Package 'nmfgpu4R'

October 17, 2016

Type Package

Title Non-Negative Matrix Factorization (NMF) using CUDA

Version 0.2.5.2

Date 2016-10-17

Author Sven Koitka [aut, cre, cph],
Christoph M. Friedrich [ctb]

Maintainer Sven Koitka <sven.koitka@fh-dortmund.de>

Description Wrapper package for the nmfgpu library, which implements several Non-negative Matrix Factorization (NMF) algorithms for CUDA platforms. By using the acceleration of GPGPU computing, the NMF can be used for real-world problems inside the R environment. All CUDA devices starting with Kepler architecture are supported by the library.

License GPL-3 | file LICENSE

URL https://github.com/razorx89/nmfgpu4R

BugReports https://github.com/razorx89/nmfgpu4R/issues

Depends R (>= 3.1.0)

Imports Rcpp (>= 0.11.4), Matrix, SparseM, stats, stringr, tools, utils

Suggests gdata

LinkingTo Rcpp

RoxygenNote 5.0.1

SystemRequirements CUDA >= v7.0, Nvidia GPU (e.g. GeForce or Tesla) with compute capability >= 3.0 (Kepler)

NeedsCompilation yes

Repository CRAN

Date/Publication 2016-10-17 12:41:21

R topics documented:

`chooseDevice` *Selects the specified device as primary computation device. All further invocations to nmfgpu will use the specified CUDA device.*

Description

Selects the specified device as primary computation device. All further invocations to nmfgpu will use the specified CUDA device.

Usage

```
chooseDevice(deviceIndex)
```

Arguments

`deviceIndex`	Index of the CUDA device, which should be used for computation.

Note

CUDA enumerates devices starting with 0 for the first device.

`deviceCount` *Retrieves the total number of installed CUDA devices.*

Description

Retrieves the total number of installed CUDA devices.

Usage

```
deviceCount()
```

deviceMemoryInfo *Requests the currently available and total amount of device memory.*

Description

Requests the currently available and total amount of device memory.

Usage

```
deviceMemoryInfo(deviceIndex = NA)
```

Arguments

deviceIndex	If specified the memory info retrieval is restricted to the passed device indices. By default no restriction is active and therefore memory information about all available CUDA devices are retrieved.

Value

On success a list of lists will be returned, containing the following informations:

index	Index of the CUDA device
free.bytes	Amount of free memory in bytes.
total.bytes	Total amount of memory in bytes.

nmf *Non-negative Matrix Factorization (NMF) on GPU*

Description

Computes the non-negative matrix factorization of a data matrix X using the factorization parameter r. Multiple algorithms and initialization methods are implemented in the nmfgpu library using CUDA hardware acceleration. Depending on the available hardware, these algorithms should outperform traditional CPU implementations.

Usage

```
nmf(...)

## Default S3 method:
nmf(data, r, algorithm = "mu",
  initMethod = "AllRandomValues", seed = floor(runif(1, 0,
  .Machine$integer.max)), threshold = 0.1, maxiter = 2000, runs = 1,
  parameters = NULL, useSinglePrecision = F, verbose = T, ssnmf = F,
  ...)
```

```
## S3 method for class 'formula'
nmf(formula, data, ...)

## S3 method for class 'nmfgpu'
fitted(object, ...)

## S3 method for class 'nmfgpu'
predict(object, newdata, ...)
```

Arguments

`...`	Other arguments
`data`	Data matrix of dimension n x m with n attributes and m observations. Please note that this differs from most other data mining/machine learning algorithms!
`r`	Factorization parameter, which affects the quality of the approximation and runtime.
`algorithm`	Choosing the right algorithm depends on the data structure. Currently the following algorithms are implemented in the nmfgpu library:

- **mu**: Multiplicative update rules presented by Lee and Seung [2] use a purely multiplicative update and are a special form of the gradient descent algorithm (special step parameter). The implemented update rules make use of the frobenius norm and therefore are faster than the Kullback-Leibler ones.
- **gdcls**: Gradient Descent Constrained Least Squares (GDCLS) presented by Pauca et al [3,4] is a hybrid algorithm. It uses a least squares solver for updating matrix `H` and the multiplicative update rule for `W` as defined by Lee and Seung [2]. Additionaly the GDCLS algorithm uses the parameter `lambda` from the `parameters` argument to control the sparsity of the matrix `H`. As from the authors presented, the sparsity parameter `lambda` should be between `0.1` and `0.0001`, but at least positive.
- **als**: Alternating Least Squares (ALS) originally presented by Paatero and Tapper [1,4] uses a least squares solver for updating both matrices `W` and `H`.
- **acls**: Alternating Constrained Least Squares (ACLS) presented by Langville et al [4] enhances the normal ALS algorithm by introducing sparsity parameters. Both `lambdaW` and `lambdaH` must be provided in the `parameters` argument and must be in the range of `0` and positive infinity.
- **ahcls**: Alternating Hoyer Constrained Least Squares (AHCLS) presented by Langville et al [4] enhances the ACLS algorithm by introducing a second set of sparsity parameters. Additionaly to `lambdaW` and `lambdaH` the sparsity parameters `alphaH` and `alphaW` must be provided in the `parameters` argument. Both should be set in the range of `0.0` and `1.0`, representing a percentage sparsity for each matrix. As recommended by the authors all four parameters should be set to `0.5` as starting values.)
- **nsnmf**: Non-smooth Non-negative Matrix Factorization (nsNMF) presented by Pascual-Montano et al [6] is an enhancement to the multiplicative update

rules [2]. With an extra parameter `theta` the user has control over the influence of the model. The value should be in the range of `0.0` and `1.0` to work like intended.

`initMethod` All initialization methods depend on the selected algorithm. Using the fact that a least squares type algorithm computes the matrix `H` in the first step, does make an initialization for `H` unnecessary. Therefore only the initialization method for matrix `W` will be executed for any least squares type algorithm.

- **CopyExisting**: Initializes the factorization matrices `W` and `H` with existing values, which requires `W` and `H` to be set in the `parameters` argument. On the one hand this enables the user to chain different algorithms, for example using a fast converging algorithm for a base approximation and and a slow algorithm with better convergence properties to finish the optimization process. On the other hand the user can supply matrix intializations, which are not supported by this interface. *Note*: Both `W` and `H` must have the same dimension as they would have from the passed arguments `X` and `r`.
- **AllRandomValues**: Initializes the factorization matrices `W` and `H` with uniformly distributed values between `0.0` and `1.0`, where `0.0` is excluded and `1.0` is included.
- **MeanColumns**: Initializes the factorization matrix `W` by computing the mean of five random data matrix columns. The matrix `H` will be initialized as it would when using `AllRandomValues`.
- **k-Means/Random**: Initializes the factorization matrix `W` by computing the k-Means cluster centers of the data matrix. The matrix `H` will be initialized as it would when using `AllRandomValues`. This method was presented by Gong et al [5] as initialization strategy H2.
- **k-Means/NonNegativeWTV**: Initializes the factorization matrix `W` by computing the k-Means cluster centers of the data matrix. The matrix `H` will be initialized with the product `t(W) %*% V`, but all negative values are clamped to zero. This method was presented by Gong et al [5] as initialization strategy H4.
- **EIn-NMF**: Initializes the factorization matrix `W` by computing the k-Means cluster centers of the data matrix. The matrix `H` will be initialized with a prefix sum equation to build weighted encoding vectors. This method was presented by Gong et al [5] as initialization strategy H5.

`seed` The `seed` is used to initialize the random number generators for initializing the factorization matrices. Setting this argument to a fixed value ensures the same initialization of the matrices. This can be handy for benchmarking different algorithms with the same initialization.

`threshold` **First convergence criterion:** The `threshold` is used to determine if the algorithm has converged to a local minimum by checking the difference between the last frobenius norm and the current one. If it is below the `threshold`, then the algorithm is assumed to be converged.

`maxiter` **Second convergence criterion:** If the first convergence criterion is not reached, but a maximum number of iterations, the execution of the algorithm will be aborted.

`runs` Performs the specified amount of runs and stores the best factorization result.

`parameters` A list of additional parameters, which are required by some `algorithm` and `initMethod` values.

`useSinglePrecision`
By default R only knows about double precision numerical data types. If this parameter is set to true, then the algorithm will convert the double precision data to single precision for computation. The result will be converted back to double precision data.

`verbose` By default information about the factorization process and current status will be written to the console. For silent execution `verbose=T` may be passed, preventing any output besides error messages.

`ssnmf` Internal flag (Don't use it)

`formula` Formula object with no intercept and labels for selected attributes. Note that die labels are selected from the rows instead of the columns, because NMF expects rows to be attributes.

`object` Object of class "`nmfgpu`"

`newdata` New data matrix compatible to the training data matrix, for computing the corresponding mixing matrix.

Value

If the factorization process was successful, then a list of the following values will be returned otherwise NULL:

`W`	Factorized matrix W with n attributes and r basis features of the data matrix.
`H`	Factorized matrix H with r mixing vectors for m data entries in the data matrix.
`Frobenius`	Contains the frobenius norm of the factorization at the end of algorithm execution.
`RMSD`	Contains the root-mean-square deviation (RMSD) of the factorization at the end of algorithm execution.
`ElapsedTime`	Contains the elapsed time for initialization and algorithm execution.
`NumIterations`	Number of iterations until the algorithm had converged.

References

1. P. Paatero and U. Tapper, "Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values", Environmetrics, vol. 5, no. 2, pp. 111-126, 1994.
2. D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization", in Advances in Neural Information Processing Systems 13 (T. Leen, T. Dietterich, and V. Tresp, eds.), pp. 556-562, MIT Press, 2001.
3. V. P. Pauca, J. Piper, and R. J. Plemmons, "Nonnegative matrix factorization for spectral data analysis", Linear Algebra and its Applications, vol. 416, no. 1, pp. 29-47, 2006. Special Issue devoted to the Haifa 2005 conference on matrix theory.
4. A. N. Langville, C. D. Meyer, R. Albright, J. Cox, and D. Duling, "Algorithms, initializations, and convergence for the nonnegative matrix factorization", CoRR, vol. abs/1407.7299, 2014.
5. L. Gong and A. Nandi, "An enhanced initialization method for non-negative matrix factorization", in 2013 IEEE International Workshop on Machine Learning for Signal Processing (MLSP), pp. 1-6, Sept 2013.

6. A. Pascual-Montano, J. M. Carazo, K. Kochi, D. Lehmann and R. D. Pascual-Marqui "Nonsmooth nonnegative matrix factorization (nsNMF)", in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 28, pp. 403-415, 2006

Examples

```
## Not run:
# Initialize the library
library(nmfgpu4R)
nmfgpu4R.init()

# Create dummy data
data <- runif(256*1024)
dim(data) <- c(256, 1024)

# Compute several factorization models
result <- nmf(data, 128, algorithm="mu", initMethod="K-Means/Random", maxiter=500)
result <- nmf(data, 128, algorithm="mu", initMethod="CopyExisting",
                parameters=list(W=result$W, H=result$H), maxiter=500)
result <- nmf(data, 128, algorithm="gdcls", maxiter=500, parameters=list(lambda=0.1))
result <- nmf(data, 128, algorithm="als", maxiter=500)
result <- nmf(data, 128, algorithm="acls", maxiter=500,
                parameters=list(lambdaH=0.1, lambdaW=0.1))
result <- nmf(data, 128, algorithm="ahcls", maxiter=500,
                parameters=list(lambdaH=0.1, lambdaW=0.1, alphaH=0.5, alphaW=0.5))
result <- nmf(data, 128, algorithm="nsnmf", maxiter=500, parameters=list(theta=0.25))

# Compute encoding matrices for training and test data
set.seed(42)
idx <- sample(1:nrow(iris), 100, replace=F)
data.train <- iris[idx,]
data.test <- iris[-idx,]

model.nmf <- nmf(t(data.train[,-5]), 2)
encoding.train <- t(predict(model.nmf))
encoding.test <- t(predict(model.nmf, t(data.test[,-5])))

plot(encoding.train, col=data.train[,5], pch=1)
points(encoding.test, col=data.test[,5], pch=4)

## End(Not run)
```

`nmfgpu4R` *R binding for computing non-negative matrix factorizations using CUDA*

Description

R binding for the libary *nmfgpu* which can be used to compute Non-negative Matrix Factorizations (NMF) using CUDA hardware acceleration.

Details

The main function to use is nmf which can be configured using various arguments. In addition to it a few helper functions are provided, but they aren't necessary for using nmf.

`nmfgpu4R.init` *Initializes the C++ library nmfgpu, which provides the core functionality of this package.*

Description

Initializes the C++ library nmfgpu, which provides the core functionality of this package.

Usage

```
nmfgpu4R.init(quiet = F)
```

Arguments

`quiet`	If true then informative messages about the found CUDA version and nmfgpu location will be suppressed.

Details

As this package depends on a C++ library, there are some restrictions in terms of usage. First this package is only compatible with x64 environments, because some CUDA libraries are not available for x86 environments. Second you need a CUDA capable device starting with Kepler architecture, CUDA device drivers and the CUDA toolkit with version 7.0 or higher. Lastly you need the nmfgpu library itself. This package provides a basic service of downloading precompiled versions from github, if it is available for your operating system and CUDA toolkit version. Otherwise you need to compile and install the library on your own by following the instructions on https://github.com/razorx89/nmfgpu.

Even if the package downloads a precompiled version, it must not neccessarily be compatible with your system. For example on Windows platforms you must have installed the "Microsoft Visual C++ Redistributable Packages for Visual Studio 2013", which can be found at https://www.microsoft.com/en-us/download/details.aspx?id=40784. Furthermore on unix systems there could be a version mismatch with the `libstdc++.so` library, because the installed compiler and the compiler which was used to build the binary could be different.

If you encounter any problems loading the nmfgpu library, then try to compile it by yourself.

`print.DeviceMemoryInfo` *Prints the information of a 'DeviceMemoryInfo' object.*

Description

Prints the information of a 'DeviceMemoryInfo' object.

Usage

```
## S3 method for class 'DeviceMemoryInfo'
print(x, ...)
```

Arguments

`x`	Object of class 'DeviceMemoryInfo'
`...`	Other arguments

Index