Package 'gputools'

May 25, 2010

Version 0.21

Date 2010-05-24

Title A few GPU enabled functions

Author Josh Buckner <bucknerj@umich.edu>, Mark Seligman <mseligman@rapidbiologics.com>, Justin Wilson

Maintainer Josh Buckner <bucknerj@umich.edu>

Depends R (>= 2.8.0)

SystemRequirements Nvidia's CUDA toolkit (>= release 2.3)

Description This package provides R interfaces to a handful of common statistical algorithms. These algorithms are implemented in parallel using a mixture of Nvidia's CUDA langauge, Nvidia's CUBLAS library, and EMI Photonics' CULA libraries. On a computer equiped with an Nvidia GPU some of these functions may be substantially more efficient than native R routines.

License GPL-3

URL http://brainarray.mbni.med.umich.edu/Brainarray/Rgpgpu

Repository CRAN

Date/Publication 2010-05-25 17:08:47

R topics documented:

`chooseGpu` *Choose which GPU device to use*

Description

Selects the GPU device to use for computation. This is only useful on a machine equipped with multiple GPU devices. The numbering starts at 0 and is assigned by the CUDA capable driver.

Choosing a device can only be done before any other GPU operation and only once per thread.

Usage

```
chooseGpu(deviceId = 0)
```

Arguments

`deviceId`	an integer >= 0 designating the GPU to use for computation.

Value

chooseGpu should print out an integer specifying the device id chosen or an error message.

`getAucEstimate` *Estimate the AUC of the ROC*

Description

This function gives a quick estimate of the area under the curve (AUC) of the receiver operating characteristic (ROC). It is a quick way to estimate the quality of a binary classifier. The algorithm is based on a paper by David Hand and Robert Till (see references).

Usage

```
getAucEstimate(classes, scores)
```

Arguments

`classes`	a vector of floating point numbers. Each entry i corresponds to the real class of a point and should be either 0 or 1. The negative class is represented by 0 and the positive class by 1. These entries correspond both in number and order to the same points associated with the scores vector.
`scores`	a vector of floating point numbers. Each entry i corresponds to the probability that a point is in the positive class of a binary classification. This will be the output of, for example, a binary classifier based on logistic regression. These entries should correspond both in number and order to the same points associated with the classes vector.

Value

a single floating point number of double precision. This number represents an estimate of the auc score for the algorithm responsible for the scores vector. The estimation is according to the method of David Hand and Robert Till (see references).

References

Hand, David J. and Till, Robert J. (2001). A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine Learning*. 45, 171–186.

Examples

```
# generate some fake data
classes <- round(runif(20, min = 0, max = 1))

# fake probability that point i is in the positive class
scores <- runif(20, min = 0, max = 1)

b <- getAucEstimate(classes, scores)
print(b)
```

getGpuId *Discover the Id of the current GPU device*

Description

Queries the CUDA driver for the GPU device currently assigned to this thread. This is the id of the device that will be used for computation. If you wish to use a different device, use the chooseGpu function.

Usage

```
getGpuId()
```

Value

The function returns a single integer indicating the id of the GPU device currently selected to carry out computation according to the CUDA driver.

Examples

```
getGpuId()
```

gpuCor *Calculate Various Correlation Coefficients With a GPU*

Description

The correlation coefficient will be calculated for each pair x_i, y_j where x_i is a column of x and y_j is a column of y. Currently, Pearson's and Kendall's correlation coefficient are implemented. Pearson's may be calculated for data sets containing NAs in which case, the implementation behaves as R-native cor function with use="pairwise.complete".

Usage

```
gpuCor(x, y = NULL, use = "everything", method = "pearson")
```

Arguments

x	a matrix of floating point values in which each column is a random variable.
y	a matrix of floating point values in which each column is a random variable.
use	a string. A character string giving a method for computing in the presence of missing values. Options are "everything" or "pairwise.complete.obs". This currently only affects the "pearson" method.
method	a string. Either "pearson" or "kendall".

Value

For method "pearson", a list with matrices 'pairs', 'coefficients', and 'ts'. The matrix entry i, j for pairs represents the number of pairs of entries x_i^k, y_j^k (the k-th entry from x_i and y_j respectively). These are the number of entries actually used to calculate the coefficients. Entry i, j of the coefficients matrix is the correlation coefficient for x_i, y_j. Entry i, j of the ts matrix is the t-score of the i, j entry of the coefficient matrix. If use="pairwise.complete.obs" then only the pairs where both entries are not NA are used in the computations.

For method "kendall", a list of matrices 'pairs' as above and 'coefficients' as follows. The matrix 'coefficients' is a matrix of floating point numbers where entry i, j is the correlation coefficient for x_i, y_j. Calculation of t-scores for the kendall coefficients is not yet implemented.

See Also

cor

Examples

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

---

gpuDist *Compute Distances Between Vectors on a GPU*

Description

This function computes the distance between each vector of the 'points' argument using the metric specified by 'method'.

Usage

```
gpuDist(points, method = "euclidean", p = 2.0)
```

Arguments

points	a matrix of floating point numbers in which each row is a vector in R^n space where n is ncol(points).
method	a string representing the name of the metric to use to calculate the distance between the vectors of 'points'. Currently supported values are: "binary", "canberra", "euclidean", "manhattan", "maximum", and "minkowski".
p	a floating point parameter for the Minkowski metric.

Value

a class of type "dist" containing floating point numbers representing the distances between vectors from the 'points' argument.

See Also

dist

Examples

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

---

gpuDistClust *Compute Distances and Hierarchical Clustering for Vectors on a GPU*

Description

This function takes a set of vectors and performs clustering on them. The function will first calculate the distance between all of the pairs of vectors and then use the distances to cluster the vectors. Both of these steps are done on the GPU.

Usage

```
gpuDistClust(points, distmethod = "euclidean", clustmethod = "complete")
```

Arguments

points	a matrix of floating point numbers in which each row is a vector in R^n space where n is ncol(points).
distmethod	a string representing the name of the metric to use to calculate the distance between the vectors of 'points'. Currently supported values are: "binary", "canberra", "euclidean", "manhattan", "maximum".
clustmethod	a string representing the name of the clustering method to be applied to distances. Currently supported method names include "average", "centroid", "complete", "flexible", "flexible group", "mcquitty", "median", "single", "ward", and "wpgma".

Value

Copied from the native R function 'hclust' documentation. A class of type "hclust" with the following attributes.

merge	an n-1 by 2 matrix. Row i of 'merge' describes the merging of clusters at step i of the clustering. If an element j in the row is negative, then observation -j was merged at this stage. If j is positive then the merge was with the cluster formed at the (earlier) stage j of the algorithm. Thus negative entries in 'merge' indicate agglomerations of singletons, and positive entries indicate agglomerations of non-singletons. Copied from the native R function 'hclust' documentation.
order	a vector giving the permutation of the original observations suitable for plotting, in the sense that a cluster plot using this ordering and matrix 'merge' will not have crossings of the branches.
height	a set of n-1 non-decreasing real values. The clustering height: that is, the value of the criterion associated with the clustering 'method' for the particular agglomeration.

See Also

gpuDist, gpuHclust.

Examples

```
numVectors <- 5
dimension <- 10
Vectors <- matrix(runif(numVectors*dimension), numVectors, dimension)
myClust <- gpuDistClust(Vectors, "maximum", "mcquitty")
plot(myClust)
```

---

gpuFastICA *GPU enabled FastICA algorithm*

Description

This is an R and C code implementation of the FastICA algorithm of Aapo Hyvarinen et al. (http://www.cis.hut.fi/aapo/) to perform Independent Component Analysis (ICA) and Projection Pursuit.

Almost all of the code and documentation for this taken directly from the fastICA package. Only the function call to do svd on the GPU is due to the author of this package.

If the installer can't find cula, this function will be disabled.

Usage

```
gpuFastICA(X, n.comp, alg.typ = c("parallel","deflation"),
        fun = c("logcosh","exp"), alpha = 1.0,
        row.norm = FALSE, maxit = 200, tol = 1e-04, verbose = FALSE,
        w.init = NULL)
```

Arguments

X	a data matrix with n rows representing observations and p columns representing variables.
n.comp	number of components to be extracted
alg.typ	if alg.typ == "parallel" the components are extracted simultaneously (the default). if alg.typ == "deflation" the components are extracted one at a time.
fun	the functional form of the G function used in the approximation to neg-entropy (see details)
alpha	constant in range [1, 2] used in approximation to neg-entropy when fun == "logcosh"
row.norm	a logical value indicating whether rows of the data matrix X should be standardized beforehand.
maxit	maximum number of iterations to perform
tol	a positive scalar giving the tolerance at which the un-mixing matrix is considered to have converged.
verbose	a logical value indicating the level of output as the algorithm runs.
w.init	Initial un-mixing matrix of dimension (n.comp,n.comp). If NULL (default) then a matrix of normal r.v.'s is used.

Details

Independent Component Analysis (ICA)

The data matrix X is considered to be a linear combination of non-Gaussian (independent) components i.e. X = SA where columns of S contain the independent components and A is a linear mixing matrix. In short ICA attempts to 'un-mix' the data by estimating an un-mixing matrix W where XW = S.

Under this generative model the measured 'signals' in X will tend to be 'more Gaussian' than the source components (in S) due to the Central Limit Theorem. Thus, in order to extract the independent components/sources we search for an un-mixing matrix W that maximizes the non-gaussianity of the sources.

In FastICA, non-gaussianity is measured using approximations to neg-entropy (J) which are more robust than kurtosis based measures and fast to compute.

The approximation takes the form

$J(y) = [E\{G(y)\} - E\{G(v)\}]^2$ where v is a N(0,1) r.v.

The following choices of G are included as options $G(u) = \frac{1}{\alpha} \log \cosh(\alpha u)$ and $G(u) = -\exp(\frac{-u^2}{2})$

Algorithm

First, the data is centered by subtracting the mean of each column of the data matrix X.

The data matrix is then 'whitened' by projecting the data onto it's principle component directions i.e. X -> XK where K is a pre-whitening matrix. The number of components can be specified by the user.

The ICA algorithm then estimates a matrix W s.t XKW = S . W is chosen to maximize the neg-entropy approximation under the constraints that W is an orthonormal matrix. This constraint ensures that the estimated components are uncorrelated. The algorithm is based on a fixed-point iteration scheme for maximizing the neg-entropy.

Projection Pursuit

In the absence of a generative model for the data the algorithm can be used to find the projection pursuit directions. Projection pursuit is a technique for finding 'interesting' directions in multi-dimensional datasets. These projections and are useful for visualizing the dataset and in density estimation and regression. Interesting directions are those which show the least Gaussian distribution, which is what the FastICA algorithm does.

Value

A list containing the following components

X	pre-processed data matrix
K	pre-whitening matrix that projects data onto th first n.comp principal components.
W	estimated un-mixing matrix (see definition in details)
A	estimated mixing matrix
S	estimated source matrix

Author(s)

J L Marchini and C Heaton

References

A. Hyvarinen and E. Oja (2000) Independent Component Analysis: Algorithms and Applications, *Neural Networks*, **13(4-5)**:411-430

Examples

```
#--------------------------------------------------
#Example 1: un-mixing two mixed independent uniforms
#--------------------------------------------------

S <- matrix(runif(10000), 5000, 2)
A <- matrix(c(1, 1, -1, 3), 2, 2, byrow = TRUE)
X <- S%*%A

a <- gpuFastICA(X, 2, alg.typ = "parallel", fun = "logcosh", alpha = 1,
             row.norm = FALSE,
 maxit = 200, tol = 0.0001, verbose = TRUE)

par(mfrow = c(1, 3))
plot(a$X, main = "Pre-processed data")
plot(a$X%*%a$K, main = "PCA components")
plot(a$S, main = "ICA components")
```

```
#-------------------------------------------
#Example 2: un-mixing two independent signals
#-------------------------------------------

S <- cbind(sin((1:1000)/20), rep((((1:200)-100)/100), 5))
A <- matrix(c(0.291, 0.6557, -0.5439, 0.5572), 2, 2)
X <- S%*%A

a <- gpuFastICA(X, 2, alg.typ = "parallel", fun = "logcosh", alpha = 1,
             row.norm = FALSE,
 maxit = 200, tol = 0.0001, verbose = TRUE)

par(mfcol = c(2, 3))
plot(1:1000, S[,1 ], type = "l", main = "Original Signals",
     xlab = "", ylab = "")
plot(1:1000, S[,2 ], type = "l", xlab = "", ylab = "")
plot(1:1000, X[,1 ], type = "l", main = "Mixed Signals",
     xlab = "", ylab = "")
plot(1:1000, X[,2 ], type = "l", xlab = "", ylab = "")
plot(1:1000, a$S[,1 ], type = "l", main = "ICA source estimates",
     xlab = "", ylab = "")
plot(1:1000, a$S[, 2], type = "l", xlab = "", ylab = "")

#-----------------------------------------------------------
#Example 3: using FastICA to perform projection pursuit on a
#           mixture of bivariate normal distributions
#-----------------------------------------------------------

if(require(MASS)){
x <- mvrnorm(n = 1000, mu = c(0, 0), Sigma = matrix(c(10, 3, 3, 1), 2, 2))
x1 <- mvrnorm(n = 1000, mu = c(-1, 2), Sigma = matrix(c(10, 3, 3, 1), 2, 2))
X <- rbind(x, x1)

a <- gpuFastICA(X, 2, alg.typ = "deflation", fun = "logcosh", alpha = 1,
             row.norm = FALSE,
 maxit = 200, tol = 0.0001, verbose = TRUE)

par(mfrow = c(1, 3))
plot(a$X, main = "Pre-processed data")
plot(a$X%*%a$K, main = "PCA components")
plot(a$S, main = "ICA components")
}
```

`gpuGlm` *Fitting generalized linear models using GPU–enabled QR decomposition*

Description

Most of this documentation is copied from R's documentation for `glm`. `gpuGlm` is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

Note: The QR decomposition employed by `gpuLm` is optimized for speed and uses minimal pivoting.

Usage

```
gpuGlm(formula, family = gaussian, data, weights, subset,
na.action, start = NULL, etastart, mustart, offset, useSingle = TRUE,
control = gpuGlm.control(useSingle, ...), model = TRUE,
method = "gpuGlm.fit", x = FALSE, y = TRUE, contrasts = NULL, ...)

gpuGlm.fit(x, y, weights = rep(1, nobs), start = NULL, etastart = NULL,
mustart = NULL, offset = rep(0, nobs), family = gaussian(), useSingle,
control = gpuGlm.control(useSingle), intercept = TRUE)
```

Arguments

`formula`	an object of class "`formula`" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
`family`	a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. (See `family` for details of family functions.)
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `glm` is called.
`weights`	an optional vector of 'prior weights' to be used in the fitting process. Should be `NULL` or a numeric vector.
`subset`	an optional vector specifying a subset of observations to be used in the fitting process.
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.
`start`	starting values for the parameters in the linear predictor.
`etastart`	starting values for the linear predictor.
`mustart`	starting values for the vector of means.
`offset`	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be `NULL` or a numeric vector of

	length equal to the number of cases. One or more `offset` terms can be included in the formula instead or as well, and if more than one is specified their sum is used. See `model.offset`.
`useSingle`	whether to use single precision arithmetic on the gpu. Only the 'TRUE' option is implemented so far.
`control`	a list of parameters for controlling the fitting process. See the documentation for `glm.control` for details.
`model`	a logical value indicating whether *model frame* should be included as a component of the returned value.
`method`	the method to be used in fitting the model. The default method `"gpuGlm.fit"` uses iteratively reweighted least squares (IWLS). The only current alternative is `"model.frame"` which returns the model frame and does no fitting.
`x, y`	For `gpuGlm`: logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value. For `gpuGlm.fit`: `x` is a design matrix of dimension `n * p`, and `y` is a vector of observations of length `n`.
`contrasts`	an optional list. See the `contrasts.arg` of `model.matrix.default`.
`intercept`	logical. Should an intercept be included in the *null* model?
`...`	For `gpuGlm`: arguments to be passed by default to `glm.control`: see argument `control`. For `weights`: further arguments passed to or from other methods.

Details

A typical predictor has the form `response ~ terms` where `response` is the (numeric) response vector and `terms` is a series of terms which specifies a linear predictor for `response`. For `binomial` and `quasibinomial` families the response can also be specified as a `factor` (when the first level denotes failure and all others success) or as a two-column matrix with the columns giving the numbers of successes and failures. A terms specification of the form `first + second` indicates all the terms in `first` together with all the terms in `second` with any duplicates removed.

A specification of the form `first:second` indicates the the set of terms obtained by taking the interactions of all terms in `first` with all terms in `second`. The specification `first*second` indicates the *cross* of `first` and `second`. This is the same as `first + second + first:second`.

The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a `terms` object as the formula.

Non-`NULL` `weights` can be used to indicate that different observations have different dispersions (with the values in `weights` being inversely proportional to the dispersions); or equivalently, when the elements of `weights` are positive integers w_i, that each response y_i is the mean of w_i unit-weight observations. For a binomial GLM prior weights are used to give the number of trials when the response is the proportion of successes: they would rarely be used for a Poisson GLM.

`gpuGlm.fit` is the workhorse function: it is not normally called directly but can be more efficient where the response vector and design mattrix have already been calculated.

If more than one of etastart, start and mustart is specified, the first in the list will be used. It is often advisable to supply starting values for a quasi family, and also for families with unusual links such as gaussian("log").

All of weights, subset, offset, etastart and mustart are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

For the background to warning messages about 'fitted probabilities numerically 0 or 1 occurred' for binomial GLMs, see Venables & Ripley (2002, pp. 197–8).

Value

gpuGlm returns an object of class inheriting from "glm" which inherits from the class "lm". See later in this section.

The function summary (i.e., summary.glm) can be used to obtain or print a summary of the results and the function anova (i.e., anova.glm) to produce an analysis of variance table.

The generic accessor functions coefficients, effects, fitted.values and residuals can be used to extract various useful features of the value returned by glm.

weights extracts a vector of weights, one for each case in the fit (after subsetting and na.action).

An object of class "glm" is a list containing at least the following components:

coefficients a named vector of coefficients

residuals the *working* residuals, that is the residuals in the final iteration of the IWLS fit. Since cases with zero weights are omitted, their working residuals are NA.

fitted.values the fitted mean values, obtained by transforming the linear predictors by the inverse of the link function.

rank the numeric rank of the fitted linear model.

family the family object used.

linear.predictors the linear fit on link scale.

deviance up to a constant, minus twice the maximized log-likelihood. Where sensible, the constant is chosen so that a saturated model has deviance zero.

aic A version of Akaike's *An Information Criterion*, minus twice the maximized log-likelihood plus twice the number of parameters, computed by the aic component of the family. For binomial and Poison families the dispersion is fixed at one and the number of parameters is the number of coefficients. For gaussian, Gamma and inverse gaussian families the dispersion is estimated from the residual deviance, and the number of parameters is the number of coefficients plus one. For a gaussian family the MLE of the dispersion is used so this is a valid value of AIC, but for Gamma and inverse gaussian families it is not. For families fitted by quasi-likelihood the value is NA.

null.deviance The deviance for the null model, comparable with deviance. The null model will include the offset, and an intercept if there is one in the model. Note that this will be incorrect if the link function depends on the data other than through the fitted mean: specify a zero offset to force a correct calculation.

iter	the number of iterations of IWLS used.
weights	the *working* weights, that is the weights in the final iteration of the IWLS fit.
prior.weights	the weights initially supplied, a vector of 1s if none were.
df.residual	the residual degrees of freedom.
df.null	the residual degrees of freedom for the null model.
y	if requested (the default) the y vector used. (It is a vector even for a binomial model.)
x	if requested, the model matrix.
model	if requested (the default), the model frame.
converged	logical. Was the IWLS algorithm judged to have converged?
boundary	logical. Is the fitted value on the boundary of the attainable values?
call	the matched call.
formula	the formula supplied.
terms	the terms object used.
data	the data argument.
offset	the offset vector used.
control	the value of the control argument used.
method	the name of the fitter function used, currently always "gpuGlm.fit".
contrasts	(where relevant) the contrasts used.
xlevels	(where relevant) a record of the levels of the factors used in fitting.
na.action	(where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-empty fits will have components qr, R and effects relating to the final weighted linear fit.

Objects of class "glm" are normally of class c("glm", "lm"), that is inherit from class "lm", and well-designed methods for class "lm" will be applied to the weighted linear model at the final iteration of IWLS. However, care is needed, as extractor functions for class "glm" such as residuals and weights do **not** just pick out the component of the fit with the same name.

If a binomial glm model was specified by giving a two-column response, the weights returned by prior.weights are the total numbers of cases (factored by the supplied case weights) and the component y of the result is the proportion of successes.

Author(s)

The original R implementation of glm was written by Simon Davies working for Ross Ihaka at the University of Auckland, but has since been extensively re-written by members of the R Core team.

The design was inspired by the S function of the same name described in Hastie & Pregibon (1992).

This function was adapted for Nvidia's CUDA–supporting GPGPUs by Mark Seligman at Rapid Biologics LLC. http://www.rapidbiologics.com

References

Dobson, A. J. (1990) *An Introduction to Generalized Linear Models.* London: Chapman and Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models.* Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models.* London: Chapman and Hall.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S.* New York: Springer.

See Also

`anova.glm`, `summary.glm`, etc. for `glm` methods, and the generic functions `anova`, `summary`, `effects`, `fitted.values`, and `residuals`.

`lm` for non-generalized *linear* models (which SAS calls GLMs, for 'general' linear models).

`loglin` and `loglm` for fitting log-linear models (which binomial and Poisson GLMs are) to contingency tables.

`bigglm` in package **biglm** for an alternative way to fit GLMs to large datasets (especially those with many cases).

`esoph`, `infert` and `predict.glm` have examples of fitting binomial glms.

Examples

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- gpuGlm(counts ~ outcome + treatment, family=poisson())
anova(glm.D93)
summary(glm.D93)

## an example with offsets from Venables & Ripley (2002, p.189)
utils::data(anorexia, package="MASS")

anorex.1 <- gpuGlm(Postwt ~ Prewt + Treat + offset(Prewt),
                family = gaussian, data = anorexia)
summary(anorex.1)

# A Gamma example, from McCullagh & Nelder (1989, pp. 300-2)
clotting <- data.frame(
    u = c(5,10,15,20,30,40,60,80,100),
    lot1 = c(118,58,42,35,27,25,21,19,18),
    lot2 = c(69,35,26,21,18,16,13,12,12))
summary(gpuGlm(lot1 ~ log(u), data=clotting, family=Gamma))
summary(gpuGlm(lot2 ~ log(u), data=clotting, family=Gamma))
```

gpuGranger *Perform Granger Causality Tests for Vectors on a GPU*

Description

This function performs, with the aid of a GPU, Granger Causality Tests on permutations of pairs of columns of the input matrices 'x' and 'y'.

Usage

```
gpuGranger(x, y=NULL, lag)
```

Arguments

x	a matrix of floating point values. Each column represents a sequence of observations for a single random variable.
y	an optional matrix of floating point values. Each column represents a sequence of observations for a single random variable.
lag	a positive integer by which to offset the sequence of observations to calculate the coefficient for Granger causality.

Value

a list of two single precision floating point matrices both of the same dimension. The two matrices are fStatistics and pValues. The fStatistics matrix holds the F-statistics from the Granger causality tests. Each element of the pValues matrix is the p-value for the corresponding element of the fStatistics matrix.

If y is NULL, the test is run on permutations of pairs of columns of x. To find the Granger causality F-statistic estimating the answer to "Does variable x[,j] Granger-cause variable x[,i]?", look at fStatistics[i, j] and pValues[i, j].

If y is not NULL, the test is run on permutations of pairs (x[,i], y[,j]). To find the Granger causality F-statistic estimating the answer to "Does variable y[,j] Granger-cause variable x[,i]?", look at fStatistics[i, j] and pValues[i, j].

Examples

```
# permutations of pairs of cols of just x
numRandVars <- 5
numSamples <- 20
randVarSequences <- matrix(runif(numRandVars*numSamples), numSamples,
numRandVars)
gpuGranger(randVarSequences, lag = 5)

# pairs of cols, one from x and one from y
numXRandVars <- 5
numXSamples <- 20
x <- matrix(runif(numXRandVars*numXSamples), numXSamples, numXRandVars)
```

```
numYRandVars <- 3
numYSamples <- 20
y <- matrix(runif(numYRandVars*numYSamples), numYSamples, numYRandVars)

result <- gpuGranger(x, y, lag = 5)
print(result)
```

---

gpuHclust *Perform Hierarchical Clustering for Vectors with a GPU*

Description

This function performs clustering on a set of points. The distance between each pair of points should be calculated first using a function like 'gpuDist' or 'dist'.

Usage

```
gpuHclust(distances, method = "complete")
```

Arguments

distances	a class of type "dist" containing floating point numbers representing distances between points. R's native dist function and the gpuDist function produce output of this type.
method	a string representing the name of the clustering method to be applied to distances. Currently supported method names include "average", "centroid", "complete", "flexible", "flexible group", "mcquitty", "median", "single", "ward", and "wpgma".

Value

Copied from the native R function 'hclust' documentation. A class of type "hclust" with the following attributes.

merge	an n-1 by 2 matrix. Row i of 'merge' describes the merging of clusters at step i of the clustering. If an element j in the row is negative, then observation -j was merged at this stage. If j is positive then the merge was with the cluster formed at the (earlier) stage j of the algorithm. Thus negative entries in 'merge' indicate agglomerations of singletons, and positive entries indicate agglomerations of non-singletons. Copied from the native R function 'hclust' documentation.
order	a vector giving the permutation of the original observations suitable for plotting, in the sense that a cluster plot using this ordering and matrix 'merge' will not have crossings of the branches.
height	a set of n-1 non-decreasing real values. The clustering height: that is, the value of the criterion associated with the clustering 'method' for the particular agglomeration.

See Also

hclust, gpuDistClust

Examples

```
numVectors <- 5
dimension <- 10
Vectors <- matrix(runif(numVectors*dimension), numVectors, dimension)
distMat <- gpuDist(Vectors, "euclidean")
myClust <- gpuHclust(distMat, "single")
plot(myClust)
```

---

gpuLm — *Fitting Linear Models using a GPU–enabled QR*

Description

Most of this documentation is copied from R's documentation for lm. gpuLm is used to fit linear models using a GPU enabled QR decomposition. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Note: The QR decomposition employed by gpuLm is optimized for speed and uses minimal pivoting. If rank-revealing pivot is desired, then the function gpuQR, should be used. The most reliable determination of rank, however, will be obtained with the svd command.

Usage

```
gpuLm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, useSingle = TRUE, offset, ...)
```

Arguments

formula	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
data	an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. Should be NULL or a numeric vector. If non-NULL, weighted least squares is used with weights weights (that is, minimizing sum(w*e^2)); otherwise ordinary least squares is used. See also 'Details',

na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The 'factory-fresh' default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.
method	the method to be used; for fitting, currently only method = "qr" is supported; method = "model.frame" returns the model frame (the same as with model = TRUE, see below).
model, x, y, qr	logicals. If TRUE the corresponding components of the fit (the model frame, the model matrix, the response, the qr decomposition) are returned.
singular.ok	logical. If FALSE (the default in S but not in R) a singular fit is an error.
contrasts	an optional list. See the contrasts.arg of model.matrix.default.
useSingle	an optional logical. In the future, setting this to FALSE will result in using double precision arithmetic on the gpu, but this is not yet implemented
offset	this can be used to specify an *a priori* known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if more than one are specified their sum is used. See model.offset.
...	additional arguments to be passed to the low level regression fitting functions (see below).

Details

Models for lm are specified symbolically. A typical model has the form response ~ terms where response is the (numeric) response vector and terms is a series of terms which specifies a linear predictor for response. A terms specification of the form first + second indicates all the terms in first together with all the terms in second with duplicates removed. A specification of the form first:second indicates the set of terms obtained by taking the interactions of all terms in first with all terms in second. The specification first*second indicates the *cross* of first and second. This is the same as first + second + first:second.

If the formula includes an offset, this is evaluated and subtracted from the response.

If response is a matrix a linear model is fitted separately by least-squares to each column of the matrix.

See model.matrix for some further details. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a terms object as the formula (see aov and demo(glm.vr) for an example).

A formula has an implied intercept term. To remove this use either y ~ x - 1 or y ~ 0 + x. See formula for more details of allowed formulae.

Non-NULL weights can be used to indicate that different observations have different variances (with the values in weights being inversely proportional to the variances); or equivalently, when the elements of weights are positive integers w_i, that each response y_i is the mean of w_i unit-weight observations (including the case that there are w_i observations equal to y_i and the data have been summarized).

lm calls the lower level functions lm.fit, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of weights, subset and offset are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

Value

lm returns an object of class "lm" or for multiple responses of class c("mlm", "lm").

The functions summary and anova are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions coefficients, effects, fitted.values and residuals extract various useful features of the value returned by lm.

An object of class "lm" is a list containing at least the following components:

coefficients	a named vector of coefficients
residuals	the residuals, that is response minus fitted values.
fitted.values	the fitted mean values.
rank	the numeric rank of the fitted linear model.
weights	(only for weighted fits) the specified weights.
df.residual	the residual degrees of freedom.
call	the matched call.
terms	the terms object used.
contrasts	(only where relevant) the contrasts used.
xlevels	(only where relevant) a record of the levels of the factors used in fitting.
offset	the offset used (missing if none were used).
y	if requested, the response used.
x	if requested, the model matrix used.
model	if requested (the default), the model frame used.
na.action	(where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-null fits will have components assign, effects and (unless not requested) qr relating to the linear fit, for use by extractor functions such as summary and effects.

Using time series

Considerable care is needed when using lm with time series.

Unless na.action = NULL, the time series attributes are stripped from the variables before the regression is done. (This is necessary as omitting NAs would invalidate the time series attributes, and if NAs are omitted in the middle of the series the result would no longer be a regular time series.)

Even if the time series attributes are retained, they are not used to line up series, so that the time shift of a lagged or differenced regressor would be ignored. It is good practice to prepare a data argument by ts.intersect(..., dframe = TRUE), then apply a suitable na.action to that data frame and call gpuLm with na.action = NULL so that residuals and fitted values are time series.

Note

Offsets specified by `offset` will not be included in predictions by `predict.lm`, whereas those specified by an offset term in the formula will be.

Author(s)

The design was inspired by the S function of the same name described in Chambers (1992). The implementation of model formula by Ross Ihaka was based on Wilkinson & Rogers (1973).

This function was adapted for Nvidia's CUDA–supporting GPGPUs by Mark Seligman at Rapid Biologics LLC. `http://www.rapidbiologics.com`

References

Chambers, J. M. (1992) *Linear models.* Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic descriptions of factorial models for analysis of variance. *Applied Statistics*, **22**, 392–9.

See Also

`summary.lm` for summaries and `anova.lm` for the ANOVA table; `aov` for a different interface.

The generic functions `coef`, `effects`, `residuals`, `fitted`, `vcov`.

`predict.lm` (via `predict`) for prediction, including confidence and prediction intervals; `confint` for confidence intervals of *parameters*.

`lm.influence` for regression diagnostics, and `glm` for **generalized** linear models.

The underlying low level functions, `lm.fit` for plain, and `lm.wfit` for weighted regression fitting.

More `lm()` examples are available e.g., in `anscombe`, `attitude`, `freeny`, `LifeCycleSavings`, `longley`, `stackloss`, `swiss`.

`biglm` in package **biglm** for an alternative way to fit linear models to large datasets (especially those with many cases).

Examples

```
# require(graphics)

## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
anova(lm.D9 <- gpuLm(weight ~ group))
summary(lm.D90 <- gpuLm(weight ~ group - 1))# omitting intercept
summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
```

```
plot(lm.D9, las = 1) # Residuals, Fitted, ...
par(opar)

## model frame :
stopifnot(identical(gpuLm(weight ~ group, method = "model.frame"),
model.frame(lm.D9)))

### less simple examples in "See Also" above
```

`gpuLm.defaultTol` *Function to switch tolerance depending on precision*

Description

This function was written by Mark Seligman at Rapid Biologics, `http://rapidbiologics.com`

The function `gpuLm.fit` calls this function to determine a default tolerance. So gpuLm.defaultTol should *not* need to be used directly.

Usage

```
gpuLm.defaultTol(useSingle = TRUE)
```

Arguments

`useSingle`	logical. If TRUE, a tolerance will be returned appropriate for single precision arithmetic. If FALSE, a tolerance will be returned appropriate for double precision arithmetic.

Value

a floating point number representing a tolerance to be used by gpuLm.fit

See Also

`gpuLm.fit` `gpuLm`

`gpuLm.fit` *Fitter functions for gpu enabled linear models*

Description

The C code called by this function was written by Mark Seligman at Rapid Biologics, `http://rapidbiologics.com`

The function `gpuLm` calls this function to fit linear models. So gpuLm.fit should *not* need to be used directly.

Usage

```
gpuLm.fit(x, y, w = NULL, offset = NULL, method = "qr",
useSingle, tol = gpuLm.defaultTol(useSingle), singular.ok = TRUE, ...)
```

Arguments

x	design matrix of dimension n * p.
y	vector of observations of length n, or a matrix with n rows.
w	vector of weights (length n) to be used in the fitting process for the wfit functions. Weighted least squares is used with weights w, i.e., sum(w * e^2) is minimized.
offset	numeric of length n). This can be used to specify an *a priori* known component to be included in the linear predictor during fitting.
method	currently, only method="qr" is supported.
useSingle	logical. If TRUE, the gpu will use single precision arithmetic. In the future, if FALSE the gpu may use double precision arithmetic, but this is not implemented yet.
tol	tolerance for the qr decomposition. Default is 1e-7.
singular.ok	logical. If FALSE, a singular model is an error.
...	currently disregarded.

Value

a list with components

coefficients	p vector
residuals	n vector or matrix
fitted.values	n vector or matrix
effects	(not null fits)n vector of orthogonal single-df effects. The first rank of them correspond to non-aliased coefficients, and are named accordingly.
weights	n vector — *only* for the *wfit* functions.
rank	integer, giving the rank
df.residual	degrees of freedom of residuals
qr	(not null fits) the QR decomposition, see qr.

See Also

gpuLm which should usually be used for linear least squares regression

Examples

```
require(utils)
set.seed(129)
n <- 7 ; p <- 2
X <- matrix(rnorm(n * p), n,p) # no intercept!
y <- rnorm(n)
w <- rnorm(n)^2

str(lmw <- gpuLm.fit(x=X, y=y, w=w))
```

`gpuLsfit` *Least squares fit using GPU–enabled QR decomposition*

Description

The least squares estimate of β in the model

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

is found.

Most of this documentation is copied from R's documentation for `lsfit`. The function `gpuLsfit` performs a least–squares fit using a GPU enabled QR decomposition.

Note: The QR decomposition employed by `gpuLm` is optimized for speed and uses minimal pivoting. If more precise pivoting is desired, then either the function `gpuQR` or, better still, `svd` should be used.

Usage

```
gpuLsfit(x, y, wt=NULL, intercept=TRUE, useSingle = TRUE, tolerance=gpuLm.defaultTo
```

Arguments

`x`	a matrix whose rows correspond to cases and whose columns correspond to variables.
`y`	the responses, possibly a matrix if you want to fit multiple left hand sides.
`wt`	an optional vector of weights for performing weighted least squares.
`intercept`	whether or not an intercept term should be used.
`useSingle`	whether to use single precision arithmetic on the gpu. Only the 'TRUE' option is implemented so far.
`tolerance`	the tolerance to be used in the matrix decomposition. This defaults to 1e-04 for single–precision GPU computation.
`yname`	names to be used for the response variables.

Details

If weights are specified then a weighted least squares is performed with the weight given to the *j*th case specified by the *j*th entry in `wt`.

If any observation has a missing value in any field, that observation is removed before the analysis is carried out. This can be quite inefficient if there is a lot of missing data.

The implementation is via a modification of the LINPACK subroutines which allow for multiple left-hand sides.

Value

A list with the following named components:

`coef`	the least squares estimates of the coefficients in the model (β as stated above).
`residuals`	residuals from the fit.
`intercept`	indicates whether an intercept was fitted.
`qr`	the QR decomposition of the design matrix.

Author(s)

This function was adapted for Nvidia's CUDA–supporting GPGPUs by Mark Seligman at Rapid Biologics LLC. `http://www.rapidbiologics.com`

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`lsfit`, `lm`, `ls.print`, `ls.diag`

Examples

```
##-- Using the same data as the lm(.) example:
lsD9 <- gpuLsfit(x = unclass(gl(2,10)), y = weight)
ls.print(lsD9)
```

gpuMatMult *Perform Matrix Multiplication with a GPU*

Description

Performs matrix multiplication using a GPU. This function is merely a couple of wrappers for the CUBLAS cublasSgemm function.

Usage

```
gpuMatMult(a, b)
```

Arguments

a	a matrix of floating point values.
b	a matrix of floating point values.

Value

A matrix of single precision floating point values. The matrix is just the product of arguments 'a' and 'b'.

Examples

```
matA <- matrix(runif(2*3), 2, 3)
matB <- matrix(runif(3*4), 3, 4)
gpuMatMult(matA, matB)
```

gpuMi *B spline based mutual information*

Description

This function estimates the mutual information for permutations of pairs of columns of a matrix using a B spline approach on a GPU device. Please note, the data must be values from the interval [0.0, 1.0].

Usage

```
gpuMi(x, y = NULL, bins = 2, splineOrder = 1)
```

Arguments

x	a matrix of floating point numbers from the interval [0.0, 1.0]. Each column represents a list of samples of a random variable. The mutual information between each column of x and each column of y will be computed. If y is NULL then each pair of columns of x will be compared.
y	a matrix of floating point numbers from the interval [0.0, 1.0]. Each column represents a list of samples of a random variable. The mutual information between each column of x and each column of y will be computed. If y is NULL then each pair of columns of x will be compared.
bins	a single integer value representing the number of equal intervals that [0.0, 1.0] will be divided into in order to determine the bins in which to place each value of the columns of x and y. In the case of splineOrder = 1, this determines the histogram for traditional mutual information. For splineOrder > 1, a single value may be placed in multiple adjoining bins with varying weights on membership.
splineOrder	a single integer value giving the degree of the spline polynomials used to define both the number of bins a single value will be placed in and the weight of membership given to the value.

Value

a matrix of single precision floating point values of order ncol(y) by ncol(x). Entry (i, j) of this matrix represents the mutual information calculation for (y_i, x_j).

References

Carten O. Daub, Ralf Steuer, Joachim Selbig, and Sebastian Kloska. 2004. Estimating mutual information using B-spline functions – an improved similarity measure for analysing gene expression data. *BMC Bioinformatics*. 5:118. Available from http://www.biomedcentral.com/1471-2105/5/118

Examples

```
# get 3 random variables each with 20 samples
x <- matrix(runif(60), 20, 3)
y <- matrix(runif(60), 20, 3)
# do something interesting
y[,2] <- 3.0 * (x[,1] + x[,3])
z <- gpuMi(x, y, bins = 10, splineOrder = 3)
print(z)
```

---

gpuQr *Estimate the QR decomposition for a matrix*

Description

gpuQR estimates the QR decomposition for a matrix using column pivoting and householder matrices. The work is done on a GPU.

Note: a rank-revealing pivoting scheme is employed, potentially resulting in pivot distinctly different from ordinary "qr".

Usage

```
gpuQr(x, tol = 1e-07)
```

Arguments

x	a matrix of floating point numbers. This is the matrix that will be decomposed into Q and R factors.
tol	a floating point value. It is used for estimating the rank of matrix x.

Value

an object of class 'qr'. This object has members qr, qraux, pivot, rank. It is meant to be identical to the output of R's base function 'qr'. From the documentation for R's 'qr' function: The attribute qr is a matrix with the same dimension as 'x'. The upper triangle contains the R of the QR decomposition. The lower triangle contains partial information to construct Q. The attribute qraux is a vector of length 'ncol(x)' contains more information to construct Q. The attribute rank is a single integer representing an estimation of the rank of input matrix x based on the results of the QR decomposition. In some cases, this rank can be wildly different from the actual rank of the matrix x and so is only an estimation. The attribute pivot contains the permutation applied to columns of x in the process of calculating the QR decomposition.

Author(s)

The low–level implementation of this function for Nvidia's CUDA–supporting GPGPUs was written by Mark Seligman at Rapid Biologics LLC. http://www.rapidbiologics.com

References

Bischof, C. B. and Van Loan, C. F. (1987) The WY Representation for Products of Householder Matrices *SIAM J Sci. and Stat. Comp*, **8**, s2–s13.

Bjorck, Ake (1996) *Numerical methods for least squares problems*. SIAM.

Golub, Gene H. and Van Loan, C. F. (1996) *Matrix Computations*, **Ed. 3**, ch. 5.

Examples

```
# get some random data of any shape at all
x <- matrix(runif(25), 5, 5)
qr <- gpuQr(x)
print(qr)
```

gpuSolve — *Estimate the solution to a matrix vector equation*

Description

This function estimates the solution to an equation of the form x * b = y where x is a matrix, b is an unknown vector, and y is a known vector. It does much calculation on a GPU. If the y argument is omitted, the function returns the inverse of x.

The function uses R's base 'qr' and then applies the gpu to the result to get the final solution.

Usage

```
gpuSolve(x, y=NULL)
```

Arguments

x	a matrix of floating point numbers.
y	a vector of floating point numbers of length nrow(x).

Value

a vector or matrix of floating point numbers. If y is not null, then the value is an estimate of the vector b of length ncol(x) where x * b = y. If y is null or omitted, the value is a matrix, an estimate of a matrix multiplicative pseudo inverse of x.

Examples

```
x <- matrix(runif(100), 10, 10)
y <- runif(10)
b <- gpuSolve(x, y)
cat("Solution:\n")
print(b)
x.inverse <- gpuSolve(x)
cat("an estimate of a pseudo inverse for x:\n")
print(x.inverse)
```

gpuSvd — *Singular Value Decomposition of a Matrix with a GPU*

Description

Compute the singular-value decomposition of a rectangular matrix using the Cula library to compute the decomposition using a GPU.

Usage

```
gpuSvd(x, nu = min(n, p), nv = min(n, p))
```

Arguments

x	a real matrix whose SVD decomposition is to be computed.
nu	the number of left singular vectors to be computed. This must between 0 and n = nrow(x).
nv	the number of right singular vectors to be computed. This must be between 0 and p = ncol(x).

Details

The computation will be more efficient if nu <= min(n, p) and nv <= min(n, p), and even more efficient if one or both are zero.

Value

The SVD decomposition of the matrix,

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}',$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal, $\boldsymbol{V}'$ means *V transposed*, and $\boldsymbol{D}$ is a diagonal matrix with the singular values D_{ii}. Equivalently, $\boldsymbol{D} = \boldsymbol{U}'\boldsymbol{X}\boldsymbol{V}$, which is verified in the examples, below.

The returned value is a list with components

d	a vector containing the singular values of x, of length min(n, p).
u	a matrix whose columns contain the left singular vectors of x, present if nu > 0. Dimension c(n, nu).
v	a matrix whose columns contain the right singular vectors of x, present if nv > 0. Dimension c(p, nv).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide.* Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.
Available on-line at http://www.netlib.org/lapack/lug/lapack_lug.html.

See Also

eigen, qr.

Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
X <- hilbert(9)[,1:6]
(s <- gpuSvd(X))
D <- diag(s$d)
s$u %*% D %*% t(s$v) #  X = U D V'
t(s$u) %*% X %*% s$v #  D = U' X V
```

---

gpuSvmPredict *A support vector machine style binary classifier*

Description

This function classifies points in a data set with a support vector machine using a GPU. The negative category is represented by -1.f and the positive one by 1.f. The underlying code is adapted from Austin Carpenter's cuSVM which can be found at http://patternsonascreen.net/cuSVM.html

Usage

```
gpuSvmPredict(data, supportVectors, svCoefficients, svOffset,
kernelWidth = 0.125, isRegression = FALSE)
```

Arguments

data	a matrix of floating point numbers. Each row will be placed into one of two categories -1.f or 1.f. Note that ncol(data) should equal ncol(supportVectors).
supportVectors	a matrix of floating point numbers. Each row of the matrix is a support vector. This matrix can be obtained from gpuSvmTrain, for example. Note that ncol(supportVector) should equal ncol(data).
svCoefficients	a vector of floating point numbers representing coefficients corresponding to the support vectors. This vector can be obtained from gpuSvmTrain, for example. Each support vector supportVectors[i,] has coefficient svCoefficients[i].
svOffset	a single floating point number. It is the offset for the prediction function. The offset can be obtained from gpuSvmTrain, for example.
kernelWidth	a single floating point number. This is the scalar Gaussian kernel parameter.
isRegression	a single logical value indicating if the supportVectors result from regression.

Value

a vector of nrow(data) entries, each either -1.f or 1.f. Each entry i corresponds to the support vector machine's prediction for the category of data[i,].

References

Carpenter, Austin, *cuSVM: a cuda implementation of support vector classification and regression*, http://http://patternsonascreen.net/cuSVM.html

Examples

```
# y is discrete: -1 or 1 and we set isRegression to FALSE
y <- round(runif(100, min = 0, max = 1))
for(i in 1:5) { if(y[i] == 0) {y[i] <- -1}}

x <- matrix(runif(500), 100, 5)

a <- gpuSvmTrain(y, x, isRegression = FALSE)
print(a)

b <- gpuSvmPredict(x, a$supportVectors, a$svCoefficients, a$svOffset,
    isRegression = FALSE)
print(b)

# this time around, y : -1 or 1 and we set isRegression to FALSE
y <- runif(100, min = -1, max = 1)

x <- matrix(runif(500), 100, 5)

a <- gpuSvmTrain(y, x, isRegression = TRUE)
print(a)

b <- gpuSvmPredict(x, a$supportVectors, a$svCoefficients, a$svOffset,
    isRegression = TRUE)
print(b)
```

---

gpuSvmTrain *Train a support vector machine on a data set*

Description

This function trains, with the aid of a GPU, a support vector machine using the input data x separated into classes y. The function is capable of both regression (the entries of y are continuous) and non-regression (each entry of y is either -1.f or 1.f). The underlying code is adapted from Austin Carpenter's cuSVM which can be found at http://patternsonascreen.net/cuSVM.html

Usage

```
gpuSvmTrain(y, x, C = 10, kernelWidth = 0.125, eps = 0.5,
    stoppingCrit = 0.001, isRegression = FALSE)
```

Arguments

y	a vector of floating point numbers. The length of y should equal the number of rows of x. In the case of isRegression = FALSE, each entry of y is the category of the row of data x. The negative category is indicated by -1 and the positive category is indicated by 1. In the case of isRegression = TRUE, the values of y may take any value between -1 and 1 inclusive. These categories are used to train the svm.
x	a matrix of floating point numbers. Each row i is a point with a category given by y[i]. This is the data set used for training the svm.
C	a single floating point number. This is the SVM regularization parameter.
kernelWidth	a single floating point number. This is the scalar Gaussian kernel parameter.
eps	a single floating point number. This is the epsilon used in regression mode.
stoppingCrit	a single floating point number. This is the optimization stopping criterion.
isRegression	a single logical value. If isRegression is set to TRUE then regression is performed and the y value may be continuously valued. If not, then we use normal svm training and each value in y must be either -1 or 1.

Value

a list consisting of the following elements: supportVectors, svCoefficients, and svOffset. The element supportVectors is a matrix of single precision floating point numbers. These are the support vectors corresponding to the coefficients in svCoefficients. Row i of supportVectors contains ncol(x) columns and has coefficient svCoefficients[i]. The element svCoefficients is a single precision vector of the support vector coefficients. The element svOffset is a single floating point number of single precision. It is the offset for the prediction function.

References

Carpenter, Austin, *cuSVM: a cuda implementation of support vector classification and regression*, http://http://patternsonascreen.net/cuSVM.html

Examples

```
# y is discrete: -1 or 1 and we set isRegression to FALSE
y <- round(runif(100, min = 0, max = 1))
for(i in 1:5) { if(y[i] == 0) {y[i] <- -1}}

x <- matrix(runif(500), 100, 5)

a <- gpuSvmTrain(y, x, isRegression = FALSE)
print(a)

b <- gpuSvmPredict(x, a$supportVectors, a$svCoefficients, a$svOffset,
    isRegression = FALSE)
print(b)

# this time around, y : -1 or 1 and we set isRegression to FALSE
y <- runif(100, min = -1, max = 1)
```

```
x <- matrix(runif(500), 100, 5)

a <- gpuSvmTrain(y, x, isRegression = TRUE)
print(a)

b <- gpuSvmPredict(x, a$supportVectors, a$svCoefficients, a$svOffset,
    isRegression = TRUE)
print(b)
```

`gpuTtest` *T-Test Estimator with a GPU*

Description

Given the number of samples and a Pearson correlation coefficient, this function estimates the t-score on a GPU. If an entry in goodPairs is zero or one then you may get a NaN as the t-test result.

Usage

```
gpuTtest(goodPairs, coeffs)
```

Arguments

`goodPairs`	a vector of positive integer values. Value i represents the number of samples used to calculate the i-th value of the 'coeffs' argument.
`coeffs`	a vector of floating point values representing Pearson correlation coefficients.

Value

a vector of single precision floating point values. The i-th entry is an estimate of the t-score of the i-th entry of the 'coeffs' argument.

See Also

gpuCor.

Examples

```
goodPairs <- rpois(10, lambda=5)
coeffs <- runif(10)
gpuTtest(goodPairs, coeffs)
```

Index