full, and occasionally a failure to adequately perform garbage collection can lead to memory full errors. The command `ggc()` may be used to perform garbage collection and report on the amount of available memory on the GPU.

The type of an object can be accessed or set using the `type` and `type<-` functions. For example, to coerce the type of `A` to `"single"`, the command `type(A) <- "s"` or alternatively `type(A) <- "single"` command may be issued. In general, `"double"` is equivalent to `"d"`, `"single"` is equivalent to `"s"`, `"integer"` is equivalent to `"i"`, and `"logical"` is equivalent to `"l"`. Numerous operations on the GPU behave much the same as on standard R objects. *By default, all operations which involve a GPU object are performed on the GPU and return a GPU object with an appropriate type.* For example, logical operators (i.e., `&` and `|`) involving at least one GPU object return a GPU object with a `"logical"` type. Most function/operations in the package attempt to coerce the type of the operands/inputs to something reasonable for the given operation.

Elementwise binary operations in R may be performed for two objects of different length. In this case, the operation recycles the elements of the smaller object. This works with GPU objects as well. For example, `(1 %to% 10 + 1 %to% 2)` and `(1:10 + 1:2)` will yield the same results with one object on the GPU and one object on the main memory. As mentioned previously, object are returned by default on the GPU. For example, `(1 %to% 10) * 100` involves the operation (*) with the operand `100` residing in main memory, and yet the final result of the operation will be an object of class `gvector`. The binary operators implemented include: `"+"`, `"-"`, `"*"`, `"/"`, `"%%"`, `"^"`, `"=="`, `"!="`, `"<="`, `">="`, `"<"`, `">"`, `"&"` and `"|"`. In addition, numerous unary elementwise functions/operators have been implemented. These include: `"!"`, `abs()`, `exp()`, `expm1()`, `log()`, `log2()`, `log10()`, `log1p()`, `sin()`, `cos()`,]`tan()`, `asin()`, `acos()`, `atan()`, `sinh()`, `cosh()`, `tanh()`, `asinh()`, `acosh()`, `atanh()`, `abs()`, `lgamma()`, `gamma()`, `sign()`, `round()`, `ceiling()`, `is.na()`, `is.nan()`, `is.finite()`, and `is.infinite()`.

The matrix multiplication functionality in **gmatrix** makes use of the **CUBLAS** library supplied by NVIDIA®CUDA **Toolkit**. We have also implemented other common matrix algebra operations such as the Kronecker product (`%x%`), outer product(`%o%, gouter()`), the `diag()` function, `diag<-` function, transpose (`t()`) and a function for multiplying a matrix times the diagonal of a vector (`gmatTimesDiagVec()`). We have used the **Thrust** library (Hoberock and Bell 2010) to implement reduction operations such as: `sum()`, `mean()`, `max()` and `min()`. Other reduction operations such as `rowSums()`, `colSums()`, `rowMeans()` and `colMeans()` were implemented using the **CUBLAS** library. We also implemented the R functions `which()` and `ifelse()` for GPU objects. The function `sort` and `gorder` were implemented using the **Thrust** library and behave very similarly to the standard R implementation of these functions. However, unlike the standard R implementation, `gorder` creates a permutation for only one vector and cannot handle multiple vectors.

Indexing operations similar to the ones performed on standard R objects have been implemented. For example, one can use the command `A[1:3,3]` to extract the first three elements of column 3 from the `gmatrix` object `A`. Or, similarly, `A[1:3,3] <- 10` may be used to set the specified elements to `10`. The statement `A[-10,]` will return `A` with row `10` removed. The form of indexing which uses a numeric matrix with one column for each dimension has not been implemented.

3. Distributions and random number generation

The early versions of **gmatrix** were very useful in our own implementation of Bayesian MCMC schemes with big data, and we believe that MCMC will be one of the primary places where our package will find usefulness. We have implemented code to make use of the random number generators in the **cuRAND** library. **cuRAND** can simulate random variables from the uniform and normal distributions, and we have created the functionality to simulate from gamma, Poisson and binomial distributions. We simulate from the gamma distribution using the approach of Marsaglia and Tsang (2000). The Poisson and binomial distributions are simulated using a modified version of the approach in the **GSL** library which it attributes to Knuth. The **gmatrix** version of random number generators may be found by putting a "g"

R Function	gmatrix Function	Notes
`matrix()`	`gmatrix()`	Create a `matrix` / `gmatrix` from a `vector` / `gvector`
`rep()`	`g.rep()`	Create a `vector` /`gvector` by replicating a smaller `vector` / `gvector`
`seq()`, `:`	`gseq()`, `%to%`	Create a sequence of numbers
`rnorm()`	`grnorm()`	Sample from a normal distribution
`dnorm()`	`gdnorm()`	Density function for a normal distribution
`rgamma()`	`grgamma()`	Sample from a gamma distribution
`dgamma()`	`gdgamma()`	Density function for a gamma distribution
`rbeta()`	`grbeta()`	Sample from a beta distribution
`dbeta()`	`gdbeta()`	Density function for a beta distribution
`runif()`	`grunif()`	Sample from a uniform distribution
`dunif()`	`gdunif()`	Density function for a uniform distribution
`rpois()`	`grpois()`	Sample from a Poisson distribution
`dpois()`	`gdpois()`	Density function for Poisson distribution
`rbinom()`	`grbinom()`	Sample from a binomial distribution
`dbinom()`	`gdbinom()`	Density function for binomial distribution
`gc()`	`ggc()`	Garbage collection on the GPU

Table 1: R **base** vs. **gmatrix** functions

in front of the R command. For example, `grnorm()` is the GPU version of `rnorm()`. The following code will create a set of random matrices with each row identically distributed:

```
R>  Z1 <- gmatrix(grnorm(100, mean=1:10), 10, 10, dup=FALSE)
R>  Z2 <- gmatrix(grgamma(100, shape=1:10), 10, 10, dup=FALSE)
R>  Z3 <- gmatrix(grbeta(100, shape1=1:10, shape2=1:10),10,10, dup=FALSE)
```

We have also implemented functionality for calculating the density functions of all of the above distributions. This again follows the convention of prepending a "g" in front of the R command. For example, `gdnorm()` is the GPU version of `dnorm()`. We have not implemented quantile functions or cumulative distribution functions for any distribution except the normal distribution: `gqnorm` and `gpnorm`.

4. Multiple GPUs

If multiple NVIDIA®GPUs are available on a system, the available devices may be listed using the function `listDevices()`. The function `getDevice` and `setDevice` get and set the current device. All GPU operations are performed on the current GPU device only. Any attempt to operate on an object which is located on a different GPU will result in an error. For example, the following code should result in an error:

```
R>  setDevice(0)
R>  Z <- gmatrix(grgamma(100, shape=1:10), 10, 10, dup=FALSE)
R>  setDevice(1)
R>  Z <- sin(Z)
```

To find out which device a given object (e.g., `Z`) is stored on, a command such as `device(Z)` may be used. To move data from one GPU to another, a command such as `device(Z)<-1` may be used. To copy the data to another device while leaving the original copy alone, the command `Z2<-gdub(Z,device=2)` may be used. For example, to avoid the problem in the code above, we may use:

```
R>  setDevice(0)
R>  Z <- gmatrix(grgamma(100, shape=1:10), 10, 10, dup=FALSE)
R>  setDevice(1)
R>  device(Z) <- 1
R>  Z <- sin(Z)
```

It is possible to use **gmatrix** in conjunction with **snow** to submit work to the GPU in parallel using the command `clusterApply`. However, the user is warned that it is *extremely* important not to attempt to pass any objects of class `gmatrix` or `gvector` between the **snow** cluster sessions and the master R session. Any such attempt is likely to cause a fatal error because the cluster sessions do not have access to the same memory as the master R session. All objects *must* be returned to the CPU memory before attempting to export them to the cluster or return them to the main session. We have found that by using **snow**, it is possible to increase speed even when using a single GPU. By submitting multiple kernels to the GPU simultaneously, more of the GPUs power can be utilized. Here is a simple, safe example to explore the Central Limit Theorem with a single GPU:

```
R>  n <- 1e7
R>  library("snow")
R>  cl <- makeSOCKcluster(rep("localhost",3))
R>  clusterExport(cl, "n")
R>  clusterEvalQ(cl, library("gmatrix"))
R>  pvals <- clusterApply(cl, 1:1000, fun = function(i) {
R>                          ggc(TRUE)
R>                          X <- exp(grnorm(n, sd=5))
R>                          Y <-exp(grnorm(n, sd=5))
R>                          Xbar <- mean(X); Ybar=mean(Y)
R>                          se <- sqrt( (sum( (X-Xbar)^2 ) +
                              sum( (Y-Ybar)^2 ))/((n-2)*n))
R>                          T <- (h(Xbar)-h(Ybar))/h(se)
R>                          return(2*pnorm(-abs(T)))
R>  })
R>  stopCluster(cl)
R>  type1error <- mean(unlist(pvals)<.05)
```

We have found this code to be about 80% faster than the simpler approach which uses `lapply` instead of `clusterApply`.

5. Benchmarking

To investigate the value of the **gmatrix** package, we have performed a number of benchmark tests. All test were performed on the CPU side with an Intel®Xeon®E5630 processor with

2.53GHz, and on the GPU side with an NVIDIA®C2050 card. Timing was performed using the package **microbenchmark**, although in many cases the time was long enough that the **microbenchmark** package would have been unnecessary. Of course, the benchmark speeds will vary widely depending on the GPU and CPU hardware components being compared. These benchmarks also only compare a particular implementation of the GPU operations with standard R implementations of various operations, and we would warn the user not to overgeneralize these benchmarks as generally representative of GPU vs. CPU comparisons. There is probably significant room for optimization of both the CPU and GPU operations. None of these benchmarks make use of the `"single"` type which could make the GPU significantly faster.

First, we investigated the performance of the random number generating functionality in **gmatrix**. The results are shown in Figure 1. The normal simulation compared `grnorm(n)` vs. `grnorm(n)` where `n` is the number on the x-axis of the figure. The gamma simulation compared `rgamma(n, shape=100)` with `grgamma(n, shape=100)`. The beta simulation compared `rbeta(n,shape1=10, shape2=10)` with `grbeta(n,shape1=10,shape2=10)`. The binomial simulation compared `rbinom(n,size=100, prob=0.5)` with `grbinom(n, size=100, prob=0.5)`. The Poisson simulation compared CPU function `rpois(n,lambda=100)` with the GPU function `grpois(n, lambda=100)`. We can see from the figure that in order for the GPU to be efficient it must be simulating approximately 3×10^4 random numbers at a time (the exact number depends on the distribution). However, when a large quantity of random numbers must be simulated, the GPU gave more than an 80x speedup for simulating from a normal distribution and beta distribution. Other distributions such as Poisson and binomial appeared to yield relatively small speedups. Of course it should be noted that particularly for the Poisson and binomial distribution, the amount of speedup may be heavily dependent on the parameters for the distribution being simulated. For example, while not shown in Figure 1, we found that when comparing `rbinom(1E7, size=2, prob=.5)` to `grbinom(1E7,size=2, prob=.5)` the speedup was approximately 100x.

Next, we looked at the performance of some simple binary operations. As shown in Figure 2, it takes a very large vector/matrix of length at least 5×10^5 to have any gains on the GPU, for most of the operations. Exponentiation (i.e., x^y) appears to enjoy greater benefits on the GPU than the other operations. However, it would not necessarily be beneficial to move an object back to the CPU for these operations because, as we will see, the cost of moving the matrix/vector back to the CPU may be quite large.

Next, we considered the performance of some logical operators and functions. Figure 3 shows the results of this investigation. The `ifelse()` function appears to be quite efficient on the GPU, and starts beating the CPU function with a vector size of only about 5000. The GPU based `&` and `|` operators start beating the CPU for vectors of length approximately 7×10^4 and speed up the operation by about 60x for vectors of length 10^7. The GPU `which` function starts outperforming CPU for vector sizes of approximately 8×10^4 and achieves a 20x improvement in speed for vectors of length 10^7.

We also looked at the performance of several mathematical special functions. As shown in Figure 4, the GPU `exp()`, `gamma()`, `sin()` function were faster than the CPU for vectors of about 8000, and all of these function had very large speedups for vectors of 10^7 ($>$ 200x speedup). The `asin()` and `log()` were a little less dramatic in their performance but still quite impressive.

Matrix multiplication is one operation in which the GPU excels for even relatively small matrices. As shown in Figure 5, the GPU easily beats the CPU (by ~7x) when multiplying two 100×100 matrices and the speedup is ~120x when multiplying two 1000×1000 matrices. Of course, the CPU based matrix multiplication could be significantly improve by setting up R to use a faster BLAS library (Eddelbuettel 2010). Similarly, the Kronecker product on the GPU easily beats the CPU when operating on two matrices of size 100×5. The transpose operation on the GPU requires larger sized matrices to be efficient.

Next, we investigated the indexing operations on the GPU. Figure 6 displays some results for indexing operations. It should be noted that the speed of indexing operations on the GPU relative to the CPU may be heavily dependent on the size (i.e., dimensions) of the objects being indexed. We were concerned that, due to the need of the GPU to coalesce its memory accesses, the indexing on the GPU might be heavily effected by the ordering of the index. This turns out not to be a big problem in the scenario shown in Figure 6 since the randomly ordered index had a similar performance to the sequentially ordered index.

Finally, we discuss the transfer of memory to and from the GPU. As shown in Figure 7, every memory transfer has an initial overhead to start the transfer regardless of the memory size being transferred. The optimal approach for a given problem may not always be to run an operation on the hardware platform which can perform it fastest. At times, it may be beneficial to perform on the GPU an operation which is faster on th CPU, to avoid the cost of bringing the data back and forth between the GPU and CPU.

6. Conclusions

This paper has shown through benchmark tests that large gains in speed can be achieved by using the **gmatrix** package in conjunction with NVIDIA®GPUs. As these devices are becoming increasingly available on academic clusters, there can be little doubt that R packages such as the one described in this paper will be useful to applied statisticians. Of course, we do not want overstate the case for using GPU power. For small problems and some exceedingly complex algorithms, GPUs are likely to slow the results down. Despite these limitations, **gmatrix** has proven useful in our own implementation of Bayesian MCMC schemes with large data sets. Many complex algorithms can be broken down into a sequence of relatively simple steps, often involving linear algebra operations, sorting and random number generation. Frequently the bottleneck steps in these algorithms can be sped up by orders of magnitude fairly simply using a GPU. We are convinced that there is a tremendous opportunity to use GPU power to take many previously computationally impossible tasks and put them well within range of our current computing power.

References

Bates D, Maechler M (2013). ***Matrix**: Sparse and Dense Matrix Classes and Methods*. R package version 1.0-11, URL http://Matrix.R-forge.R-project.org/.

da Silva AF (2011). "**cudaBayesreg**: Parallel Implementation of a Bayesian Multilevel Model for fMRI Data Analysis." *Journal of Statistical Software*, **44**(4), 1–24. ISSN 1548-7660. URL http://www.jstatsoft.org/v44/i04.

Eddelbuettel D (2010). ***gcbd**: GPU/CPU Benchmarking in Debian-based systems*. URL http://cran.r-project.org/web/packages/gcbd/.

Hoberock J, Bell N (2010). "**Thrust**: A Parallel Template Library." Version 1.7.0, URL http://thrust.github.io/.

J Buckner MS, Wilson J (2013). ***gputools**: A few GPU enabled functions*. R package version 0.28, URL http://www.hiplar.org/hiplar-b.html.

Kempenaar M, Dijkstra M (2013). ***R/GPU**: Using the Graphics Processing Unit to speedup bioinformatics analysis with R*. R package version 0.8-1, URL https://gforge.nbic.nl/projects/rgpu/.

Marsaglia G, Tsang WW (2000). "A simple method for generating gamma variables." *ACM Trans. Math. Softw.*, **26**(3), 363–372. ISSN 0098-3500. doi:10.1145/358407.358414. URL http://doi.acm.org/10.1145/358407.358414.

Morris N (2013). ***rcula**: An R plugin for matrix factorization and inversion*. R package version 0.1, URL https://github.com/njm18/rcula/.

Nash P, Szeremi V (2012). ***HiPLARM**: High Performance Linear Algebra in R*. R package version 0.1, URL http://cran.r-project.org/web/packages/gcbd/.

Smith BJ (2013). ***magma**: Matrix Algebra on GPU and Multicore Architectures*. R package version 1.3.0-2, URL http://icl.cs.utk.edu/magma/.

Szeremi V (2012). ***HiPLARb**: High Performance Linear Algebra in R*. R package version 0.1.3, URL http://www.hiplar.org/hiplar-b.html.

Affiliation:

Nathan Morris
Department of Epidemiology and Biostatistics
Case Western Reserve University
2103 Cornell Rd, Wolsten Bldg
Cleveland OH 44106, USA
E-mail: nathan.morris@case.edu
URL: http://epbiwww.case.edu/index.php/people/faculty/97-morris

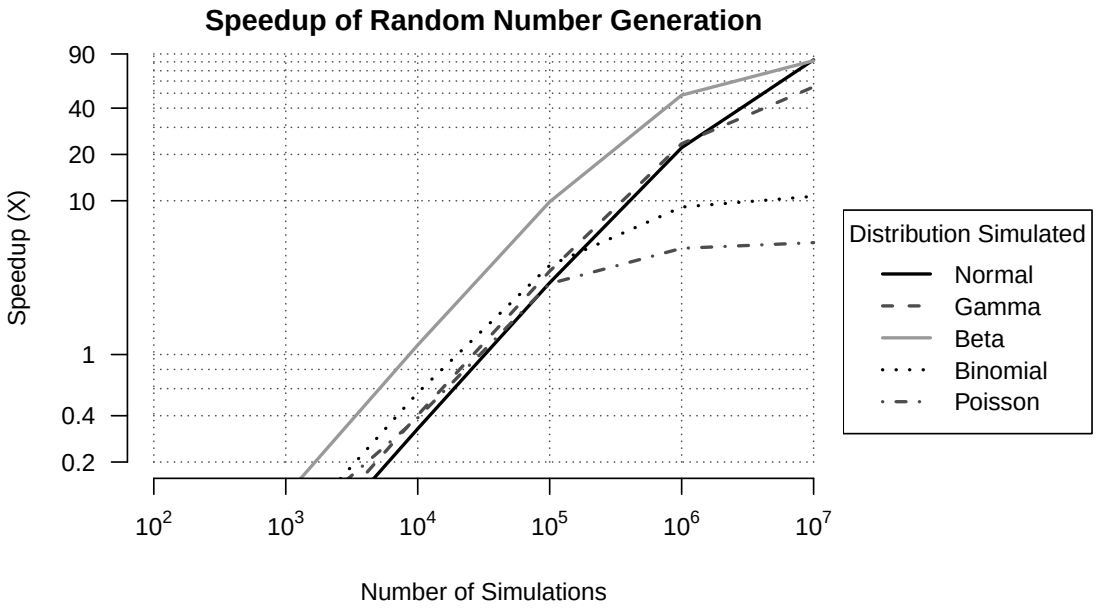

Figure 1: The speedup times observed for simulating from various distributions. Note the log scale on both axis.

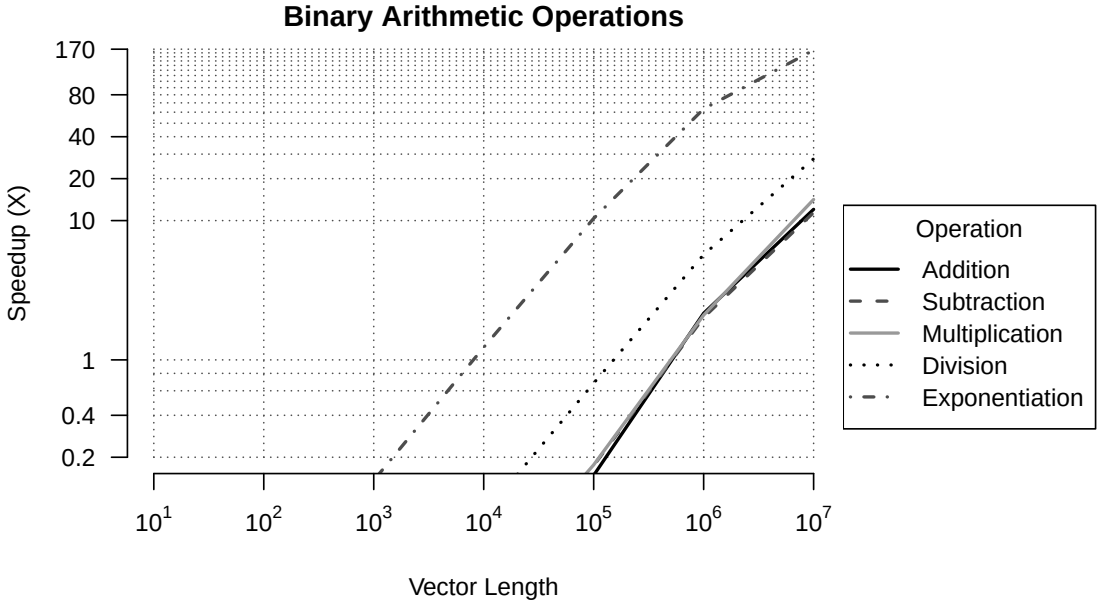

Figure 2: The speedup times observed for different binary operations. Note the log scale on both axis.

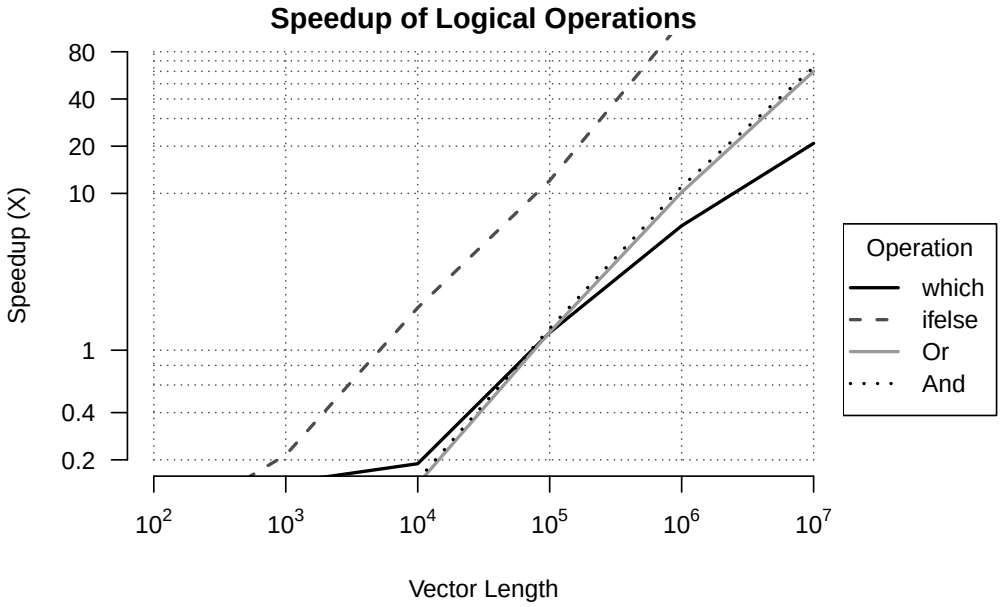

Figure 3: The speedup times observed for some logical operations. Note the log scale on both axis.

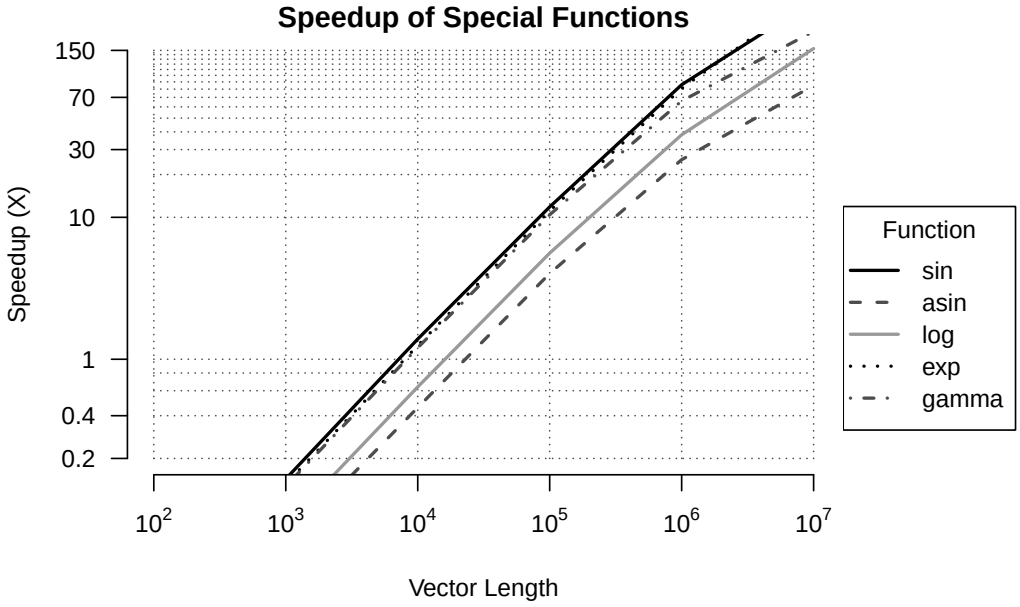

Figure 4: The speedup times observed for some mathematical functions. Note the log scale on both axis.

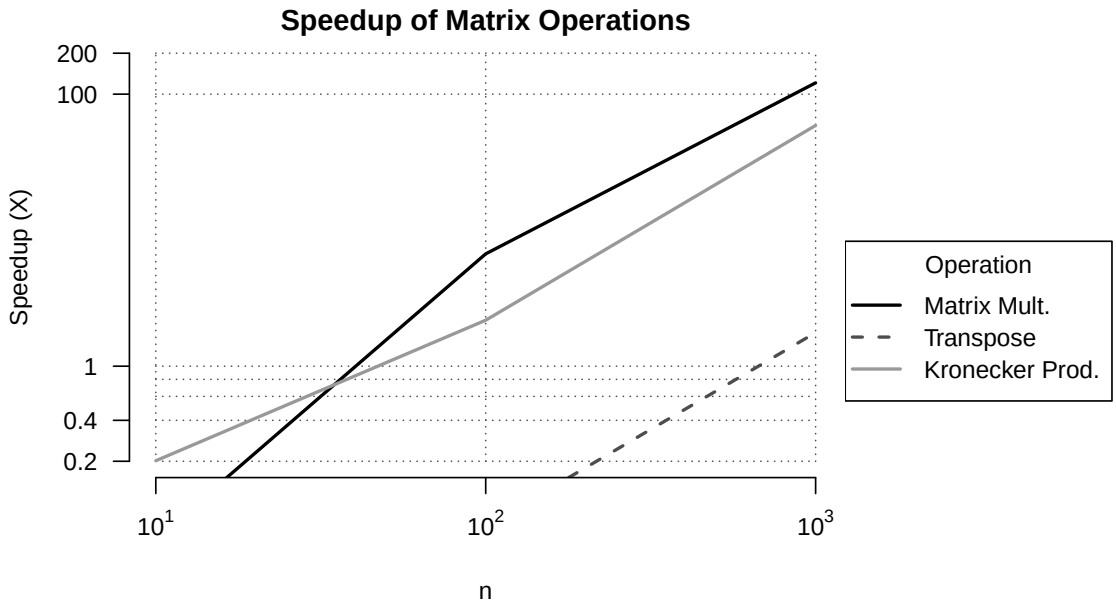

Figure 5: The speedup times observed for some matrix operations. The matrix multiplication is of two $n \times n$ matrices. The transpose is of an $n \times n$ matrix. The Kronecker product is for two $n \times 5$ matrices. Note the log scale on both axis.

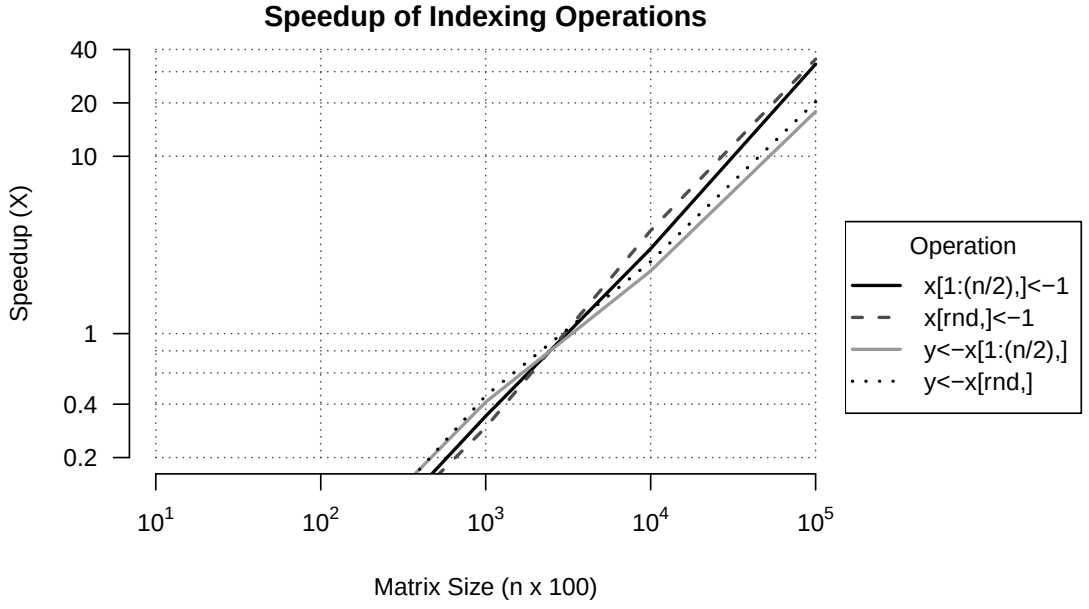

Figure 6: Speedup associated with indexing operations performed on an $n \times 100$ matrix. For the seqential index (i.e., `1:(n/2)`) the index was calculated as `1 %to% (n/2)` for the GPU indexing operation. The variable '`rnd`' referred to in the legend is defined as `sample(1:(n/2))` and neither the creation time for '`rnd`' nor the transfer time were included in the benchmark calculations.

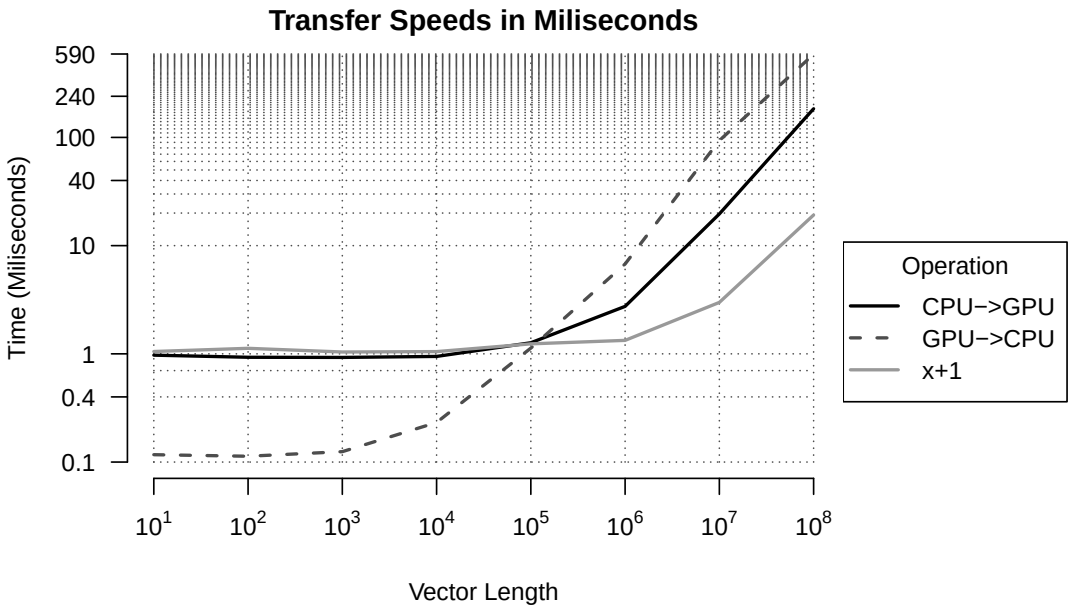

Figure 7: The amount of time it takes to transfer an object of type `"double"` to and from the GPU. For comparison sake we show the amount of time it would take to add 1 to all the elements of the vector on the GPU.