GPUmatrix: Seamlessly harness the power of GPU computing in R

by César Lobato-Fernández, Juan A. Ferrer-Bonsoms, and Angel Rubio

Abstract GPUs are invaluable for data analysis, particularly in statistics and linear algebra, but integrating them with R has been challenging due to the lack of transparent, easily maintainable packages that don't require significant code alterations. Recognizing this gap, we've developed the GPUmatrix package, now available on CRAN, which emulates the Matrix package's behavior, enabling R to harness the power of GPUs for computations with minimal code adjustments. GPUmatrix supports both single (FP32) and double (FP64) precision data types and includes support for sparse matrices, ensuring broad applicability. Designed for ease of use, it requires only slight modifications to existing code, leveraging the Torch or Tensorflow R packages for GPU operations. We've validated its effectiveness in various statistical and machine learning tasks, including non-negative matrix factorization, logistic regression, and general linear models, and provided a comparative analysis of GPU versus CPU performance, highlighting significant efficiency gains.

1 Introduction

R(R Core Team, 2022) is a programming language broadly used in statistics and computational biology. R is part of a collaborative and open project where users can publish packages that extend its base configuration. CRAN (Comprehensive R Archive Network) and Bioconductor are repositories where R packages are stored after passing strict code quality criteria.

Graphics Processing Units (GPUs) perform computations in parallel, making them exceptionally powerful for tasks involving large amounts of data processing. At the hardware level, GPUs consist of many processing cores that can execute thousands of threads simultaneously. These threads are organised into groups called warps (in NVIDIA GPUs) or wavefronts (in AMD GPUs), which typically contain 32 or 64 threads. All threads within a warp operate in lockstep, following the Single Instruction, Multiple Thread (SIMT) model. This means that each thread executes the same instruction at the same time, but on different data elements, enabling massive parallelism and high throughput for suitable tasks.

While the SIMT architecture excels at parallel execution, it presents challenges when threads within a warp need to follow different execution paths due to conditional statements or branching. Matrix operations are particularly well suited to GPUs due to their regular computational patterns and minimal branching. Operations such as matrix multiplication, element-wise arithmetic and linear algebra computations involve applying the same operation to large data sets, which fits perfectly with the SIMT execution model. This makes matrix computations highly efficient on GPUs, leading to significant performance gains in applications such as machine learning, scientific simulation and data analysis.

Statistical and machine learning methods are mostly based on linear algebra operations. Using a GPU for these operations greatly increases the computational power. Since R is a language designed for these types of methods, adapting it to use the GPU would result in a significant computational improvement.

There have been previous attempts to exploit the power of the GPU using R packages. Some of these have been method-specific - such as cuRnet (Bonnici et al., 2018) for graph analysis, qrpca (de Souza et al., 2022) for PCA analysis, rgtsvm (Wang et al., 2017) for support vector machines, **xgboost** (Chen and Guestrin, 2016) for extreme grading boosting, and **gputools** (Buckner et al., 2009) for microarray analysis. **gpuR** (Determan, 2017), on the other hand, provided "GPU-enabled functions in a simple and accessible way". gpuMagic (Wang and Morgan, 2022), allows users to compile R functions in OpenCL and run the code on the GPU. None of these are currently available on CRAN or Bioconductor except gpuMagic and xgboost.

torch (Falbel and Luraschi, 2023) and **tensorflow** (Allaire and Tang, 2022) are other GPU-enabled packages for R that target machine learning methods whose workhorse are tensor-like objects, a generalization of the concepts of scalar, vector, and matrix. These packages provide the functionality of the Python packages PyTorch and Tensorflow respectively. Both are available on CRAN and are GPU enabled. Both are maintained by developers from RStudio (RStudio Team, 2020) (now Posit). Both packages represent a rather complex learning process for the standard R user as they focus on machine learning rather than linear algebra and the syntax is reminiscent of its ''Pythonic" origins.

Here we present **GPUmatrix** which, by including methods for most statistical and linear algebra

functions, allows the full functionality of R to be used for GPU computation with almost no learning for the user, making it very easy to adapt to existing programs. GPUmatrix uses either Torch or Tensorflow internally to perform the computations.

The paper is organized as follows: in the first section we give a general description of the package, its objects and features. In the second section we show several statistical applications of the package and their respective performance improvements. The third section contains the results of the comparison of the speed of the operations and functions using either CPUs or GPUs. In the last section we discuss these results.

2 Overall Description

GPUmatrix is based on the S4 object structure provided by R, and mimics the behavior of the **Matrix** package (Bates et al., 2023). It implements sparse matrices and **float** (Schmidt, 2017) data types (from Matrix and float packages) as part of the gpu.matrix objects. It includes casting operations i.e conversions from one object type to a different one between gpu.matrix objects, Matrix -including sparse-, float32, and the default matrix objects in R.

For a list of all available methods, see Tables T2, T3 and T4 in the package vignette. These include the standard operations on matrices (sum, difference, inverse, standard matrix product, Hadamard product, etc.), access to matrix elements (entry, rows, columns, submatrices, etc.), matrix factorizations (qr, svd, chol, lu, etc.), algorithms for sorting, computing the fft of a vector, etc. GPUmatrix also includes operations from the **matrixStats** package (Bengtsson, 2022) (rowMedians, rowMaxs, etc.)

Installing GPUmatrix itself is straightforward. It depends on the installation of either torch or tensorflow, which is more involved. In turn, if GPU processing is required, torch and tensorflow depend on specific versions of CUDA (not necessarily identical). In the GPUmatrix vignette we have included a brief explanation on how to install these packages.

GPUmatrix requires either torch or tensorflow to be installed. GPUmatrix can be used if any of them is installed. For this reason, neither of them are strictly required(if the other is available) an we have included no dependencies between GPUmatrix and torch or tensorflow.

2.1 Examples of Use

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

```
library(GPUmatrix)

#> Torch tensors allowed

#> Your Torch installation does not have CUDA tensors available. Please check the Torch requirements and installat

#R matrix initialization
m <- matrix(c(1:10)+40,5,2)
#Show CPU matrix
m

#>      [,1] [,2]
#> [1,]   41   46
#> [2,]   42   47
#> [3,]   43   48
#> [4,]   44   49
#> [5,]   45   50

#GPU matrix initialization
Gm <- gpu.matrix(c(1:10)+40,5,2)
#Show GPU matrix
Gm

#> GPUmatrix
#> torch_tensor
```

```
#>  41  46
#>  42  47
#>  43  48
#>  44  49
#>  45  50
#> [ CPUDoubleType{5,2} ]
```

Although the indexing of tensors in both torch and tensorflow is 0-based, the indexing of GPUmatrix objects is 1-based, making it as close as possible to working with native R matrices and more convenient for the user. 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 their elements and assigning values to them. The concatenation operators `rbind()` and `cbind()` work independently of the type of matrices being concatenated, resulting in a `gpu.matrix()`:

```
Gm[c(2,3),1]
```

```
#> GPUmatrix
#> torch_tensor
#>  42
#>  43
#> [ CPUDoubleType{2,1} ]
```

```
Gm[,2]
```

```
#> GPUmatrix
#> torch_tensor
#>  46
#>  47
#>  48
#>  49
#>  50
#> [ CPUDoubleType{5,1} ]
```

```
Gm2 <- cbind(Gm[c(1,2),], Gm[c(3,4),])
Gm2
```

```
#> GPUmatrix
#> torch_tensor
#>  41  46  43  48
#>  42  47  44  49
#> [ CPUDoubleType{2,4} ]
```

```
Gm2[1,3] <- 0
Gm2
```

```
#> GPUmatrix
#> torch_tensor
#>  41  46   0  48
#>  42  47  44  49
#> [ CPUDoubleType{2,4} ]
```

It is also possible to initialize the data with NaN values:

```
Gm3 <- gpu.matrix(nrow = 2,ncol=3)
Gm3[,2]
```

```
#> GPUmatrix
#> torch_tensor
#> nan
#> nan
#> [ CPUDoubleType{2,1} ]
```

```
Gm3[1,2] <- 1
Gm3
```

Table 1: Cast options from other packages. If back cast is TRUE, then it is possible to convert a gpu.matrix to this object and vice versa. If is FALSE, it is possible to convert these objects to gpu.matrix but not vice versa.

MatrixClass	Package	DataTypeDefault	SPARSE	BackCast
matrix	base	float64	FALSE	TRUE
data.frame	base	float64	FALSE	TRUE
integer	base	float64	FALSE	TRUE
numeric	base	float64	FALSE	TRUE
dgeMatrix	Matrix	float64	FALSE	FALSE
ddiMatrix	Matrix	float64	TRUE	FALSE
dpoMatrix	Matrix	float64	FALSE	FALSE
dgCMatrix	Matrix	float64	TRUE	FALSE
float32	float	float32	FALSE	FALSE
torch_tensor	torch	float64	Depends of tensor type	TRUE
tensorflow.tensor	tensorflow	float64	Depends of tensor type	TRUE

```
#> GPUmatrix
#> torch_tensor
#> nan  1 nan
#> nan nan nan
#> [ CPUDoubleType{2,3} ]

Gm3[1,3] <- 0
Gm3

#> GPUmatrix
#> torch_tensor
#> nan  1  0
#> nan nan nan
#> [ CPUDoubleType{2,3} ]
```

These examples demonstrate that, contrary to standard R, subsetting a gpu.matrix —even when selecting only one column or row— still results in a gpu.matrix. This behavior is analogous to using `drop=FALSE` in standard R. The default standard matrices in R have limitations. The only allowed numeric data types are int and float64. It neither natively allows the creation or handling of sparse matrices. To make up for this lack of functionality, other R packages hosted in CRAN have been created to manage these types.

2.2 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 GPUmatrix type have been implemented (Table 1).

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 to create a GPUmatrix.

Create 'Gm' from 'm' matrix R-base:

```
m <- matrix(c(1:10)+40,5,2)
Gm <- gpu.matrix(m)
Gm

#> GPUmatrix
#> torch_tensor
#>  41  46
#>  42  47
#>  43  48
#>  44  49
#>  45  50
#> [ CPUDoubleType{5,2} ]
```

Create 'Gm' from 'M' with Matrix package:

```
library(Matrix)
M <- Matrix(c(1:10)+40,5,2)
Gm <- gpu.matrix(M)
Gm

#> GPUmatrix
#> torch_tensor
#>  41  46
#>  42  47
#>  43  48
#>  44  49
#>  45  50
#> [ CPUDoubleType{5,2} ]
```

Create 'Gm' from 'mfloat32' with float package:

```
library(float)
mfloat32 <- fl(m)
Gm <- gpu.matrix(mfloat32)
Gm

#> GPUmatrix
#> torch_tensor
#>  41  46
#>  42  47
#>  43  48
#>  44  49
#>  45  50
#> [ CPUFloatType{5,2} ]
```

Interestingly, GPUmatrix returns a float32 data type matrix if the input is a float matrix. It is also possible to a gpu.matrix create 'Gms' type sparse from 'Ms' type sparse dgCMatrix, dgeMatrix, ddiMatrix or dpoMatrix with Matrix package:

```
Ms <- Matrix(sample(0:1, 10, replace = TRUE), nrow=5, ncol=2, sparse=TRUE)
Ms

#> 5 x 2 sparse Matrix of class "dgCMatrix"
#>
#> [1,] . 1
#> [2,] 1 .
#> [3,] . 1
#> [4,] . .
#> [5,] . 1

Gms <- gpu.matrix(Ms)
Gms

#> GPUmatrix
#> torch_tensor
#> [ SparseCPUFloatType{}
#> indices:
#>  0  1  2  4
#>  1  0  1  1
#> [ CPULongType{2,4} ]
#> values:
#>  1
#>  1
#>  1
#>  1
#> [ CPUFloatType{4} ]
#> size:
#> [5, 2]
#> ]
```

2.3 Data type and sparsity

The data types allowed for GPUmatrix are: **float64**, **float32**, **int**, **bool** or **logical**, **complex64** and exclusively in torch **complex32**. We can create a GPU matrix with a specific data type using the `dtype**` parameter of the `gpu.matrix()**` constructor function. It is also possible 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 return 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 change it 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(1:10)+40,5,2, dtype = "float32")
Gm32
```

```
#> GPUmatrix
#> torch_tensor
#>  41  46
#>  42  47
#>  43  48
#>  44  49
#>  45  50
#> [ CPUFloatType{5,2} ]
```

Creating a non sparse matrix with data type float32 from a sparse matrix type float64:

```
Ms <- Matrix(sample(0:1, 10, replace = TRUE), nrow=5, ncol=2, sparse=TRUE)
Gm32 <- gpu.matrix(Ms, dtype = "float32", sparse = F)
Gm32
```

```
#> GPUmatrix
#> torch_tensor
#>  0  1
#>  0  1
#>  1  0
#>  0  1
#>  1  1
#> [ CPUFloatType{5,2} ]
```

Convert Gm32 in sparse matrix Gms32:

```
Gms32 <- to_sparse(Gm32)
Gms32
```

```
#> GPUmatrix
#> torch_tensor
#> [ SparseCPUFloatType{}
#> indices:
#>  2  4  0  1  3  4
#>  0  0  1  1  1  1
#> [ CPULongType{2,6} ]
#> values:
#>  1
#>  1
#>  1
#>  1
#>  1
#>  1
#> [ CPUFloatType{6} ]
#> size:
#> [5, 2]
#> ]
```

Convert data type Gms32 into float64:

```
Gms64 <- Gms32
dtype(Gms64) <- "float64"
Gms64
```

```
#> GPUmatrix
#> torch_tensor
#> [ SparseCPUDoubleType{}
#> indices:
#>  2  4  0  1  3  4
#>  0  0  1  1  1  1
#> [ CPULongType{2,6} ]
#> values:
#>  1
#>  1
#>  1
#>  1
#>  1
#>  1
#> [ CPUDoubleType{6} ]
#> size:
#> [5, 2]
#> ]
```

2.4 GPUmatrix functions

Arithmetic and comparison operators

GPUmatrix supports all of the 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 packages mentioned above.

```
(Gm + Gm) == (m + m)
```

```
#>      [,1] [,2]
#> [1,] TRUE TRUE
#> [2,] TRUE TRUE
#> [3,] TRUE TRUE
#> [4,] TRUE TRUE
#> [5,] TRUE TRUE
```

```
(Gm + M) == (mfloat32 + Gm)
```

```
#>      [,1] [,2]
#> [1,] TRUE TRUE
#> [2,] TRUE TRUE
#> [3,] TRUE TRUE
#> [4,] TRUE TRUE
#> [5,] TRUE TRUE
```

```
(M + M) == (mfloat32 + Gm)
```

```
#>      [,1] [,2]
#> [1,] TRUE TRUE
#> [2,] TRUE TRUE
#> [3,] TRUE TRUE
#> [4,] TRUE TRUE
#> [5,] TRUE TRUE
```

As seen in the previous examples, the comparison operators (`==`, `!=`, `\textgreater{}`, `\textless{}`, `\textgreater{=}`, `\textless{=}`) also work following the same dynamic of the arithmetic operators.

Table 2: Mathematical operators that accept a gpu.matrix as input.

MathematicalOperators	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)

Table 3: Complex operators that accept a gpu.matrix with complex type as input.

MathematicalOperators	Usage
Re	Re(Gm)
Im	Im(Gm)
Conj	Conj(Gm)
Arg	Arg(Gm)
Mod	Mod(Gm)

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 (Table 2).

Complex operators

There are certain functions only applicable to numbers of complex type. In R these functions are grouped as complex operators and all of them are available for GPUmatrix matrices with the same functionality as in R base. `Gm` is a `gpu.matrix` variable (Table 3).

Other functions

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

```
library(GPUmatrix)
m <- matrix(c(1:10)+40,5,2)
Gm <- gpu.matrix(c(1:10)+40,5,2)

head(tcrossprod(m),2)

#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] 3797 3884 3971 4058 4145
#> [2,] 3884 3973 4062 4151 4240

head(tcrossprod(Gm),2)

#> GPUmatrix
#> torch_tensor
#>  3797  3884  3971  4058  4145
#>  3884  3973  4062  4151  4240
#> [ CPUDoubleType{2,5} ]

Gm <- tail(Gm,3)
rownames(Gm) <- c("a","b","c")
tail(Gm,2)

#> GPUmatrix
#> torch_tensor
#>  44  49
#>  45  50
#> [ CPUDoubleType{2,2} ]
#> rownames: b c

colMaxs(Gm)

#> [1] 45 50
```

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

2.5 Using GPUMatrix on CPU

In the GPUmatrix constructor, when using torch, 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, oriented towards algebraic operations in R using the GPU, it is hosted on the GPU by default, but it allows the same functionalities to be used with 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(1:10)+40,5,2,device="cpu")
#Show CPU matrix from GPUmatrix
Gm

#> GPUmatrix
#> torch_tensor
#>  41  46
#>  42  47
#>  43  48
#>  44  49
#>  45  50
#> [ CPUDoubleType{5,2} ]
```

Notice that instead of CUDADoubleType, now the object is CPUDoubleType.

The standard distribution of R includes a version of basic linear algebra subprograms (BLAS) that is not multithreaded and not fully optimized for present computers. R can be modified to use non-standard BLAS libraries such as OpenBlas, Intel MKL or Accelerate. Switching from Standard R

to R using MKL implies changing the default behavior of R and there can be side-effects. For example, some standard packages such as igraph do not work in this case.

The standard BLAS is so slow, that we excluded it from the comparison. We have compared the CUDA-GPU (using GPUmatrix) with base R using Intel MKL (MKL-R).

Torch also runs on the CPU and its backend is MKL. 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 does.

In addition, MKL-R does not provide acceleration for float32 -since base R does not include this type of variable-. The multiplication of float32 matrices on MKL-R 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 about twice faster that the float64 counterparts).

3 Examples of Statistical Applications

The main advantage of GPUmatrix is its versatility: R code needs only minor changes to adapt it to work in the CPU. We are showing here three statistical applications where its advantages are more apparent.

3.1 Non negative factorization of a matrix

The non-negative factorization (NMF) of a matrix is an approximate factorization were an initial matrix $\mathbf{V}$ is approximated by the product of two matrices $\mathbf{W}$ and $\mathbf{H}$ so that,

$$\mathbf{V}_{m\times n} \approx \mathbf{W}_{m\times k}\mathbf{H}_{k\times n}$$

We have implemented our own non-negative matrix factorization (NMF) function using Lee and Seung(Lee and Seung, 1999) multiplicative update rules.

These rules are

$$\mathbf{W}^{n+1}_{[i,j]} \leftarrow \mathbf{W}^{n}_{[i,j]} \frac{\left(\mathbf{V}\left(\mathbf{H}^{n+1}\right)^T\right)_{[i,j]}}{\left(\mathbf{W}^n\mathbf{H}^{n+1}\left(\mathbf{H}^{n+1}\right)^T\right)_{[i,j]}}$$

and

$$\mathbf{H}^{n+1}_{[i,j]} \leftarrow \mathbf{H}^{n}_{[i,j]} \frac{\left(\left(\mathbf{W}^n\right)^T\mathbf{V}\right)_{[i,j]}}{\left(\left(\mathbf{W}^n\right)^T\mathbf{W}^n\mathbf{H}^n\right)_{[i,j]}}$$

to update the $\mathbf{W}$ and $\mathbf{H}$ respectively.

The implemented function is `NMFgpumatrix`. This function operates in the same way with basic R matrices as with GPUmatrix matrices, and it does not require any additional changes beyond initializing the input matrix as a GPUmatrix. Indeed, the input matrices $\mathbf{W}$, $\mathbf{H}$, and $\mathbf{V}$ can be either gpu.matrix or R base matrices interchangeably. Figure 1 shows that using GPUmatrix boosts the performance using both GPU and CPU. This improvement is especially apparent with float32 matrices.

3.2 Logistic regression of large models

Logistic regression is a widespread statistical analysis technique that is the "first to test" method for classification problems where the outcome is binary. R-base implements it in the `glm` function. However, `glm` can be very slow for big models and, in addition, does not accept sparse coefficient matrices as input. In this example, we have implemented a logistic regression solver that accepts as input both dense or sparse matrices.

The developed function performs the logistic regression using the Conjugate Gradient method (CG) Figure 2. This method has shown to be very effective for logistic regression of big models (Minka, 2003). The code is general enough to accommodate standard R matrices, sparse matrices from the Matrix package and, more interestingly, GPUmatrices from the GPUmatrix package.

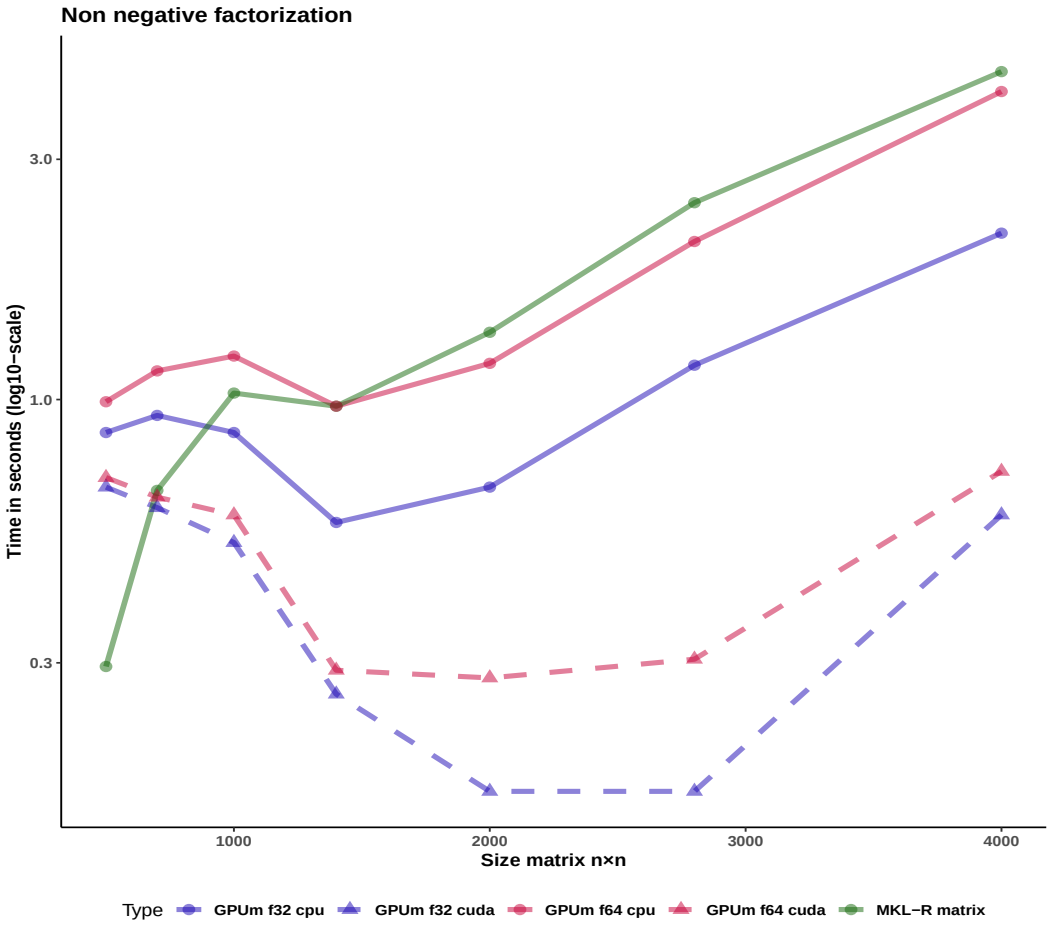

Figure 1: Computation time (in seconds) of non-negative factorization for MKL-R (i.e. R with the optimized MKL BLAS library, solid green), solid lines for CPU, dashed lines for GPU with CUDA, pink lines for GPUmatrix with float64, and blue lines for GPUmatrix with float32. Time shown in y-axis is in logarithmic scale. All calculations are performed on square matrices. The x-axis represents the number of rows in the matrices. The internal size of the factorization is 10.

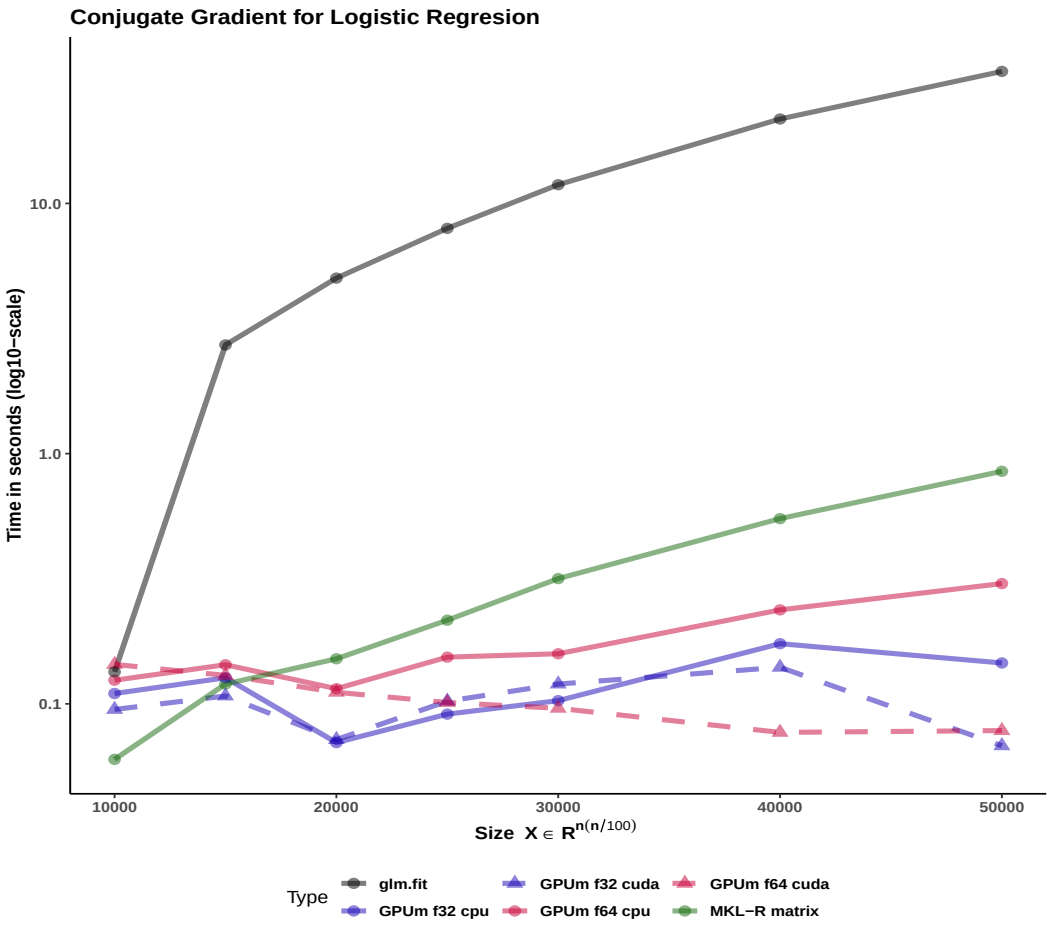

Figure 2: Computation time (in seconds) of the logistic regression using the conjugate gradient method for MKL-R (i.e. R with the optimized MKL BLAS library, solid green), solid lines for CPU, dashed lines for GPU with CUDA, pink lines for GPUmatrix with float64, and blue lines for GPUmatrix with float32. Time shown in y-axis is in logarithmic scale. The calculations are performed on random matrices whose size are n x (n/100). Therefore, the leftmost part of the graph shows the computing time for a 10,000 x 100 matrix and the rightmost part a 50,000 x 500 matrix.

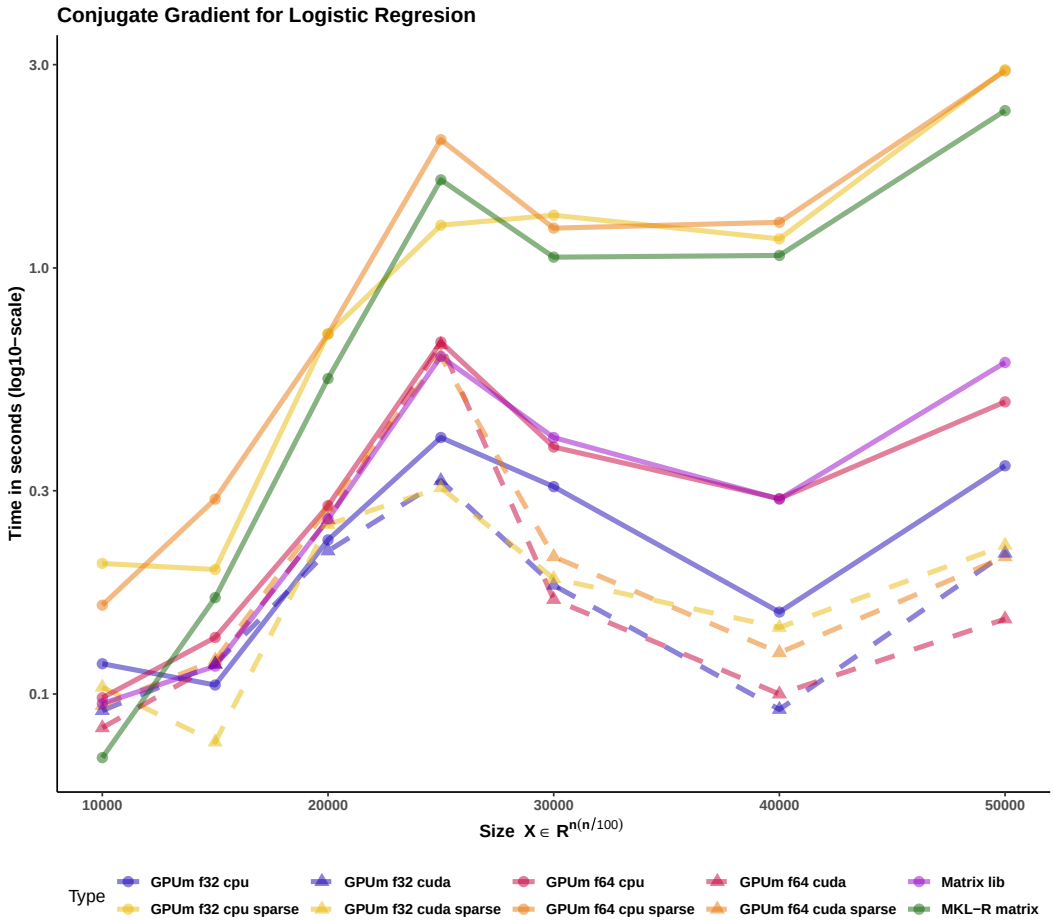

Figure 3: Computation time (in seconds) of the logistic regression using the conjugate gradient method in a sparse matrix. Solid green for MKL-R dense case (i.e. the computation is performed without any consideration of the sparsity of the matrix). Solid lines for CPU, dashed lines for GPU with CUDA, pink lines for GPUmatrix dense with float64, blue lines for GPUmatrix dense with float32, yellow lines for GPUmatrix sparse with float32, orange lines for GPUmatrix sparse with float64. Violet line, using Matrix package(that implicitly considers the matrix to be sparse). Time shown in y-axis is in logarithmic scale. The calculations are performed on random matrices whose size are n x (n/100). Therefore, the leftmost part of the graph shows the computing time for a 10,000 x 100 matrix and the rightmost part a 50,000 x 500 matrix.

We would like to stress several points here. Firstly, the conjugate gradient is an efficient technique that outperforms glm.fit in this case. Secondly, this code runs on matrix, Matrix or GPUmatrix objects without the need of carefully selecting the type of input. Thirdly, using the GPUmatrix accelerates the computation time two-fold if compared to standard R (and more than ten fold if compared to glm.fit function)

We have tested the function also using a sparse input. In this case, the memory requirements are much smaller. However, there are no advantages in execution time (despite the sparsity is 90 %). Torch (and Tensorflow) for R only provides a type of sparse matrices: the "coo" coding where each element is described by its position (i and j) and its value. Matrix (and torch and tensorflow for Python) include other storage models (column compressed format, for example) where the matrix computations can be better optimized. It seems that there is still room for improvement in the sparse matrix algebra in Torch for R.

Sparse Matrix -that includes the column compressed form- performs extraordinary well in this case Figure 3. Despite Matrix is single-threaded, it is roughly ten times faster than GPUmatrix in sparse matrices.

3.3 General Linear Models

One of the most frequently used functions in R is `glm`, that stands for generalized linear models. In turn, `glm` relies on the `glm.fit`. This function uses the iteratively reweighted least squares algorithm to provide the solution of a generalized linear model. glm.fit, subsequently, calls a C function to do the "hard work" for solving the linear models, including the least squares solution of the intermediate linear systems.

Since glm.fit calls a C function is not easy to develop a "plug-in" substitute using GPUmatrix. Specifically, the qr algorithm in torch or tensorflow for R and in the C function (that is similar to base R) substantially differ: in torch the function returns directly the Q and R matrices whereas the C function called by glm.fit returns a matrix and several auxiliary vectors that can be used to reconstruct the Q and R matrices.

One side-effect of the dependency of base glm on a C function is that, even if R is compiled to use an optimized linear algebra library, the function does not exploit that and the execution time is much longer than expected for this task.

In order to ameliorate this problem, there are other packages (**fastlm** and **speedglm**) that mimic the behavior of the base glm. fastlm relies on using multithreaded RcppEigen functions to boost the performance. On the contrary, speedglm is written in plain R and the performance is increased especially if R runs using an optimized BLAS library.

We have used speedglm as starting point for our glm implementation in the GPU since all the code is written in R and includes no calls to external functions. Specifically, the most time-consuming task for large general linear models, is the solution of the intermediate least squares problems. In the GPUmatrix implementation, we used most of the code of speed.glm. The only changes are casts on the matrix of the least squares problem and the corresponding independent term to solve the problem using the GPU. Timing results of the last section show that `GPUglm` outperforms speed.glm for large models.

The following script illustrates how to use the `GPUglm` function in toy examples.

```
# Standard glm

counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
glm.D93 <- glm(counts ~ outcome + treatment, family = poisson())
summary(glm.D93)

#>
#> Call:
#> glm(formula = counts ~ outcome + treatment, family = poisson())
#>
#> Coefficients:
#>              Estimate Std. Error z value Pr(>|z|)
#> (Intercept)  3.045e+00  1.709e-01  17.815   <2e-16 ***
#> outcome2    -4.543e-01  2.022e-01  -2.247   0.0246 *
#> outcome3    -2.930e-01  1.927e-01  -1.520   0.1285
#> treatment2   1.338e-15  2.000e-01   0.000   1.0000
#> treatment3   1.421e-15  2.000e-01   0.000   1.0000
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for poisson family taken to be 1)
#>
#>     Null deviance: 10.5814  on 8  degrees of freedom
#> Residual deviance:  5.1291  on 4  degrees of freedom
#> AIC: 56.761
#>
#> Number of Fisher Scoring iterations: 4

# Using speedglm
library(speedglm)
sglm.D93 <- speedglm(counts ~ outcome + treatment, family = poisson())
summary(sglm.D93)
```

```
#> Generalized Linear Model of class 'speedglm':
#>
#> Call:  speedglm(formula = counts ~ outcome + treatment, family = poisson())
#>
#> Coefficients:
#>  ---------------------------------------------------------------------
#>               Estimate Std. Error    z value  Pr(>|z|)
#> (Intercept)  3.045e+00     0.1709  1.781e+01 5.427e-71 ***
#> outcome2    -4.543e-01     0.2022 -2.247e+00 2.465e-02 *
#> outcome3    -2.930e-01     0.1927 -1.520e+00 1.285e-01
#> treatment2  -5.309e-17     0.2000 -2.655e-16 1.000e+00
#> treatment3  -2.655e-17     0.2000 -1.327e-16 1.000e+00
#>
#> ---------------------------------------------------------------------
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> ---
#> null df: 8; null deviance: 10.58;
#> residuals df: 4; residuals deviance: 5.13;
#> # obs.: 9; # non-zero weighted obs.: 9;
#> AIC: 56.76132; log Likelihood: -23.38066;
#> RSS: 5.2; dispersion: 1; iterations: 4;
#> rank: 5; max tolerance: 2e-14; convergence: TRUE.

# GPU glm
library(GPUmatrix)
gpu.glm.D93 <- GPUglm(counts ~ outcome + treatment, family = poisson())
summary(gpu.glm.D93)

#> Generalized Linear Model of class 'summary.GPUglm':
#>
#> Call:  glm(formula = ..1, family = ..2, method = "glm.fit.GPU")
#>
#> Coefficients:
#>  ---------------------------------------------------------------------
#>               Estimate Std. Error    z value  Pr(>|z|)
#> (Intercept)  3.045e+00     0.1709  1.781e+01 5.427e-71 ***
#> outcome2    -4.543e-01     0.2022 -2.247e+00 2.465e-02 *
#> outcome3    -2.930e-01     0.1927 -1.520e+00 1.285e-01
#> treatment2   2.049e-15     0.2000  1.025e-14 1.000e+00
#> treatment3   2.099e-15     0.2000  1.050e-14 1.000e+00
#>
#> ---------------------------------------------------------------------
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> ---
#> null df: 8; null deviance: 10.58;
#> residuals df: 4; residuals deviance: 5.13;
#> # obs.: 9; # non-zero weighted obs.: 9;
#> AIC: 56.76132; log Likelihood: -23.38066;
#> RSS: 5.2; dispersion: 1; iterations: 4;
#> rank: 5; max tolerance: 2.28e-14; convergence: TRUE.
```

Results are identical in this test. Figure 4 compares the timing of the implementation using speedglm, glm and the implementation of glm in GPUmatrix. GPUmatrix outperforms speedglm with R-MKL. This is especially apparent for float32 matrices.

4 Performance comparison

We have compared the computation time for different matrix functions, different precision types and running the operations either on the GPU or on the CPU (using both GPUmatrix and R with MKL as linear algebra library).

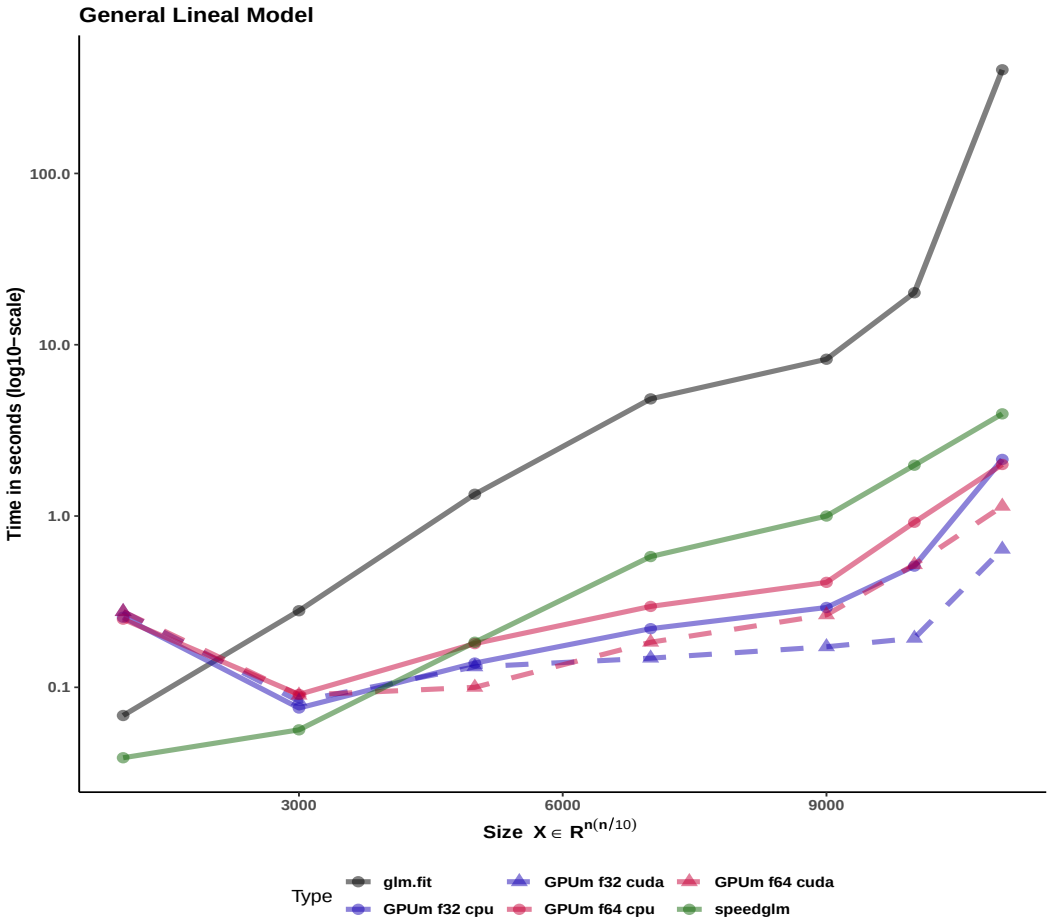

Figure 4: Computation time (in seconds) of general linear model using *speedglm* function with MKL-R matrix (i.e. R with the optimized MKL BLAS library, solid green), solid black line for glm function, solid lines for CPU, dashed lines for GPU with CUDA, pink lines for GPUmatrix with float64, and blue lines for GPUmatrix with float32. Time shown in y-axis is in logarithmic scale. The calculations are performed on random matrices whose size are n x (n/10). Therefore, the leftmost part of the graph shows the computing time for a 1,000 x 100 matrix and the rightmost part a 10,000 x 1000 matrix.

The functions that we tested are * (Hadamard or element-wise product of matrices), exp, (exponential of each element of a matrix), rowMeans (means of the rows of a matrix), \%*\% (standard product of matrices), solve (inverse of a matrix) and svd (singular value decomposition of a matrix).

These functions were tested on square matrices whose row sizes are 500, 700, 1000, 1400, 2000, 2800 and 4000.

Figure 5 compares the different computational architectures, namely, CPU -standard R matrices running on MKL on FP64-, CPU64 -GPUmatrix matrices computed on the CPU with FP64-, CPU32 -similar to the previous using FP32-, GPU64 -GPUmatrix matrices stored and computed on the GPU with FP64- and GPU32 -identical to the previous using FP32-.

It is important to note that the y-axis is in logarithmic scale. For example, the element-wise product of a matrix on GPU32 is around five times faster than the same operation on CPU. The results show that the GPU is particularly effective for element-wise operations (Hadamard product, exponential of a matrix). For these operations, it is easier to fully utilize the huge number of cores of a GPU. R-MKL seems to use a single core to perform element-wise operations. The torch implementation is much faster, but not as much as using the GPU. rowMeans is also faster on the GPU than on the CPU. In this case, the GPUmatrix CPU implementation is on par with the GPU. When the operations become more complex, as in the standard product of matrices and computing the inverse, CPU and GPU (using double precision) are closer to each other. However, GPU32 is still much faster than its CPU32 counterpart. Finally, it is not advisable -in terms of speed- to use the GPU for even more complex operations such as the SVD. In this case, GPU64 is the slowest method. GPU32 hardly stands up the comparison with CPU32.

Regarding sparse matrices, Torch or Tensorflow for R have very limited support for them. In many cases, it is necessary to cast the sparse matrix to a full matrix and perform the operations on the full matrix. However, there are two operations that do not require this casting: element-wise multiplication of matrices and multiplication of a sparse matrix and a full matrix. We created four sparse matrices. The size of two of them (small ones) is 2,000 x 2,000. The fraction of nonzero entries is 0.01 and 0.001. The size of the other two (large ones) is 20,000 x 20,000. The fraction of non-null entries is also 0.01 and 0.001. Figure 6 shows the results for these matrices. The element-wise multiplication is faster using GPUmatrix. On the contrary, the time to multiply a sparse and a full matrices is similar using Matrix and GPUmatrix. As mentioned earlier, the implementation of sparse operations in torch or tensorflow for R is far from perfect, probably, because the only storage mode is "coo" in their respective R packages.

5 Discusion

GPUmatrix is an R package that facilitates the use of GPUs for linear algebra operations. It relies heavily on either the torch or tensorflow packages. GPUmatrix will work as long as torch or tensorflow work. Currently, the R implementation of these packages lags behind their Python counterparts. For example, in the case of sparse matrices, only the coo sparse tensor implementation is allowed, although compressed row and column formats are readily available in Python torch. Nevertheless, we have shown that GPUmatrix is a convenient package for easily integrating GPU computations into R programs.

GPUmatrix, like any software that performs a computation on the GPU, has some drawbacks. There is a large overhead associated with moving data from main memory to GPU memory. For simple operations with small objects, the time required to move the object can be greater than the computation itself. We have taken special care to avoid unnecessary movements from CPU to GPU memory and vice versa. In some cases – an operation with a matrix in the CPU and a matrix in the GPU – this overhead is unavoidable. In this case, we favored the GPU memory and the result will be a GPU matrix.

GPUmatrix also introduces some overhead due to casting objects from one type to another. These considerations lead to an important question: when is it worth using the GPU for linear algebra operations? The answer, of course, is that it depends. Here are some factors that affect the answer to this question.

It depends on the size of the operations and the type of operations. GPU is faster (and compensates the burden of moving the data from the CPU to the GPU memory) only when the operations are performed on large matrices (each dimension is at least in the hundreds or even the thousands of elements). It also depends on the nature of the operations. Element-wise operations are easily parallelized. However, more complex algorithms such as the SVD factorizations are much harder to implement in parallel, so the improvement from using the GPU is small (if any).

Of course, it also depends on the relative performance of the CPU and GPU. We tested GPUmatrix on several systems (Computer 1: i7-10700 plus RTX 2080Ti, Computer 2: i7-11700 plus RTX 3060 Ti,

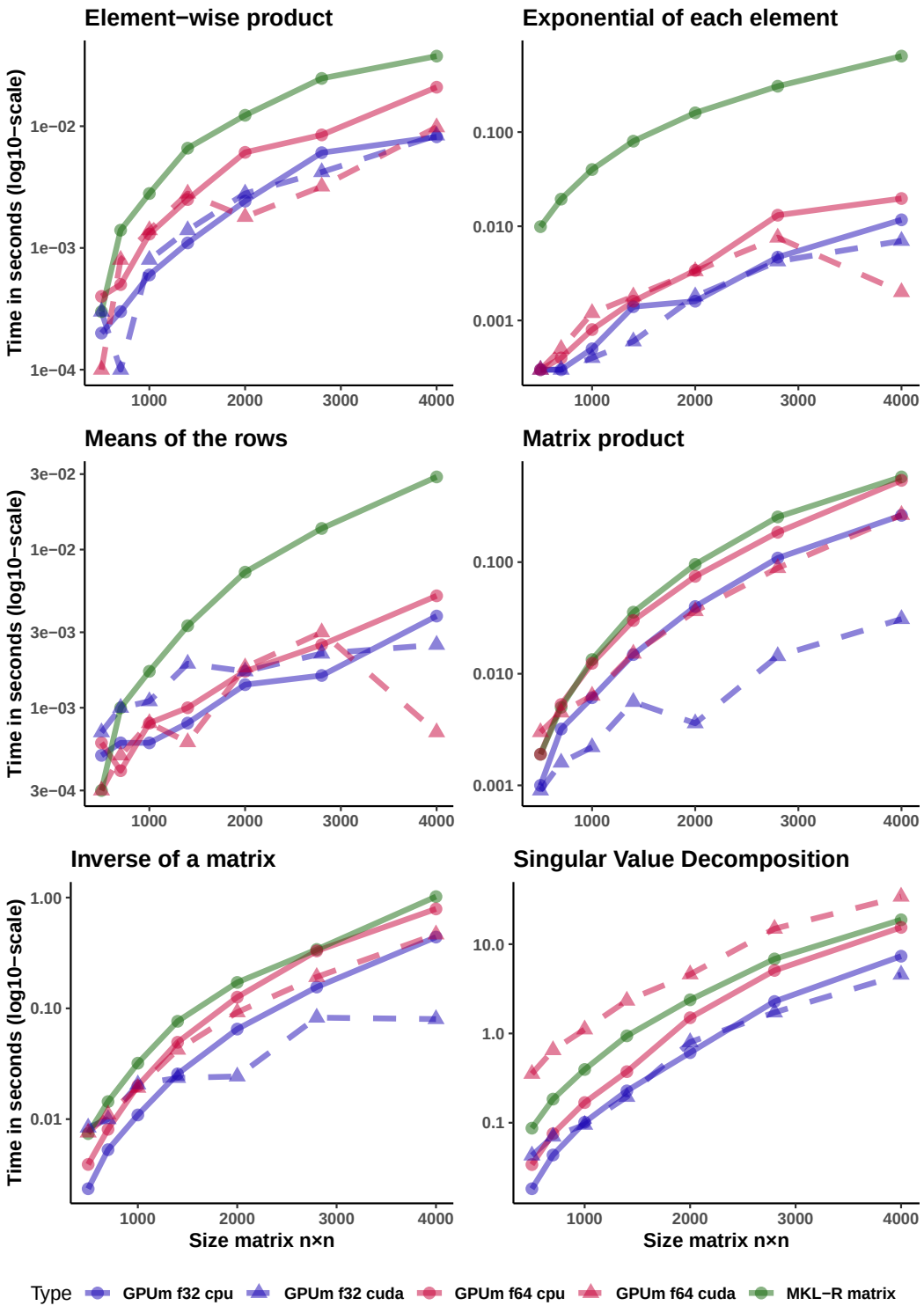

Figure 5: Computation time (in seconds) where MKL-R solid green, solid lines for CPU, dashed lines for GPU with CUDA, pink lines for GPUmatrix with float64 and blue lines for GPUmatrix with float32. All calculations are performed on square matrices. The x-axis represents the number of rows in the matrices. The operations are the element-wise or Hadamard product of two matrices, the exponential of each element of a matrix, the mean of the rows of a matrix, the standard matrix product, the inverse of a matrix, and the singular value decomposition of a matrix (SVD).

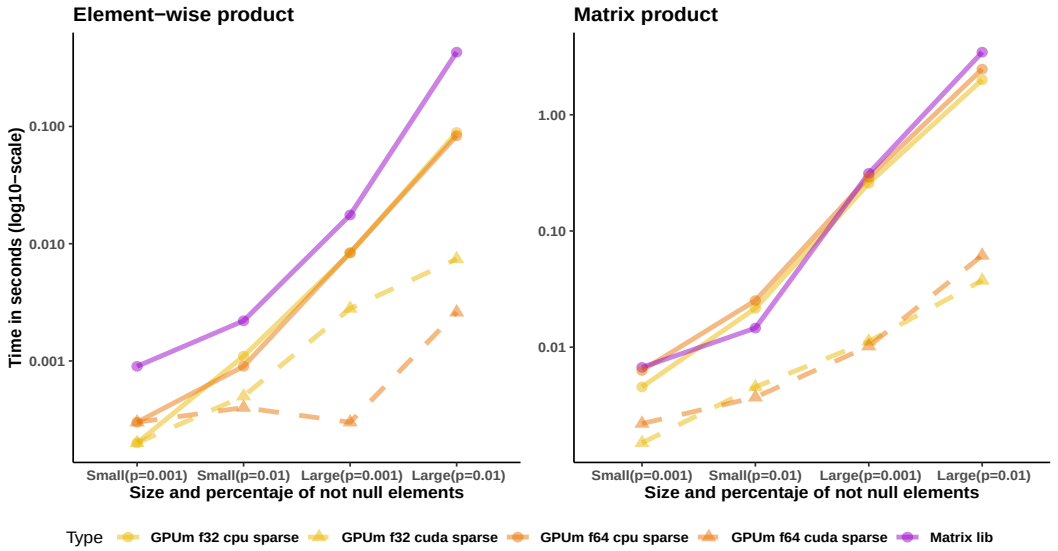

Figure 6: Computation time (in seconds) for the Matrix package (solid violet), yellow lines for GPUmatrix with float32, orange lines for GPUmatrix with float64, solid lines for CPU and dashed lines for GPU with CUDA. Time shown in y-axis is in logarithmic scale. The small model is a random square matrix of size 2,000 x 2,000. The proportion of non-zero elements is either 0.001 or 0.01. The large model is a 20,000 x 20,000 matrix with the same proportion of non-zero elements. The element-wise multiplication is performed on the sparse matrix. The right panel shows the time required to multiply these matrices by dense matrices whose sizes are 2,000 x 500 and 20,000 x 500, respectively.

Computer 3: i9-10940X and RTX A4000). Only results for Computer 1 are shown. These desktops can be considered mid-range to high-end in terms of both GPU and CPU. Computer 3 has an i9 processor with 14 cores. Although the graphics card is slightly better for this computer - unless using FP32 - the CPU is generally faster than the GPU in this case. In Computer 2, the RTX3060Ti is particularly fast with FP32 operations, and this definitely affects its performance.

One of the key points is that GPUs excel at FP32 operations. In the case of a CPU, FP32 operations are about twice as fast as FP64. In the case of GPUs, this gain theoretically skyrockets to 32 times faster (for the NVIDIA 2080Ti and the A4000) or even 64 times faster (for the 3060Ti). In practice, these factors are smaller, closer to 10 or 20. Still, the difference is huge, especially when compared to the CPU. If the operations can be done in FP32, then the GPU is usually the fastest method regardless of the type of operation.

Another crucial point is the linear algebra library installed on the CPU. The standard R library does not take advantage of multiple cores, and all operations run in a single thread. OpenBlas, Intel MKL, or Accelerate (for Mac OS) are popular substitutes to enhance the standard linear algebra library. Using these libraries requires tweaking the R installation, and in some cases there may be unwanted side effects. In addition, although OpenBlas (in Linux) and Accelerate accept FP32 data as input and speed up about two times compared to FP64 data, the Microsoft R implementation of MKL does not include any speedup for FP32. In fact, FP32 data from the float package is forced to use single-threaded operations. As a result, CPU operations for FP32 are even slower than for FP64 and therefore, these results were not included.

Torch is the default backbone for GPUmatrix, although Tensorflow can also be used. An advantage of Torch is that operations can be performed on either the GPU or the CPU. Torch implements a very efficient version of MKL on the CPU. Even if a GPU is not available, using GPUmatrix makes it possible to use MKL (even for FP32 data) without changing standard R. Since the standard R library is unchanged, there are no problems with packages that depend on specific functions of it. Operations on torch tensors are slightly faster than on tensorflow tensors (data not shown). Surprisingly, performing the operations on either CPU64 or CPU32 (i.e., using the GPUmatrix package) is slightly faster than using the CPU with R-MKL optimization.

GPUmatrix stands "on the shoulders of giants". As long as either torch or tensorflow is properly installed and GPU enabled, GPUmatrix will take advantage of it. Since these packages are the workhorses of Rstudio, they will be maintained for the foreseeable future, guaranteeing the functionality of GPUmatrix. GPUmatrix is a first approach to filling the gap of harnessing GPU power in R programs. It is easy to use and allows seamless use of GPU power with little change to the standard code.

6 Acknowledgments

This research was funded by Cancer Research UK [C355/A26819] and FC AECC and AIRC under the Accelerator Award Programme, the project PIBA_2020_1_0055 (funded by the Basque Government) and the Synlethal Project (RETOS Investigacion, Spanish Government). Conflict of Interest: none declared.

7 Summary

To see more available functions, check the **GPUmatrix** package on CRAN.

References

J. Allaire and Y. Tang. tensorflow: R interface to 'tensorflow'. 2022. URL https://CRAN.R-project.org/package=tensorflow. R package version 2.11.0. [p1]

D. Bates, M. Maechler, and M. Jagan. Matrix: Sparse and dense matrix classes and methods. 2023. URL https://CRAN.R-project.org/package=Matrix. R package version 1.5-4. [p2]

H. Bengtsson. matrixstats: Functions that apply to rows and columns of matrices (and to vectors). 2022. URL https://CRAN.R-project.org/package=matrixStats. R package version 0.63.0. [p2]

V. Bonnici, F. Busato, S. Aldegheri, M. Akhmedov, L. Cascione, A. A. Carmena, F. Bertoni, N. Bombieri, I. Kwee, and R. Giugno. curnet: an r package for graph traversing on gpu. *BMC Bioinformatics*, 19:356, 2018. ISSN 1471-2105. doi: 10.1186/s12859-018-2310-3. URL https://doi.org/10.1186/s12859-018-2310-3. [p1]

J. Buckner, J. Wilson, M. Seligman, B. Athey, S. Watson, and F. Meng. The gputools package enables GPU computing in R. *Bioinformatics*, 26(1):134–135, 10 2009. ISSN 1367-4803. doi: 10.1093/bioinformatics/btp608. URL https://doi.org/10.1093/bioinformatics/btp608. [p1]

T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. pages 785–794, 2016. doi: 10.1145/2939672.2939785. URL http://doi.acm.org/10.1145/2939672.2939785. [p1]

R. S. de Souza, X. Quanfeng, S. Shen, C. Peng, and Z. Mu. qrpca: QR-based Principal Components Analysis. art. ascl:2208.002, Aug. 2022. [p1]

C. Determan. A short introduction to the gpur package. *Vignette*, 2017. [p1]

D. Falbel and J. Luraschi. *torch: Tensors and Neural Networks with 'GPU' Acceleration*, 2023. URL https://CRAN.R-project.org/package=torch. R package version 0.11.0. [p1]

D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, Oct 1999. ISSN 1476-4687. doi: 10.1038/44565. URL https://doi.org/10.1038/44565. [p10]

T. P. Minka. A comparison of numerical optimizers for logistic regression. 2003. URL https://tminka.github.io/papers/logreg/minka-logreg.pdf. [p10]

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2022. URL https://www.R-project.org/. [p1]

RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, PBC., Boston, MA, 2020. URL http://www.rstudio.com/. [p1]

D. Schmidt. *Introducing the float package: 32-Bit Floats for R*, 2017. URL https://cran.r-project.org/package=float. R Vignette. [p2]

J. Wang and M. Morgan. gpumagic: An opencl compiler with the capacity to compile r functions and run the code on gpu. 2022. R package version 1.14.0. [p1]

Z. Wang, T. Chu, L. A. Choate, and C. G. Danko. Rgtsvm: Support vector machines on a gpu in r, 2017. [p1]

César Lobato-Fernández
University of Navarra
Department of Biomedical Engineering and Sciences
Paseo Mikeletegui,48, 20009, San Sebastian, Gipuzkoa, Spain
ORCiD: 0000-0001-9576-7236
clobatofern@unav.es

Juan A. Ferrer-Bonsoms
University of Navarra
Department of Biomedical Engineering and Sciences
Paseo Mikeletegui,48, 20009, San Sebastian, Gipuzkoa, Spain
ORCiD: 0000-0002-1588-2195
jafhernandez@unav.es

Angel Rubio
University of Navarra
Department of Biomedical Engineering and Sciences
Paseo Mikeletegui,48, 20009, San Sebastian, Gipuzkoa, Spain
ORCiD: 0000-0002-3274-2450
arubio@unav.es