Package: gputools (via r-universe)

October 24, 2026

Version 2.1.0

Title GPU Drop-Ins for Statistics on Wide Data

Depends R (>= 4.0.0)

Imports utils

Suggests testthat (>= 3.0.0), knitr, rmarkdown

Config/testthat/edition 3

VignetteBuilder knitr

SystemRequirements NVIDIA CUDA toolkit (>= 11.7); an NVIDIA GPU to run the GPU functions

Description GPU versions of base R statistics for data with many variables, using the NVIDIA 'CUDA' toolkit: correlation matrices (Pearson, Spearman and Kendall) and their tests, the strongest correlated pairs among tens of thousands of variables without forming the matrix, association scans with covariates as in expression quantitative trait loci studies (Shabalin 2012 <doi:10.1093/bioinformatics/bts163>), permutation tests with Westfall-Young maxT adjustment, distance matrices, hierarchical and k-means clustering, Granger causality tests, B-spline mutual information, principal components, and least-squares fits. Results agree with the base R functions they replace. Without 'CUDA' the package installs, and its functions explain how to get a 'CUDA' build. Described in Buckner et al. (2010) <doi:10.1093/bioinformatics/btp608>.

License GPL-3

URL https://github.com/nullsatz/gputools

BugReports https://github.com/nullsatz/gputools/issues

NeedsCompilation yes

Author Josh Buckner [aut, cre], Mark Seligman [aut], Fan Meng [aut], Justin Wilson [ctb]

Maintainer Josh Buckner <nullsatz@gmail.com>

Config/pak/sysreqs nvidia-cuda-dev

Repository https://cran.r-universe.dev

Date/Publication 2026-10-24 16:40:02 UTC

RemoteUrl https://github.com/cran/gputools

RemoteRef HEAD

RemoteSha d4bdc5f4486243bc401572ae31014a73415d09fe

Contents

chooseGpu . . . 3
cpuMatMult . . . 3
getGpuId . . . 4
gpuAssoc . . . 4
gpuAvailable . . . 6
gpuCor . . . 7
gpuCorPairs . . . 8
gpuCorPerm . . . 10
gpuCrossprod . . . 12
gpuDist . . . 12
gpuDistClust . . . 13
gpuGlm . . . 15
gpuGranger . . . 20
gpuHclust . . . 21
gpuKmeans . . . 23
gpuLm . . . 24
gpuLm.defaultTol . . . 28
gpuLm.fit . . . 29
gpuLsfit . . . 30
gpuMatMult . . . 32
gpuMi . . . 32
gpuPrcomp . . . 34
gpuQr . . . 35
gpuRowTtest . . . 36
gpuSolve . . . 38
gpuSvd . . . 39
gpuTcrossprod . . . 40
gpuTtest . . . 40

Index **42**

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.

cpuMatMult — *Perform Matrix Multiplication*

Description

Performs matrix multiplication using R's BLAS. This function is merely a wrapper for the BLAS dgemm function.

Usage

```
cpuMatMult(a, b)
```

Arguments

a	a numeric matrix.
b	a numeric matrix.

Value

A numeric matrix. 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)
cpuMatMult(matA, matB)
```

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()
```

gpuAssoc — *Association Scan: Every Column of x Against Every Column of y, With a GPU*

Description

Tests every column of `x` against every column of `y` by linear regression, `lm(y[, j] ~ x[, i] + covariates)`, and returns the significant associations. It is the scan of expression quantitative trait loci (eQTL) studies, genotypes against gene expression, and of other many-by-many association screens; it computes what Matrix eQTL's linear model does (Shabalin 2012), on the GPU.

Usage

```
gpuAssoc(x, y, covariates = NULL, pvalue = 1e-5, top = NULL,
  limit = 1e7, tile = 2048L,
  precision = getOption("gputools.precision", "single"))
```

Arguments

`x`	a numeric matrix, a column per tested variable (a SNP's genotypes, say, coded 0, 1, 2), a row per sample.
`y`	a numeric matrix, a column per outcome (a gene's expression), the same rows.
`covariates`	`NULL` or a numeric matrix of covariates to adjust for (sex, age, principal components of ancestry, ...), the same rows. An intercept is always included.
`pvalue`	report every association with a p-value at most this.
`top`	or instead, report each column of `y`'s `top` most significant columns of `x`; `pvalue` is then not used.
`limit`	the most associations to return: a `pvalue` that lets more through is an error, not a result of unexpected size.
`tile`	the block size the GPU works in; see `gpuCorPairs`. It has no effect on the result.
`precision`	`"single"` or `"double"`: the floating point precision the GPU computes in; see `gpuCor`.

Details

For each pair, the slope of `x[, i]` in the least-squares fit of `y[, j]` on it, an intercept and the covariates is tested for zero with the t test `summary(lm(...))` reports, on $n - 2 - k$ degrees of freedom for n samples and k covariates. As in Matrix eQTL, the intercept and covariates are projected out of every column of `x` and `y` once, after which each test is a correlation: the scan is one large matrix product, computed a block at a time and filtered on the GPU.

A column of `x` or `y` that the covariates explain completely, or a constant one, has no test and is never reported. Missing values are not allowed: impute them first (Matrix eQTL uses the SNP's mean genotype) or drop the sample.

The whole of `x` and `y` is copied to the GPU. For more SNPs than fit, call `gpuAssoc` on a chromosome or a block of SNPs at a time: the results are the same apart from the false discovery rates, which then count only that call's tests.

Value

A data frame, a row per reported association:

`i`	the column of `x`.
`j`	the column of `y`.
`x`, `y`	the columns' names, if they have them.
`beta`	the slope of `x[, i]`.
`t`	its t statistic.
`p.value`	its two-sided p-value.
`fdr`	the Benjamini-Hochberg false discovery rate over all `ncol(x) * ncol(y)` tests. It is computed from the reported p-values alone, as Matrix eQTL computes it, and so can overstate the rate but never understate it.

With `pvalue`, rows are ordered from the most significant; with `top`, by `j` and then from the most significant.

References

Shabalin, A. A. (2012) Matrix eQTL: ultra fast eQTL analysis via large matrix operations. *Bioinformatics* **28**, 1353–1358.

See Also

lm, gpuCorPairs

Examples

```
set.seed(1)
samples <- 200
snps <- matrix(rbinom(samples * 1000, 2, 0.3), samples, 1000,
               dimnames = list(NULL, paste0("rs", 1:1000)))
age <- rnorm(samples, 50, 10)
genes <- matrix(rnorm(samples * 300), samples, 300,
                dimnames = list(NULL, paste0("gene", 1:300)))
genes[, 7] <- genes[, 7] + 0.6 * snps[, 42] + 0.05 * age   # one real eQTL

gpuAssoc(snps, genes, covariates = age, pvalue = 1e-6)

# the same test by lm()
summary(lm(genes[, 7] ~ snps[, 42] + age))$coefficients[2, ]
```

---

gpuAvailable *Is a CUDA Device Usable?*

Description

Reports whether gputools can use a CUDA device: the package was built with CUDA, the NVIDIA driver is installed and at least one GPU is visible. The GPU functions stop with an error when it is FALSE.

Usage

```
gpuAvailable()
```

Details

A gputools installed where configure could not find the CUDA toolkit (for example a binary built on a server without CUDA) loads, but its GPU functions cannot run; it says so when attached. Install the NVIDIA CUDA toolkit (11.7 or later), make it findable – set CUDA_HOME to its directory, or put nvcc on the PATH – and reinstall gputools from source.

Value

TRUE, or FALSE with a "reason" attribute saying why no device can be used.

See Also

chooseGpu, getGpuId

Examples

```
if (gpuAvailable()) getGpuId()
```

---

gpuCor *Calculate Various Correlation Coefficients With a GPU*

Description

Correlates each column of x with each column of y, as cor does, with Pearson's, Kendall's (tau-b) or Spearman's coefficient, and tests each correlation for zero, as cor.test does.

Usage

```
gpuCor(x, y = NULL, use = "everything", method = "pearson",
  test = FALSE, precision = getOption("gputools.precision", "single"))
```

Arguments

x	a numeric matrix whose columns are the variables.
y	a numeric matrix with the same number of rows, or NULL for x with itself.
use	"everything" (a variable with a missing value gives NA) or "pairwise.complete.obs" (each pair of variables uses the rows where both are present), as with cor.
method	"pearson", "kendall" or "spearman".
test	TRUE to also return p-values (and Kendall's z), as cor.test would. They are computed on the CPU, at about 0.3 microseconds each: negligible for hundreds of variables, but for thousands (millions of pairs) they take longer than the correlations.
precision	"single" or "double": the floating point precision the GPU computes in. Single precision is the default, set with options(gputools.precision = "double"); it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

Details

Kendall's coefficient is tau-b and Spearman's is Pearson's on the ranks, ties averaged; with use = "pairwise.complete.obs", each pair is ranked over its complete rows only. All three match cor.

The p-values are two-sided and match cor.test: exactly for Pearson's (a t test on pairs - 2 degrees of freedom), and as cor.test(exact = FALSE) gives them for Spearman's (the same t approximation) and Kendall's (a normal approximation with the tie correction). cor.test's exact small-sample tests for the rank coefficients are not computed.

Value

A list of matrices, each with a row per column of `x` and a column per column of `y`, named as `cor` names them:

`coefficients`	the correlations.
`ts`	for Pearson's and Spearman's, the t statistics, $r\sqrt{(n-2)/(1-r^2)}$.
`z`	for Kendall's, with `test = TRUE`, the normal statistics.
`p.value`	with `test = TRUE`, the two-sided p-values.
`pairs`	the number of rows each correlation used.

See Also

`cor`, `cor.test`

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")
gpuCor(A, B, method="spearman", test=TRUE)$p.value
A[3,2] <- NA
gpuCor(A, B, use="pairwise.complete.obs", method="pearson")
```

`gpuCorPairs` *The Strongest Correlations Among Many Variables, With a GPU*

Description

Finds the strongly correlated pairs among the columns of `x` (or between the columns of `x` and those of `y`) without forming the whole correlation matrix: every pair whose correlation reaches a threshold, or each variable's strongest partners. It is for data too wide for `gpuCor` or `cor`: tens of thousands of genes, whose correlation matrix would take gigabytes, reduced to the edge list of a co-expression network.

Usage

```
gpuCorPairs(x, y = NULL, threshold = NULL, top = NULL,
  method = "pearson", absolute = TRUE, test = FALSE, limit = 1e7,
  tile = 2048L, precision = getOption("gputools.precision", "single"))
```

Arguments

`x`	a numeric matrix whose columns are the variables, with no missing values.
`y`	a numeric matrix with the same number of rows, or `NULL` for `x` with itself.
`threshold`	keep every pair whose correlation is at least this strong.
`top`	or instead, keep each column of `y`'s `top` strongest partners among the columns of `x`. Give one of `threshold` and `top`.
`method`	`"pearson"` or `"spearman"`.
`absolute`	`TRUE` to measure strength by $\lvert r \rvert$, so that strong negative correlations count; `FALSE` to keep positive ones only.
`test`	`TRUE` to add t statistics and two-sided p-values, as `gpuCor` computes them.
`limit`	the most pairs to return: a threshold that lets more through is an error, not a result of unexpected size. Raise it if you want them all.
`tile`	the GPU computes the correlation matrix in blocks of `tile` x `tile`. The default suits GPUs with a few gigabytes free; it has no effect on the result.
`precision`	`"single"` or `"double"`: the floating point precision the GPU computes in; see `gpuCor`.

Details

The correlations are the ones `cor` computes (Spearman's is Pearson's on the ranks, ties averaged). A constant column correlates with nothing, as its correlations are `NA`, and is never returned.

With `y = NULL`, a threshold returns each pair once, with `i < j`; `top` returns each column's partners, so a pair that is in both columns' lists appears twice, and a column is never its own partner.

A p-value cut-off α is a threshold on $|r|$, since every pair has the same n: $|r| \geq t/\sqrt{n-2+t^2}$ with t the $1-\alpha/2$ quantile of the t distribution on $n-2$ degrees of freedom. Remember how many pairs are tested when choosing α.

The whole of `x` (and `y`) is copied to the GPU, in the precision computed in; the correlation matrix itself never is.

Value

A data frame, a row per pair:

`i`	the column of `x`.
`j`	the column of `y` (of `x` when `y` is `NULL`).
`r`	their correlation.
`x`, `y`	the columns' names, if they have them.
`t`, `p.value`	with `test = TRUE`.

Rows are ordered by `j`, then by `i` for a threshold, and from strongest to weakest for `top`.

See Also

`gpuCor` for the whole matrix.

Examples

```
set.seed(1)
genes <- matrix(rnorm(50 * 2000), 50, 2000,
                dimnames = list(NULL, paste0("g", 1:2000)))
genes[, 2] <- genes[, 1] + rnorm(50, sd = 0.1)

# every pair correlated at |r| >= 0.6
head(gpuCorPairs(genes, threshold = 0.6))

# each gene's three strongest partners, with p-values
head(gpuCorPairs(genes, top = 3, test = TRUE))
```

---

gpuCorPerm *Permutation Tests of Correlations, With a GPU*

Description

Tests every correlation between the columns of x and those of y (or among the columns of x) by permutation: the rows of y are shuffled B times and each correlation's p-value is how often a shuffle gives one as strong. It also gives p-values adjusted for testing all the pairs at once, by Westfall and Young's maxT. Permutation p-values need no assumption of normality, which for small samples and skewed data the t test of gpuCor makes; each shuffle costs a whole correlation matrix, which the GPU computes in one matrix product.

Usage

```
gpuCorPerm(x, y = NULL, B = 1000L, method = "pearson",
  precision = getOption("gputools.precision", "single"))
```

Arguments

x	a numeric matrix whose columns are the variables, with no missing values.
y	a numeric matrix with the same number of rows, or NULL for x with itself.
B	the number of permutations. The smallest p-value possible is $1/(B+1)$.
method	"pearson" or "spearman".
precision	"single" or "double": the floating point precision the GPU computes in; see gpuCor.

Details

The permutations are drawn with sample.int, so set.seed makes the results reproducible. Each is applied to the rows of y (of a copy of x when y is NULL), and all of a permutation's correlations are computed from it, so the tests share their permutations, as the maxT adjustment needs.

The p-value of a correlation r is $(1+b)/(1+B)$, where b of the B permutations gave a correlation with $|r_b| \geq |r|$: two-sided, and counting the observed data as one of the permutations (Phipson

and Smyth 2010), so that it is never zero. The adjusted p-value counts instead the permutations whose largest $|r_b|$ over all the tests reaches $|r|$: single-step maxT (Westfall and Young 1993), which controls the family-wise error rate.

The whole `ncol(x)` by `ncol(y)` matrix is held on the GPU, twice; for wider data, split the columns of `y` (the maxT adjustment is then within each part).

Value

A list:

`coefficients`	the correlations, as `cor` gives them.
`p.value`	the permutation p-values.
`p.adjusted`	the maxT adjusted p-values.
`permutations`	`B`.

With `y = NULL` the diagonal is `NA`.

References

Westfall, P. H. and Young, S. S. (1993) *Resampling-Based Multiple Testing*. Wiley.

Phipson, B. and Smyth, G. K. (2010) Permutation p-values should never be zero. *Statistical Applications in Genetics and Molecular Biology* **9**, 39.

See Also

`gpuCor`, `cor.test`

Examples

```
set.seed(1)
# 15 samples of skewed data; one real association
x <- matrix(rexp(15 * 40), 15, 40)
y <- matrix(rexp(15 * 30), 15, 30)
y[, 1] <- y[, 1] + 2 * x[, 1]
res <- gpuCorPerm(x, y, B = 2000)
res$p.value[1, 1]
res$p.adjusted[1, 1]
min(res$p.adjusted[-1, -1])
```

gpuCrossprod — *Perform Matrix Cross-product with a GPU*

Description

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

Usage

```
gpuCrossprod(a, b=NULL)
```

Arguments

a	a matrix of floating point values.
b	a matrix of floating point values. A null value defaults to 'a'.

Value

A matrix of single precision floating point values. The matrix is the cross-product of arguments 'a' and 'b', i.e., t(a) * b.

Examples

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

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,
 precision = getOption("gputools.precision", "single"))
```

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.
precision	`"single"` or `"double"`: the floating point precision the GPU computes in. Single precision is the default, set with `options(gputools.precision = "double")`; it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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",
 beta = 0.5,
 precision = getOption("gputools.precision", "single"))
```

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".

`beta` the parameter of the `"flexible"` and `"flexible group"` methods, ignored by the others; it must be below 1. The flexible-beta update of Lance and Williams (1967) is

$$d(k, i \cup j) = \frac{1-\beta}{2}\,(d(k,i) + d(k,j)) + \beta\, d(i,j);$$

flexible UPGMA (`"flexible group"`, Belbin, Faith and Milligan 1992) weights $d(k,i)$ and $d(k,j)$ by the sizes of clusters i and j instead of equally. The default, 0.5, is the value earlier versions of gputools used (it was fixed then). Lance and Williams recommend a small negative value, `beta = -0.25`, which is the usual choice in the literature; positive values contract the space, so clusters tend to chain.

`precision` `"single"` or `"double"`: the floating point precision the GPU computes in. Single precision is the default, set with `options(gputools.precision = "double")`; it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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.

References

Lance, G. N. and Williams, W. T. (1967). A general theory of classificatory sorting strategies: 1. Hierarchical systems. *The Computer Journal*, 9(4), 373–380.

Belbin, L., Faith, D. P. and Milligan, G. W. (1992). A comparison of two approaches to beta-flexible clustering. *Multivariate Behavioral Research*, 27(3), 417–433.

See Also

gpuDist, gpuHclust.

Examples

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

---

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 = getOption("gputools.precision", "single") == "single",
 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 = getOption("gputools.precision", "single") == "single",
 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 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.
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	logical: TRUE computes in single precision on the GPU, FALSE in double. The default follows options(gputools.precision): single unless that is "double". It also sets the default tolerance (see gpuLm.defaultTol).
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 `1`s 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 for fitting log-linear models (which binomial and Poisson GLMs are) to contingency tables.

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)

# 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,
  precision = getOption("gputools.precision", "single"))
```

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.
precision	"single" or "double": the floating point precision the GPU computes in. Single precision is the default, set with options(gputools.precision = "double"); it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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.

Both matrices have dimnames cause (rows) and effect (columns), taken from the column names of x and y.

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[,i] Granger-cause variable x[,j]?", look at fStatistics[i, j] and pValues[i, j]. The diagonal, a variable against itself, is NA.

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 x[,i] Granger-cause variable y[,j]?", 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", beta = 0.5,
 precision = getOption("gputools.precision", "single"))
```

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".

`beta` the parameter of the `"flexible"` and `"flexible group"` methods, ignored by the others; it must be below 1. The flexible-beta update of Lance and Williams (1967) is

$$d(k, i \cup j) = \frac{1-\beta}{2}\,(d(k,i) + d(k,j)) + \beta\, d(i,j);$$

flexible UPGMA (`"flexible group"`, Belbin, Faith and Milligan 1992) weights $d(k,i)$ and $d(k,j)$ by the sizes of clusters i and j instead of equally. The default, 0.5, is the value earlier versions of gputools used (it was fixed then). Lance and Williams recommend a small negative value, `beta = -0.25`, which is the usual choice in the literature; positive values contract the space, so clusters tend to chain.

`precision` `"single"` or `"double"`: the floating point precision the GPU computes in. Single precision is the default, set with `options(gputools.precision = "double")`; it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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.

References

Lance, G. N. and Williams, W. T. (1967). A general theory of classificatory sorting strategies: 1. Hierarchical systems. *The Computer Journal*, 9(4), 373–380.

Belbin, L., Faith, D. P. and Milligan, G. W. (1992). A comparison of two approaches to beta-flexible clustering. *Multivariate Behavioral Research*, 27(3), 417–433.

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)
```

---

gpuKmeans *K-Means Clustering With a GPU*

Description

Partitions the rows of `x` into `k` clusters by Lloyd's algorithm, as `kmeans(x, centers, algorithm = "Lloyd")` does, with the distances and the cluster means computed on the GPU. It is for many points and many clusters: hundreds of thousands of cells of a single-cell experiment into hundreds of clusters, say.

Usage

```
gpuKmeans(x, centers, iter.max = 10L, nstart = 1L,
  precision = getOption("gputools.precision", "single"))
```

Arguments

`x`	a numeric matrix, a row per point, with no missing values.
`centers`	the number of clusters, `k`, or a `k`-row matrix of distinct starting centres.
`iter.max`	the most iterations to run.
`nstart`	with a number of clusters, how many random starts to try; the best (least total within-cluster sum of squares) is kept.
`precision`	`"single"` or `"double"`: the floating point precision the GPU computes in; see `gpuCor`.

Details

Each iteration assigns every point to its nearest centre and moves each centre to the mean of its points, until no assignment changes. The random starts are chosen exactly as `kmeans` chooses them, so with the same seed the two start from the same centres, and with the same starts they take the same steps: the results agree, apart from a point equally near two centres, which rounding can send either way (more often in single precision).

`kmeans`' default is the Hartigan-Wong algorithm, which also moves single points between clusters and usually ends at a lower within-cluster sum of squares from the same start; Lloyd's is the one that parallelises. A few random starts (`nstart`) make up much of the difference.

As `kmeans` does, it warns of a cluster left empty (its centre is then `NaN`) and of not converging in `iter.max` iterations.

Value

An object of class "kmeans", as kmeans returns, so its print and fitted methods apply.

See Also

kmeans

Examples

```
set.seed(1)
x <- rbind(matrix(rnorm(2000, sd = 0.3), ncol = 2),
           matrix(rnorm(2000, mean = 2, sd = 0.3), ncol = 2))
cl <- gpuKmeans(x, 2, nstart = 5)
cl$size
cl$centers

# the same starts give the same clustering as kmeans()
set.seed(2)
g <- gpuKmeans(x, 4, precision = "double")
set.seed(2)
k <- kmeans(x, 4, algorithm = "Lloyd")
all(g$cluster == k$cluster)
```

---

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 = getOption("gputools.precision", "single") == "single", 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	logical: TRUE computes in single precision on the GPU, FALSE in double. The default follows options(gputools.precision): single unless that is "double". It also sets the default tolerance (see gpuLm.defaultTol).
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 `NA`s would invalidate the time series attributes, and if `NA`s 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 = getOption("gputools.precision", "single") == "single",
 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: `TRUE` computes in single precision on the GPU, `FALSE` in double. The default follows `options(gputools.precision)`: single unless that is `"double"`. It also sets the default tolerance (see `gpuLm.defaultTol`).
`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 = getOption("gputools.precision", "single") == "single",
  tolerance=gpuLm.defaultTol(useSingle),
  yname = NULL)
```

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	logical: TRUE computes in single precision on the GPU, FALSE in double. The default follows options(gputools.precision): single unless that is "double". It also sets the default tolerance (see gpuLm.defaultTol).
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 wrapper for the CUBLAS cublasDgemm function.

Usage

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

Arguments

a	a numeric matrix.
b	a numeric matrix.

Value

A numeric matrix. 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,
  precision = getOption("gputools.precision", "single"))
```

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.
`precision`	`"single"` or `"double"`: the floating point precision the GPU computes in. Single precision is the default, set with `options(gputools.precision = "double")`; it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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 doi:10.1186/147121055118

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)
```

gpuPrcomp *Principal Components Analysis on a GPU*

Description

Principal components of a data matrix, as prcomp computes them, with the singular value decomposition on the GPU (gpuSvd).

Usage

```
gpuPrcomp(x, retx = TRUE, center = TRUE, scale. = FALSE, tol = NULL,
 rank. = NULL, precision = getOption("gputools.precision", "single"))
```

Arguments

x, retx, center, scale., tol, rank.
: as for prcomp.

precision
: "single" or "double": the floating point precision the GPU computes in. Single precision is the default, set with options(gputools.precision = "double"); it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

Value

An object of class "prcomp", as prcomp returns, so summary, predict, biplot and screeplot work on it. Each component's sign is arbitrary, as with any principal components, so a column of rotation and of x can be the negative of prcomp's.

See Also

prcomp, gpuSvd

Examples

```
pc <- gpuPrcomp(USArrests, scale. = TRUE)
summary(pc)
head(predict(pc, USArrests), 3)
```

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,
  precision = getOption("gputools.precision", "single"))
```

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.
precision	"single" or "double": the floating point precision the GPU computes in. Single precision is the default, set with options(gputools.precision = "double"); it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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)
```

`gpuRowTtest` *Two-Group t Tests of Every Row, With Permutation p-Values on a GPU*

Description

Tests every row of `x` (every gene of an expression matrix, say) for a difference in mean between two groups of columns, as `t.test` does one row at a time, and optionally gives each test a permutation p-value and a p-value adjusted for testing all the rows, by Westfall and Young's maxT. The permutations run on the GPU, every row under every relabelling of the samples at once.

Usage

```
gpuRowTtest(x, group, var.equal = FALSE, B = 0L,
  precision = getOption("gputools.precision", "single"))
```

Arguments

`x`	a numeric matrix, a row per variable (gene), a column per sample, with no missing values.
`group`	the samples' groups: a vector or factor with one entry per column of `x` and exactly two levels, at least two samples in each.
`var.equal`	`FALSE` for Welch's test, `TRUE` for Student's with the variances pooled, as in `t.test`.
`B`	the number of permutations, or 0 for none.
`precision`	`"single"` or `"double"`: the floating point precision the GPU computes the permutations in; see `gpuCor`.

Details

The t statistics, degrees of freedom and parametric p-values are those of `t.test(x[i, group == level1], x[i, group == level2],var.equal = var.equal)`, the first level's mean minus the second's. They are computed in double precision on the CPU: a t test of each row is too little work to be worth sending to the GPU.

With `B > 0`, the samples' labels are permuted `B` times (drawn with `sample.int`, so `set.seed` makes them reproducible) and every row's t statistic is computed under each permutation, on the GPU: two matrix products give every row's group sums and sums of squares under all the labellings. A row's permutation p-value is $(1 + b)/(1 + B)$ with b the permutations giving $|t_b| \geq |t|$ (Phipson and Smyth 2010); its adjusted p-value counts the permutations whose largest $|t_b|$ over all the rows reaches $|t|$: single-step maxT, which controls the family-wise error rate (Westfall and Young 1993). Permuting the labels keeps the dependence between genes, which the adjustment exploits, and makes no assumption of normality.

A constant row, or one constant within both groups, has no test: its statistic is `NaN` and its p-values `NA`.

Value

A data frame, a row per row of `x`, named as `x`'s rows:

`mean1`, `mean2`	the groups' means.
`statistic`	the t statistic.
`df`	its degrees of freedom.
`p.value`	the two-sided p-value from the t distribution.
`p.perm`	with `B > 0`, the permutation p-value.
`p.adjusted`	with `B > 0`, the maxT adjusted p-value.

References

Westfall, P. H. and Young, S. S. (1993) *Resampling-Based Multiple Testing*. Wiley.

Phipson, B. and Smyth, G. K. (2010) Permutation p-values should never be zero. *Statistical Applications in Genetics and Molecular Biology* **9**, 39.

See Also

`t.test`, `gpuCorPerm`

Examples

```
set.seed(1)
genes <- matrix(rnorm(2000 * 12), 2000, 12,
                dimnames = list(paste0("g", 1:2000), NULL))
group <- rep(c("treated", "control"), each = 6)
genes[1:5, group == "treated"] <- genes[1:5, group == "treated"] + 3
res <- gpuRowTtest(genes, group, B = 5000)
head(res[order(res$p.adjusted), ])
```

gpuSolve *Estimate the solution to a matrix vector equation*

Description

Solves x b = y for b on the GPU, like `solve`. If y is omitted, returns the inverse of x.

A square x is solved by LU decomposition with partial pivoting, as `solve` does (NVIDIA's cuSOLVER, `getrf`/`getrs`). An x with more rows than columns is solved in the least-squares sense by Householder QR (`geqrf`/`ormqr`), and its "inverse" is the pseudo-inverse.

Usage

```
gpuSolve(x, y=NULL,
 precision = getOption("gputools.precision", "single"))
```

Arguments

x	a matrix of floating point numbers.
y	a vector of length nrow(x), or a matrix with nrow(x) rows.
precision	`"single"` or `"double"`: the floating point precision the GPU computes in. Single precision is the default, set with `options(gputools.precision = "double")`; it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

Value

b: a vector of length ncol(x) if y is a vector, otherwise a matrix with ncol(x) rows. If y is omitted, the ncol(x) by nrow(x) inverse (or pseudo-inverse) of x. Rows are named by the columns of x, as with `solve`.

A square x that is exactly singular is an error, as with `solve`; so is an overdetermined x whose columns are linearly dependent, since the least-squares solution is then not unique.

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 on a GPU*

Description

The singular value decomposition of a matrix, as svd computes it, on the GPU with NVIDIA's cuSOLVER (gesvd).

Usage

```
gpuSvd(x, nu = min(n, p), nv = min(n, p),
 precision = getOption("gputools.precision", "single"))
```

Arguments

x	a numeric matrix, without missing or infinite values.
nu, nv	the number of left and right singular vectors to return, as in svd; at most min(nrow(x), ncol(x)), since the thin decomposition is computed.
precision	"single" or "double": the floating point precision the GPU computes in. Single precision is the default, set with options(gputools.precision = "double"); it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

Value

As svd: a list with d, the singular values in decreasing order, and, when asked for, u and v, the left and right singular vectors as columns. As with any SVD, each pair of singular vectors is determined only up to sign, so their signs can differ from svd's.

See Also

svd, gpuPrcomp

Examples

```
x <- matrix(rnorm(200 * 20), 200, 20)
s <- gpuSvd(x, precision = "double")
s$d
max(abs(x - s$u %*% diag(s$d) %*% t(s$v)))
```

gpuTcrossprod — *Perform Matrix Transposed Cross-product with a GPU*

Description

Performs transposed matrix cross-product using a GPU. This function is merely a couple of wrappers for the CUBLAS cublasSgemm function.

Usage

```
gpuTcrossprod(a, b)
```

Arguments

a	a matrix of floating point values.
b	a matrix of floating point values. If null, defaultsto 'a'.

Value

A matrix of single precision floating point values. The matrix is the transposed cross-product of arguments 'a' and 'b', i.e., a * t(b).

Examples

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

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,
 precision = getOption("gputools.precision", "single"))
```

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.
`precision`	`"single"` or `"double"`: the floating point precision the GPU computes in. Single precision is the default, set with `options(gputools.precision = "double")`; it runs many times faster on most consumer GPUs. Results are returned as doubles either way.

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

∗ **algebra**
- cpuMatMult, 3
- gpuCrossprod, 12
- gpuMatMult, 32
- gpuSvd, 39
- gpuTcrossprod, 40

∗ **array**
- cpuMatMult, 3
- gpuCrossprod, 12
- gpuLm.fit, 29
- gpuMatMult, 32
- gpuTcrossprod, 40

∗ **cluster**
- gpuDistClust, 13
- gpuHclust, 21

∗ **math**
- gpuDist, 12

∗ **models**
- gpuGlm, 15

∗ **multivariate**
- gpuPrcomp, 34

∗ **regression**
- gpuGlm, 15
- gpuLm, 24
- gpuLm.fit, 29
- gpuLsfit, 30

∗ **utilities**
- gpuAvailable, 6

anova, *17*, *19*, *26*
anova.glm, *17*, *19*
anova.lm, *27*
anscombe, *27*
aov, *24*, *26, 27*
as.data.frame, *16*, *25*
attitude, *27*

binomial, *19*

chooseGpu, 3, *7*
class, *26*
coef, *27*
coefficients, *17*
confint, *27*
cor, *7, 8*
cor.test, *7, 8*, *11*
cpuMatMult, 3

effects, *19*, *26, 27*

factor, *17*
family, *16*, *18*
fitted, *27*
fitted.values, *19*
formula, *16*, *25, 26*
freeny, *27*

getGpuId, 4, *7*
glm, *27*
glm.control, *16, 17*
gpuAssoc, 4
gpuAvailable, 6
gpuCor, *5*, 7, *8–11*, *23*, *36*, *41*
gpuCorPairs, *5, 6*, 8
gpuCorPerm, 10, *37*
gpuCrossprod, 12
gpuDist, 12, *15*
gpuDistClust, 13, *22*
gpuGlm, 15
gpuGranger, 20
gpuHclust, *15*, 21
gpuKmeans, 23
gpuLm, 24, *28–30*
gpuLm.defaultTol, *16*, *25*, 28, *29*, *31*
gpuLm.fit, *28*, 29
gpuLsfit, 30
gpuMatMult, 32
gpuMi, 32
gpuPrcomp, 34, *39*
gpuQr, 35

gpuRowTtest, 36
gpuSolve, 38
gpuSvd, *34*, 39
gpuTcrossprod, 40
gpuTtest, 40

kmeans, *23, 24*

LifeCycleSavings, *27*
lm, *6*, *19*, *31*
lm.fit, *26, 27*
lm.influence, *27*
lm.wfit, *27*
loglin, *19*
longley, *27*
ls.diag, *31*
ls.print, *31*
lsfit, *31*

model.frame, *19*, *26*
model.matrix, *26*
model.matrix.default, *25*
model.offset, *16*, *25*

na.exclude, *16*, *25*
na.fail, *16*, *25*
na.omit, *16*, *25*

offset, *16*, *25, 26*
options, *16*, *25*

prcomp, *34*
predict, *27*
predict.lm, *27*

qr, *29*
quasi, *17*

residuals, *19*, *27*

sample.int, *10*, *37*
solve, *38*
stackloss, *27*
summary, *17*, *19*
summary.glm, *17*, *19*
summary.lm, *27*
svd, *39*
swiss, *27*

t.test, *36, 37*

terms, *18*, *26*
ts.intersect, *27*

vcov, *27*