Get started with GPUmatrix package

Seamlessly harness the power of GPU compu-ting in R

Cesar Lobato-Fernandez, Juan A. Ferrer-Bonsoms & Ángel Rubio

February 2023

## Abstract

**Motivation:** GPU computational power is a great resource for computational biology specifically in statistics and linear algebra. Unfortunately, very few packages connect R with the GPU and none of them are transparent enough to perform the computations on the GPU without substantial changes of the code. Another problem of these packages is lacking proper maintenance: several of the previous attempts were removed from CRAN. It would be desirable to have a R package, properly maintained, that exploits the use of the GPU with minimal changes in the existing code.

**Results:** We have developed the GPUMatrix package (available at CRAN). GPUMatrix mimics the behavior of the Matrix package. Therefore, is easy to learn and very few changes in the code are required to work on the GPU. GPUMatrix relies on either the tensorflow or the torch R packages to perform the GPU operations.

Before starting, please be advised that this R package is designed to have the lowest learning curve for the R user to perform algebraic operations using the GPU. Therefore, this tutorial will mostly cover procedures that will go beyond the operations that the user can already perform with R’s CPU matrices.

# 0 Installation

## 0.1 Dependences

GPUmatrix is an R package that utilizes tensors through the **torch** or **tensorflow** packages (see Advanced Users section for more information). One or the other must be installed for the use of GPUmatrix. Both packages are hosted in CRAN and have specific installation instructions. In both cases, it is necessary to have an NVIDIA® GPU card with the latest drivers installed in order to use the packages, as well as a version of Python 3. The NVIDIA card must be compatible; please see the list of capable cards [here](https://developer.nvidia.com/cuda-gpus#compute). If there is no compatible graphics card or not graphic card at all, you can still install tensorFlow and torch, but only with the CPU version, which means that GPUmatrix will only be able to run in CPU mode.

### For torch: ([Link installation here](https://cran.r-project.org/web/packages/torch/vignettes/installation.html))

install.packages("torch")

* **MUST INSTALL:**
  + CUDA Toolkit 11.3. Link [here](https://developer.nvidia.com/cuda-11.3.0-download-archive).
  + cuDNN 8.4 . Link [here](https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-840/install-guide/index.html).

### For Tensorflow: ([Link installation here](https://tensorflow.rstudio.com/install/))

The installation of TensorFlow allows the selection to install the GPU, CPU, or both versions. This will depend on the version of TensorFlow that we install with the install\_tensorflow() function. The mode in which the tensors are created using GPUmatrix, if we choose to use TensorFlow, will depend on the installation mode. The options to switch from CPU to GPU are not enabled when using GPUmatrix with TensorFlow for this precise reason. To install the GPU version, it is not necessary to specify the version since if it detects that the CUDA dependencies are met, it will automatically install using the GPU mode. If you want to install the CPU version, you need to specify it as follows:

install\_tensorflow(version="nightly-cpu")

install.packages("tensorflow")  
library(tensorflow)  
install\_tensorflow(version = "nightly-gpu")

* **MUST INSTALL:**
  + CUDA Toolkit 11.2. Link [here](https://developer.nvidia.com/cuda-11.3.0-download-archive).
  + cuDNN 8.1 . Link [here](https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-840/install-guide/index.html).

## 0.2 GPUmatrix installation

Once the dependencies for Torch or TensorFlow are installed, the GPUmatrix package, being a package hosted on CRAN, can be easily installed using:

install.packages("GPUmarix")

# 1 Initialization GPUmatrix

The GPUmatrix package is based on S4 objects in R and we have created a constructor function that acts similarly to the default matrix() constructor in R for CPU matrices. The constructor function is gpu.matrix() and accepts the same parameters as matrix():

>library(GPUmatrix)  
#R matrix initialization  
>m <- matrix(c(1:20)+40,10,2)  
#Show CPU matrix  
>m  
  
 [,1] [,2]  
 [1,] 41 51  
 [2,] 42 52  
 [3,] 43 53  
 [4,] 44 54  
 [5,] 45 55  
 [6,] 46 56  
 [7,] 47 57  
 [8,] 48 58  
 [9,] 49 59  
[10,] 50 60  
  
#GPU matrix initialization  
>Gm <- gpu.matrix(c(1:20)+40,10,2)  
#Show GPU matrix  
>Gm  
  
GPUmatrix  
torch\_tensor  
 41 51  
 42 52  
 43 53  
 44 54  
 45 55  
 46 56  
 47 57  
 48 58  
 49 59  
 50 60  
[ CUDADoubleType{10,2} ]

In the previous example, a normal R CPU matrix called m and its GPU counterpart Gm are created. Just like regular matrices, the created GPU matrices allow for indexing of its elements and assignment of values. The concatenation operators rbind() and cbind() work independently of the type of matrices that are to be concatenated, resulting in a ***gpu.matrix***:

>Gm[c(2,3),1]  
  
GPUmatrix  
torch\_tensor  
 42  
 43  
[ CUDADoubleType{2,1} ]  
  
>Gm[,2]  
  
GPUmatrix  
torch\_tensor  
 51  
 52  
 53  
 54  
 55  
 56  
 57  
 58  
 59  
 60  
[ CUDADoubleType{10,1} ]  
   
>Gm2 <- cbind(Gm[c(1,2),], Gm[c(6,7),])  
>Gm2  
   
 GPUmatrix  
torch\_tensor  
 41 51 46 56  
 42 52 47 57  
[ CUDADoubleType{2,4} ]  
  
>Gm2[1,3] <- 0  
>Gm2  
  
GPUmatrix  
torch\_tensor  
 41 51 0 56  
 42 52 47 57  
[ CUDADoubleType{2,4} ]

# 2 Cast GPU matrices and data types

The default matrices in R have limitations. The numeric data types it allows are int and float64, with float64 being the type used generally in R by default. It also does not natively allow for the creation and handling of sparse matrices. To make up for this lack of functionality, other R packages hosted in CRAN have been created that allow for programming these types of functionality in R. The problem with these packages is that in most cases they are not compatible with each other, meaning we can have a sparse matrix with float64 and a non-sparse matrix with float32, but not a sparse matrix with float32.

## 2.1 Cast from other packages

GPUmatrix allows for compatibility with sparse matrices and different data types such as float32. For this reason, casting operations between different matrix types from multiple packages to GPU matrix type have been implemented:

| Matrix class | Package | Data type default | SPARSE | Back cast |
| --- | --- | --- | --- | --- |
| matrix | base | float64 | FALSE | Yes |
| data.frame | base | float64 | FALSE | Yes |
| integer | base | float64 | FALSE | Yes |
| numeric | base | float64 | FALSE | Yes |
| dgeMatrix | Matrix | float64 | FALSE | No |
| ddiMatrix | Matrix | float64 | TRUE | No |
| dpoMatrix | Matrix | float64 | FALSE | No |
| dgCMatrix | Matrix | float64 | TRUE | No |
| float32 | float | float32 | FALSE | No |
| torch\_tensor | torch | float64 | Depends of tensor type | Yes |
| tensorflow.tensor | tensorflow | float64 | Depends of tensor type | Yes |

There are two functions for casting to create a ***gpu.matrix***: **as.gpu.matrix()** and the **gpu.matrix()** constructor itself. Both have the same input parameters for casting: the object to be cast and extra parameters for creating a GPU matrix.

#Create 'Gm' from 'm' matrix  
>m <- matrix(c(1:20)+40,10,2)  
>Gm <- gpu.matrix(m)  
>Gm  
  
GPUmatrix  
torch\_tensor  
 41 51  
 42 52  
 43 53  
 44 54  
 45 55  
 46 56  
 47 57  
 48 58  
 49 59  
 50 60  
[ CUDADoubleType{10,2} ]   
  
#Create 'Gm' from 'M' with Matrix package  
>library(Matrix)  
>M <- Matrix(c(1:20)+40,10,2)  
>Gm <- gpu.matrix(M)  
>Gm  
   
GPUmatrix  
torch\_tensor  
 41 51  
 42 52  
 43 53  
 44 54  
 45 55  
 46 56  
 47 57  
 48 58  
 49 59  
 50 60  
[ CUDADoubleType{10,2} ]  
  
#Create 'Gm' from 'mfloat32' with float package  
>library(float)  
>mfloat32 <- fl(m)  
>Gm <- gpu.matrix(mfloat32)  
>Gm  
   
GPUmatrix  
torch\_tensor  
 41 51  
 42 52  
 43 53  
 44 54  
 45 55  
 46 56  
 47 57  
 48 58  
 49 59  
 50 60  
[ CUDAFloatType{10,2} ] #Float32 data type  
   
#Create 'Gms' type sparse from 'Ms' type sparse dgCMatrix with Matrix package  
>Ms <- Matrix(sample(0:1, 20, replace = TRUE), nrow=10, ncol=2, sparse=TRUE)  
>Ms  
   
10 x 2 sparse Matrix of class "dgCMatrix"  
   
 [1,] 1 1  
 [2,] . 1  
 [3,] . 1  
 [4,] 1 1  
 [5,] . .  
 [6,] . .  
 [7,] . .  
 [8,] 1 1  
 [9,] . .  
[10,] . .  
   
>Gms <- gpu.matrix(Ms)  
>Gms  
   
GPUmatrix  
torch\_tensor  
[ SparseCUDADoubleType{}  
indices:  
 0 0 1 2 3 3 7 7  
 0 1 1 1 0 1 0 1  
[ CUDALongType{2,8} ]  
values:  
 1  
 1  
 1  
 1  
 1  
 1  
 1  
 1  
[ CUDADoubleType{8} ]  
size:  
[10, 2]  
]

## 2.2 Data type and sparsity

The data types allowed by GPUmatrix are: **float64**, **float32**, **int**, **bool** or **logical**, **complex64** and **complex32**. We can create a GPU matrix with a specific data type using the **dtype** parameter of the **gpu.matrix()** constructor function or change the data type of a previously created GPU matrix using the **dtype()** function. The same applies to GPU sparse matrices, we can create them from the constructor using the **sparse** parameter, which will obtain a Boolean value of TRUE/FALSE depending on whether we want the resulting matrix to be sparse or not. We can also modify the sparsity of an existing GPU matrix with the functions **to\_dense()**, if we want it to go from sparse to dense, and **to\_sparse()**, if we want it to go from dense to sparse.

#Creating a float32 matrix  
>Gm32 <- gpu.matrix(c(1:20)+40,10,2, dtype = "float32")  
>Gm32  
  
GPUmatrix  
torch\_tensor  
 41 51  
 42 52  
 43 53  
 44 54  
 45 55  
 46 56  
 47 57  
 48 58  
 49 59  
 50 60  
[ CUDAFloatType{10,2} ] #Float32 data type  
  
#Creating a non sparse martix with data type float32 from a sparse matrix type float64  
>Ms <- Matrix(sample(0:1, 20, replace = TRUE), nrow=10, ncol=2, sparse=TRUE)  
>Gm32 <- gpu.matrix(Ms, dtype = "float32", sparse = F)  
>Gm32  
  
GPUmatrix  
torch\_tensor  
 1 1  
 0 1  
 0 1  
 1 1  
 0 0  
 0 0  
 0 0  
 1 1  
 0 0  
 0 0  
[ CUDAFloatType{10,2} ]  
   
#Convert Gm32 in sparse matrix Gms32  
>Gms32 <- to\_sparse(Gm32)  
>Gms32  
  
GPUmatrix  
torch\_tensor  
[ SparseCUDAFloatType{}  
indices:  
 0 0 1 2 3 3 7 7  
 0 1 1 1 0 1 0 1  
[ CUDALongType{2,8} ]  
values:  
 1  
 1  
 1  
 1  
 1  
 1  
 1  
 1  
[ CUDAFloatType{8} ]  
size:  
[10, 2]  
]  
  
##Convert data type Gms32 in float64  
>Gms64 <- Gms32  
>dtype(Gms64) <- "float64"  
>Gms64  
  
GPUmatrix  
torch\_tensor  
[ SparseCUDADoubleType{}  
indices:  
 0 0 1 2 3 3 7 7  
 0 1 1 1 0 1 0 1  
[ CUDALongType{2,8} ]  
values:  
 1  
 1  
 1  
 1  
 1  
 1  
 1  
 1  
[ CUDADoubleType{8} ]  
size:  
[10, 2]  
]

# 3 GPUmatrix functions

## 3.1 Arithmetic and comparison operators

GPUmatrix supports all basic arithmetic operators in R: +, -, \*, ^, /, %\*% and %%. Its usage is the same as for basic R matrices, and it allows compatibility with other matrix objects from the previously mentioned packages, always returning the result in GPUmatrix format.

>(Gm + Gm) == (m + m)  
  
 [,1] [,2]  
 [1,] TRUE TRUE  
 [2,] TRUE TRUE  
 [3,] TRUE TRUE  
 [4,] TRUE TRUE  
 [5,] TRUE TRUE  
 [6,] TRUE TRUE  
 [7,] TRUE TRUE  
 [8,] TRUE TRUE  
 [9,] TRUE TRUE  
[10,] TRUE TRUE  
  
>(Gm + M) == (mfloat32 + Gm)  
  
 [,1] [,2]  
 [1,] TRUE TRUE  
 [2,] TRUE TRUE  
 [3,] TRUE TRUE  
 [4,] TRUE TRUE  
 [5,] TRUE TRUE  
 [6,] TRUE TRUE  
 [7,] TRUE TRUE  
 [8,] TRUE TRUE  
 [9,] TRUE TRUE  
[10,] TRUE TRUE  
  
>(M + M) == (mfloat32 + Gm)  
  
 [,1] [,2]  
 [1,] TRUE TRUE  
 [2,] TRUE TRUE  
 [3,] TRUE TRUE  
 [4,] TRUE TRUE  
 [5,] TRUE TRUE  
 [6,] TRUE TRUE  
 [7,] TRUE TRUE  
 [8,] TRUE TRUE  
 [9,] TRUE TRUE  
[10,] TRUE TRUE  
  
>(Ms + Ms) > (Gms + Gms)\*2  
  
 [,1] [,2]  
 [1,] TRUE TRUE  
 [2,] FALSE TRUE  
 [3,] FALSE TRUE  
 [4,] TRUE TRUE  
 [5,] FALSE FALSE  
 [6,] FALSE FALSE  
 [7,] FALSE FALSE  
 [8,] TRUE TRUE  
 [9,] FALSE FALSE  
[10,] FALSE FALSE

As seen in the previous example, the comparison operators (==, !=, >, <, >=, <=) also work following the same dynamic as the arithmetic operators.

## 3.2 Math operators

Similarly to arithmetic operators, mathematical operators follow the same operation they would perform on regular matrices of R. Gm is a *gpu.matrix* variable:

| Mathematical operators | Usage |
| --- | --- |
| log | log(Gm) |
| log2 | log2(Gm) |
| log10 | log10(Gm) |
| cos | cos(Gm) |
| cosh | cosh(Gm) |
| acos | acos(Gm) |
| acosh | acosh(Gm) |
| sin | sin(Gm) |
| sinh | sinh(Gm) |
| asin | asin(Gm) |
| asinh | asinh(Gm) |
| tan | tan(Gm) |
| atan | atan(Gm) |
| tanh | tanh(Gm) |
| atanh | atanh(Gm) |
| sqrt | sqrt(Gm) |
| abs | abs(Gm) |
| sign | sign(Gm) |
| ceiling | ceiling(Gm) |
| floor | floor(Gm) |
| cumsum | cumsum(Gm) |
| cumprod | cumprod(Gm) |
| exp | exp(Gm) |
| expm1 | expm1(Gm) |

## 3.2 Other functions

In the manual, we can find a multitude of functions that can be applied to *gpu.matrix* type matrices. Most of the functions are functions from the base R package that can be used on *gpu.matrix* matrices in the same way they would be applied to regular matrices of R. There are other functions from other packages like **Matrix** or **matrixStats** that have been implemented due to their widespread use within the user community, such as rankMatrix or colMaxs. The output of these functions, which originally produced R default matrix type objects, will now return *gpu.matrix* type matrices if the input type of the function is *gpu.matrix*.

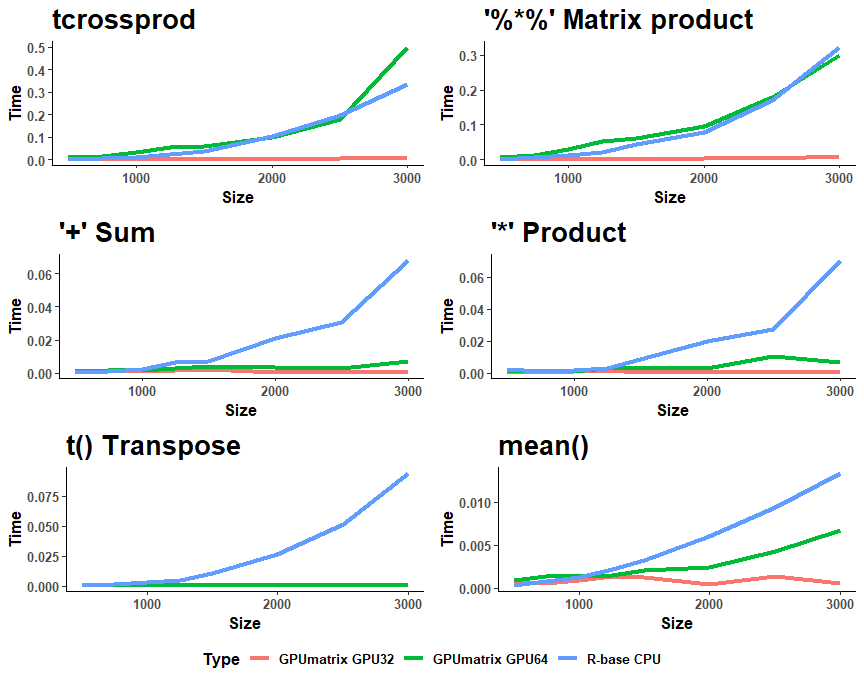
>m <- matrix(c(1:20)+40,10,2)  
>Gm <- gpu.matrix(c(1:20)+40,10,2)  
  
>head(tcrossprod(m),2)  
  
 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
[1,] 4282 4374 4466 4558 4650 4742 4834 4926 5018 5110  
[2,] 4374 4468 4562 4656 4750 4844 4938 5032 5126 5220  
  
>head(tcrossprod(Gm),2)  
  
GPUmatrix  
torch\_tensor  
 4282 4374 4466 4558 4650 4742 4834 4926 5018 5110  
 4374 4468 4562 4656 4750 4844 4938 5032 5126 5220  
[ CUDADoubleType{2,10} ]  
  
>Gm <- tail(Gm,3)  
>rownames(Gm) <- c("a","b","c")  
>tail(Gm,2)  
   
GPUmatrix  
torch\_tensor  
 49 59  
 50 60  
[ CUDADoubleType{2,2} ]  
rownames: b c  
  
>colMaxs(Gm)  
  
[1] 50 60

There is a wide variety of functions implemented in GPUmatrix, and they are adapted to be used just like regular R matrices.

| Functions | Usage | Package |
| --- | --- | --- |
| determinant | determinant(Gm, logarithm=T) | base |
| det | fft(Gm) | base |
| sort | sort(Gm,decreasing=F) | base |
| round | round(Gm, digits=0) | base |
| show | show(Gm) | base |
| length | length(Gm) | base |
| dim | dim(Gm) | base |
| dim<- | dim(Gm) <- c(...,...) | base |
| rownames | rownames(Gm) | base |
| rownames<- | rownames(Gm) <- c(...) | base |
| row.names | row.names(Gm) | base |
| row.names<- | row.names(Gm) <- c(...) | base |
| colnames | colnames(Gm) | base |
| colnames<- | colnames(Gm) <- c(...) | base |
| rowSums | rowSums(Gm) | Matrix |
| colSums | colSums(Gm) | Matrix |
| cbind | cbind(Gm,...) | base |
| rbind | rbind(Gm,...) | base |
| head | head(Gm,...) | base |
| tail | tail(Gm,...) | base |
| nrow | nrow(Gm) | base |
| ncol | ncol(Gm) | base |
| t | t(Gm) | base |
| crossprod | crossprod(Gm,...) | base |
| tcrossprod | tcrossprod(Gm,…) | base |
| outer | outer(Gm,…) | base |
| %o% | Gm %o% … || … %o% Gm | base |
| %x% | Gm %x% … || … %x% Gm | base |
| %^% | Gm %^% … || … %^% Gm | base |
| diag | diag(Gm) | base |
| diag<- | diag(Gm) <- c(…) | base |
| solve | solve(Gm, …) | base |
| qr | qr(Gm) | base |
| rankMatrix | rankMatrix(Gm) | Matrix |
| eigen | eigen(Gm) | base |
| svd | svd(Gm) | base |
| ginv | ginv(Gm, tol = sqrt(.Machine$double.eps)) | base |
| chol | chol(Gm) | base |
| chol\_solve | chol\_solve(Gm, …) | GPUmatrix |
| mean | mean(Gm) | base |
| density | density(Gm) | base |
| hist | hist(Gm) | base |
| colMeans | colMeans(Gm) | Matrix |
| rowMeans | rowMeans(Gm) | Matrix |
| sum | sum(Gm) | base |
| min | min(Gm) | base |
| max | max(Gm) | base |
| which.max | which.max(Gm) | base |
| which.min | which.min(Gm) | base |
| aperm | aperm(Gm) | base |
| apply | apply(Gm, MARGIN, FUN, …, simplify=TRUE) | base |
| cov | cov(Gm) | stats |
| cov2cor | cov2cor(Gm) | stats |
| cor | cor(Gm, …) | stats |
| rowVars | rowVars(Gm) | matrixStats |
| colVars | colVars(Gm) | matrixStats |
| colMaxs | colMaxs(Gm) | matrixStats |
| rowMaxs | rowMaxs(Gm) | matrixStats |
| rowRanks | rowRanks(Gm) | matrixStats |
| colRanks | colRanks(Gm) | matrixStats |
| colMins | colMins(Gm) | matrixStats |
| rowMins | rowMins | matrixStats |
| dtype | dtype(Gm) | GPUmatrix |
| dtype<- | dtype(Gm) | GPUmatrix |
| to\_dense | to\_dense(Gm) | GPUmatrix |
| to\_sparse | to\_sparse(Gm) | GPUmatrix |

## 3.3 Function time comparison

The computation time for the different functions and operations differs depending on the operation to be performed (Fig 1). Although the default data type is float64, operations with float32 have no comparison in terms of computation time. For this reason, we recommend their use whenever the data types and the objective allow it. This comparison is made using the Intel MKL BLAS.



Computation time for GPU and R-base CPU for different operations. Time is in seconds and Size=n where matrix is (n x n) dimension.

# 4. Toy example: Non negative factorization of a matrix

As a toy example We will show a simple example on performing the non negative matrix factorization of a matrix (NMF) using the Lee and Seung multiplicative update rule.

The rules are

and

to update the and respectively.

It is straightforward to build two functions for these rules. The corresponding R code is:

>updateH <- function(V,W,H) {  
> H <- H \* (t(W) %\*% V)/((t(W) %\*% W) %\*% H)}  
>updateW <- function(V,W,H) {  
> W <- W \* (V %\*% t(H))/(W %\*% (H %\*% t(H)) )}

We include a simple script that builds a matrix and run this update rules 100 times.

>A <- matrix(runif(200\*10),200,10)  
>B <- matrix(runif(10\*100),10,100)  
>V <- A %\*% B  
  
>W <- W1 <- matrix(runif(200\*10),200,10)  
>H <- H1 <- matrix(runif(10\*100),10,100)  
  
>for (iter in 1:100) {  
> W <- updateW(V,W,H)  
> H <- updateH(V,W,H)  
>}  
  
>print(W[1,1])  
>print(H[1,1])  
  
[1] 0.5452606  
[1] 0.5010532

We include now a similar script where the operations are done on the GPU:

>library(GPUmatrix)  
>Vg <- gpu.matrix(V)  
  
>Wg <- gpu.matrix(W1)  
>Hg <- gpu.matrix(H1)  
  
>for (iter in 1:100) {  
> Wg <- updateW(Vg,Wg,Hg)  
> Hg <- updateH(Vg,Wg,Hg)  
>}  
  
>print(Wg[1,1])  
>print(Hg[1,1])  
  
GPUmatrix  
torch\_tensor  
 0.5453  
[ CUDADoubleType{1,1} ]  
GPUmatrix  
torch\_tensor  
 0.5011  
[ CUDADoubleType{1,1} ]

Results are identical since the initial values also coincide.

# 5. Advanced options

## 5.1 Using GPUMatrix on CPU

In the GPUmatrix constructor, we can specify the location of the matrix, i.e., we can decide to host it on the GPU or in RAM memory to use it with the CPU. As a package, as its name suggests, oriented towards algebraic operations in R using the GPU, it will by default be hosted on the GPU, but it allows the same functionalities using the CPU. To do this, we use the device attribute of the constructor and assign it the value ***“cpu”***.

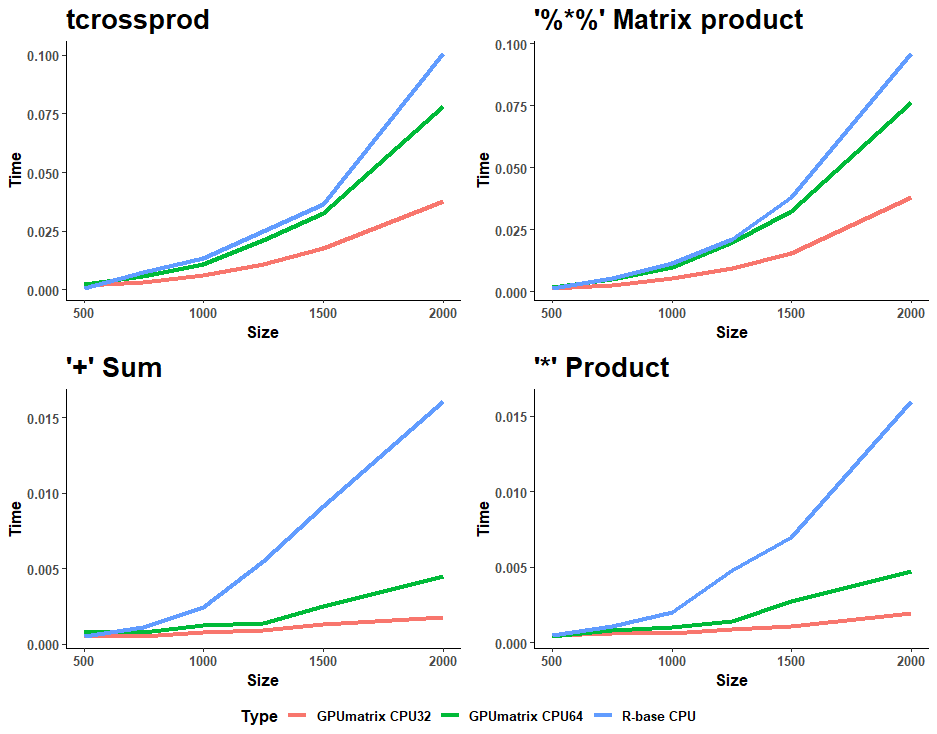
#GPUmatrix initialization with CPU option  
>Gm <- gpu.matrix(c(1:20)+40,10,2,device="cpu")  
#Show CPU matrix from GPUmatrix  
>Gm  
  
GPUmatrix  
torch\_tensor  
 41 51  
 42 52  
 43 53  
 44 54  
 45 55  
 46 56  
 47 57  
 48 58  
 49 59  
 50 60  
[ CPUDoubleType{10,2} ] #CPU tensor

R provides a standard BLAS version that is not multithreaded and not fully optimized for present computers. In the previous paragraphs, we compared the CUDA-GPU with MKL-R, i.e. using CUDA for linear algebra through torch or tensorflow or boosting the standard R with the Intel MKL library. Switching from Standard R to MKL R implies changing the default behavior of R and ther can be side-effects. For examples some standard packages such as igraph do not work in this case.

Torch and Tensorflow on the CPU are compiled using MKL as linear algebra library. Therefore, the performance between using MKL-R or using the GPUMatrix library on the CPU should be similar. The only differences would be related to the overhead from translating the objects or the different versions of the MKL library.

Interestingly, the standard R matrix operations are indeed slightly slower than using the GPUMatrix package -perhaps owing to a more recent version of the MKL library- (Fig 2), especially in element-wise operations, where MKL-R does not seem to exploit the multithreaded implementation of the Intel MKL BLAS version and Torch and Tensorflow does.

In addition, if MKL-R is not implemented for float32 -since R does not include this type of variable-. The multiplication of float32 matrices on MKL-R does not use MKL and is, in fact, much slower than multiplying float64 matrices (data not shown). Torch and Tensorflow do include MKL for float32 and there is an improvement in the performance (they are around two-fold faster).



Computation time for GPUMatrix on CPU and MKL-R for different operations. Time is in seconds and Size=n where matrix is (n x n) dimension. There is a substantial speed performance in element-wise operations.

## 5.2 Using GPUMatrix with Tensorflow

As commented in the introduction and dependency section, GPUmatrix can be used with both TensorFlow and Torch. By default, the GPU matrix constructor is initialized with Torch tensors because, in our opinion, it provides an advantage in terms of installation and usage compared to TensorFlow. Additionally, it allows the use of GPUmatrix not only with GPU tensors but also with CPU tensors. To use GPUmatrix with TensorFlow, simply use the type attribute in the constructor function and assign it the value **“tensorflow”** as shown in the following example:

# library(GPUmatrix)  
>tensorflowGPUmatrix <- gpu.matrix(c(1:20)+40,10,2, type = "tensorflow")  
>tensorflowGPUmatrix  
  
GPUmatrix  
tf.Tensor(  
[[41. 51.]  
 [42. 52.]  
 [43. 53.]  
 [44. 54.]  
 [45. 55.]  
 [46. 56.]  
 [47. 57.]  
 [48. 58.]  
 [49. 59.]  
 [50. 60.]], shape=(10, 2), dtype=float64)

#### References

Bates D, Maechler M, Jagan M (2022). Matrix: Sparse and Dense Matrix Classes and Methods. R package version 1.5-3, <https://CRAN.R-project.org/package=Matrix>.

Schmidt D (2022). “float: 32-Bit Floats.” R package version 0.3-0, <https://cran.r-project.org/package=float>.

Falbel D, Luraschi J (2022). torch: Tensors and Neural Networks with ‘GPU’ Acceleration. R package version 0.9.0, <https://CRAN.R-project.org/package=torch>.

Allaire J, Tang Y (2022). tensorflow: R Interface to ‘TensorFlow’. R package version 2.11.0,  <https://CRAN.R-project.org/package=tensorflow>.