Package 'cudaverse'

September 10, 2026

Title Lightweight 'CUDA' Numerical Computing

Version 0.4.1

Description Provides a lightweight interface to graphics processing unit (GPU)-accelerated numerical computing using 'CUDA'. Dense tensors, sparse matrices, decompositions, distances, exact nearest neighbours, clustering, graph workflows, and embeddings share one consistent interface. The native backend discovers the 'NVIDIA CUDA Driver API', 'cuBLAS', and 'cuSOLVER' libraries at runtime without bundling 'LibTorch' or the 'CUDA Runtime'. Stage-level provenance records the backend, device, and data transfers used by each result. A portable implementation supports package validation on systems without 'CUDA'. Background for the included Leiden community detection and uniform manifold approximation and projection methods is given by Traag, Waltman and van Eck (2019) <doi:10.1038/s41598-019-41695-z> and McInnes et al. (2018) <doi:10.21105/joss.00861>, respectively.

License MIT + file LICENSE

URL https://cudaverse.github.io/cudaverse/, https://github.com/cudaverse/cudaverse

BugReports https://github.com/cudaverse/cudaverse/issues

Encoding UTF-8

Language en-US

Imports Matrix, methods, stats

Suggests igraph (>= 2.0.0), knitr, rmarkdown, RSpectra, Rtsne, S4Vectors, SingleCellExperiment, testthat (>= 3.0.0), torch, uwot

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

SystemRequirements For GPU execution on Windows or Linux: NVIDIA CUDA-capable GPU, NVIDIA driver with CUDA Driver API, NVIDIA cuBLAS 12, and NVIDIA cuSOLVER 11

NeedsCompilation yes

Author Yaoxiang Li [aut, cre]

Maintainer Yaoxiang Li <liyaoxiang@outlook.com>

Repository CRAN

Date/Publication 2026-09-10 09:30:02 UTC

Contents

`as_adjacency_matrix` *Extract a graph adjacency matrix*

Description

Extract a graph adjacency matrix

Usage

```
as_adjacency_matrix(graph)
```

Arguments

`graph`	A `cuda_graph`.

Value

A symmetric sparse `Matrix::dgCMatrix`.

Examples

```
index <- matrix(c(2, 3, 1, 3, 1, 2), 3, byrow = TRUE)
distance <- matrix(c(1, 2, 1, 1, 2, 1), 3, byrow = TRUE)
graph <- cuda_knn_graph(list(index = index, distance = distance))
as_adjacency_matrix(graph)
```

`as_coo` *Convert sparse storage format*

Description

Convert sparse storage format

Usage

```
as_coo(x)

as_csr(x)
```

Arguments

x	A cudasparse matrix.

Value

A cudasparse matrix.

Examples

```
x <- cuda_sparse(diag(3), device = "cpu")
as_coo(x)
as_csr(x)
```

---

cudatensor-operators *Arithmetic operators for GPU-aware tensors*

Description

cudatensor objects support element-wise +, -, *, /, and ^. Operands follow trailing-dimension broadcasting. Mixed dtypes are promoted without silently truncating fractional values; integer arithmetic is promoted to float64 to avoid R integer overflow. Compatible dimension labels are retained. When both operands label the same non-broadcast dimension, their labels must be identical.

Usage

```
## S3 method for class 'cudatensor'
Ops(e1, e2)

## S3 method for class 'cudatensor'
x %*% y
```

Arguments

e1, e2	A cudatensor or numeric object for element-wise arithmetic.
x, y	A cudatensor or numeric matrix for matrix multiplication.

Details

Use %*% for matrix multiplication.

Value

A cudatensor on the device of the tensor operand on the left (or the tensor operand on the right when the left operand is a base object).

Examples

```
x <- cuda_tensor(matrix(1:6, 2, 3), device = "cpu")
to_cpu(x + c(0.5, 1, 1.5))

y <- cuda_tensor(matrix(1:6, 3, 2), device = "cpu")
to_cpu(x %*% y)
```

---

cudatensor-subset *Subset and replace tensor values*

Description

Tensor indices follow ordinary one-based R array semantics. Subsetting returns a `cudatensor`, including when a single value is selected. Replacement preserves the tensor dtype; fractional values therefore cannot be assigned to an integer tensor.

Usage

```
## S3 method for class 'cudatensor'
x[..., drop = TRUE]

## S3 replacement method for class 'cudatensor'
x[...] <- value
```

Arguments

`x`	A `cudatensor`.
`...`	One-based R array indices.
`drop`	Whether dimensions of length one are dropped.
`value`	Numeric replacement values or another `cudatensor`.

Details

Backends may implement value gathering and replacement directly. The native CUDA backend evaluates only R index metadata on the host and keeps tensor values on the device. A selection whose linear indices form one increasing contiguous range becomes an allocation-free view that shares the source device allocation; other supported selections use a device gather. Compatibility backends without indexing operations use a recorded CPU round trip. Subscripts containing `NA` currently use the compatibility path. A replacement tensor on the same device and backend is cast to the target floating dtype on that device before replacement. Integer targets retain exact host validation for non-integer replacement values.

Value

A `cudatensor` on the same device as `x`.

cuda_available *Detect a usable CUDA backend*

Description

Detection uses the built-in lightweight native backend or a CUDA-enabled installation of the optional `torch` package. NVIDIA libraries are loaded only when diagnostics or CUDA selection is requested.

Usage

```
cuda_available()
```

Value

A single logical value.

Examples

```
cuda_available()
```

cuda_diagnostics *Diagnose the optional CUDA runtime*

Description

Inspecting the runtime is non-destructive and never installs or downloads torch. The returned `reason` is suitable for logs and provenance.

Usage

```
cuda_diagnostics()
```

Value

A named list containing the legacy fields `torch_installed`, `torch_version`, `cuda_available`, `cuda_device_count`, `reason`, and `detection_error`, plus `available_backends`, `auto_eligible_backends`, `auto_selection_reason`, `selected_backend`, and per-backend diagnostic details. The additive `status`, `summary`, `next_steps`, and `backend_status` fields provide a user-facing health result without removing the machine-readable details. Each backend detail distinguishes advertised `capabilities` from callable internal `operations`. Native automatic eligibility requires a compatible backend contract, the complete tensor/algorithm capability set, all runtime components, and a passing cached self-test. The legacy fields are retained throughout the 0.4 compatibility cycle.

Examples

```
cuda_diagnostics()
```

`cuda_diffusion_map` *Diffusion-map-style embedding*

Description

Pairwise distances can use the cudaverse CUDA path. Kernel construction and eigendecomposition currently run on the CPU.

Usage

```
cuda_diffusion_map(
  x,
  n_components = 2L,
  sigma = NULL,
  diffusion_time = 1,
  metric = c("euclidean", "cosine"),
  device = c("auto", "cuda", "cpu"),
  reduced_dim = NULL
)
```

Arguments

`x`	Numeric observation-by-feature matrix, compatible cudaverse result, or a `SingleCellExperiment` with a reduced dimension.
`n_components`	Output dimensions.
`sigma`	Gaussian kernel bandwidth. Defaults to the median positive pairwise distance.
`diffusion_time`	Non-negative diffusion time exponent.
`metric`	Euclidean or cosine distance.
`device`	Device passed to `cuda_distance()`.
`reduced_dim`	For a `SingleCellExperiment`, the reduced-dimension name to embed. See `cuda_umap()` for automatic selection.

Value

A `cuda_embedding` with the stable fields documented by `cuda_umap()`, stage-level distance/kernel/eigendecomposition provenance, an optional `distance_input` stage when resident native storage is reused, and an additional `eigenvalues` element.

Examples

```
cuda_diffusion_map(
  matrix(rnorm(120), 40, 3),
  n_components = 2,
  device = "cpu"
)
```

cuda_distance *Pairwise distances with an optional CUDA backend*

Description

Pairwise distances with an optional CUDA backend

Usage

```
cuda_distance(
  x,
  y = NULL,
  metric = c("euclidean", "cosine"),
  device = c("auto", "cuda", "cpu"),
  batch_size = 256L
)
```

Arguments

x, y	Numeric matrices with observations in rows. When y is NULL, computes all pairwise distances within x.
metric	"euclidean" or "cosine".
device	One of "auto", "cuda", or "cpu".
batch_size	Maximum number of query rows in each compute block. The final dense result is still allocated in host memory.

Details

On CPU, Euclidean distances use a common translation and global scaling before a vectorized calculation. Pairs at risk of cancellation or non-finite intermediate results are recomputed from direct observation differences with a scale-first norm. This avoids cancellation from large shared offsets and avoids avoidable overflow and underflow for extreme finite values. All built-in backends honor batch_size. The native CUDA backend uploads each input once, keeps the reference matrix and its norms device-resident, and transfers only completed distance blocks to R. This bounds operation-owned device memory without silently changing backend.

Value

A dense numeric distance matrix with a device attribute. Input observation names are retained as row and column names when present.

Examples

```
cuda_distance(matrix(1:12, 4, 3), device = "cpu")
```

cuda_kmeans *GPU-aware k-means clustering*

Description

GPU-aware k-means clustering

Usage

```
cuda_kmeans(
  x,
  centers,
  iter.max = 100L,
  tolerance = 1e-06,
  seed = NULL,
  batch_size = 256L,
  device = c("auto", "cuda", "cpu")
)
```

Arguments

x	Numeric matrix with observations in rows.
centers	Number of clusters or a matrix of initial centres.
iter.max	Maximum Lloyd iterations.
tolerance	Convergence tolerance for centre movement.
seed	Optional random seed used for initial centres.
batch_size	Maximum number of observations whose centre-distance block is materialized at once. The native backend keeps observations, centres, assignments, and updates on the GPU while bounding temporary distance storage to approximately batch_size * n_centers values.
device	Device used for the numerical clustering stages.

Details

The native CUDA backend uploads the observations and initial centres once, then keeps distance calculation, deterministic assignment, accumulation, and centre updates on the device. Only the small convergence movement summary is inspected between iterations; final assignments, centres, and within-cluster sums are transferred to R. Compatibility backends without a resident k-means operation retain the established distance-on-backend and update-on-CPU implementation.

Value

A cuda_kmeans list containing integer cluster assignments, final centers, per-cluster withinss, tot.withinss, the number of iteration count in iter, a logical converged flag, and the actual distance device. Observation and feature names are retained when supplied.

Examples

```
set.seed(1)
x <- rbind(matrix(rnorm(40), 20, 2), matrix(rnorm(40, 4), 20, 2))
cuda_kmeans(x, centers = 2, seed = 1, device = "cpu")
```

---

cuda_knn *k-nearest neighbours*

Description

k-nearest neighbours

Usage

```
cuda_knn(
  x,
  k = 15L,
  metric = c("euclidean", "cosine"),
  device = c("auto", "cuda", "cpu"),
  batch_size = 256L
)
```

Arguments

x	Numeric matrix or cudasparse object with observations in rows.
k	Number of neighbours.
metric	Exact distance metric, "euclidean" or "cosine".
device	One of "auto", "cuda", or "cpu".
batch_size	Maximum number of query rows in each dense distance block. Larger batches may be faster but use more memory.

Details

Neighbours are exact: every row is compared with every other row. The observation itself is always excluded. Equal distances are resolved deterministically in favour of the smaller row index.

The implementation constructs at most a min(batch_size, nrow(x))-by-nrow(x) dense distance block instead of a complete pairwise distance matrix. The native CUDA backend keeps distance blocks and deterministic top-k selection on the GPU. A torch backend with stable-sort support follows the same residency contract. Both transfer only the final n-by-k index and distance matrices. Compatibility backends without device-side selection transfer each distance block to the CPU for stable ordering. On CPU, Euclidean blocks use the same guarded translated-and-scaled implementation as cuda_distance().

Value

A cuda_knn list with index and distance matrices of size nrow(x) by k, followed by the selected metric and actual device. Neighbours in every row are ordered by distance and then row index. When x has row names, both matrices retain them as query identifiers; neighbour identities can be recovered with rownames(result$index)[result$index].

Examples

```
cuda_knn(
  matrix(rnorm(30), 10, 3),
  k = 3,
  batch_size = 4,
  device = "cpu"
)
```

---

cuda_knn_graph *Build a sparse graph from nearest neighbours*

Description

The input may have been computed on CUDA, but graph assembly itself is currently performed on the CPU with a sparse Matrix. Union graphs retain an edge observed in either direction, whereas mutual graphs require both directed neighbour relations. When the two directions have different weights, the undirected edge retains the stronger affinity.

Usage

```
cuda_knn_graph(
  neighbors,
  weighting = c("binary", "distance", "gaussian"),
  symmetrize = c("union", "mutual"),
  sigma = NULL
)
```

Arguments

neighbors	A cuda_knn() result or compatible list.
weighting	Edge weighting: binary, inverse-distance, or Gaussian.
symmetrize	Keep the union or only mutual nearest-neighbour edges.
sigma	Gaussian bandwidth. Defaults to the median positive distance.

Value

A cuda_graph list containing sparse adjacency, counts of vertices and undirected edges, weighting, symmetrize, source_device, and the graph-assembly backend. Named kNN observations are retained as adjacency dimnames and in vertex_names.

Examples

```
index <- matrix(c(2, 3, 1, 3, 1, 2), 3, byrow = TRUE)
distance <- matrix(c(1, 2, 1, 1, 2, 1), 3, byrow = TRUE)
cuda_knn_graph(list(index = index, distance = distance))
```

---

cuda_leiden *Cluster a graph with Leiden*

Description

Community detection currently runs on the CPU through `igraph`.

Usage

```
cuda_leiden(graph, resolution = 1, n_iterations = 2L)
```

Arguments

`graph`	A `cuda_graph`.
`resolution`	Positive modularity resolution.
`n_iterations`	Number of Leiden refinement iterations.

Value

A `cuda_communities` list with the stable fields documented by `cuda_louvain()`.

Examples

```
index <- matrix(c(2, 3, 1, 3, 1, 2), 3, byrow = TRUE)
distance <- matrix(c(1, 2, 1, 1, 2, 1), 3, byrow = TRUE)
graph <- cuda_knn_graph(list(index = index, distance = distance))
if (requireNamespace("igraph", quietly = TRUE)) {
  cuda_leiden(graph)
}
```

cuda_louvain *Cluster a graph with Louvain*

Description

Community detection currently runs on the CPU through `igraph`.

Usage

```
cuda_louvain(graph, resolution = 1)
```

Arguments

`graph`	A `cuda_graph`.
`resolution`	Positive modularity resolution.

Value

A `cuda_communities` list containing integer `membership`, the number of `communities`, `modularity`, `algorithm`, `resolution`, `source_device`, and clustering `backend`. Membership is named when the graph has vertex identifiers.

Examples

```
index <- matrix(c(2, 3, 1, 3, 1, 2), 3, byrow = TRUE)
distance <- matrix(c(1, 2, 1, 1, 2, 1), 3, byrow = TRUE)
graph <- cuda_knn_graph(list(index = index, distance = distance))
if (requireNamespace("igraph", quietly = TRUE)) {
  cuda_louvain(graph)
}
```

cuda_memory_info *Inspect CUDA memory*

Description

Reports physical device memory when the selected backend exposes it and allocator-owned current and peak bytes when those counters are available. The native backend reports physical CUDA-driver memory plus allocations owned by cudaverse. The optional torch backend reports its allocator's allocated and reserved bytes. CPU selection returns an unavailable report rather than pretending host RAM is CUDA memory.

Usage

```
cuda_memory_info(device = c("auto", "cuda", "cpu"))
```

Arguments

device	Requested device: "auto", "cuda", or "cpu".

Details

This function does not reset allocator peaks or retain a user tensor. The first CUDA selection in an R session can run the small runtime self-test, so native peak bytes can include its released temporary allocations. An automatic request is safe on a machine without CUDA and records the CPU fallback. An explicit device = "cuda" request remains strict.

Value

A cuda_memory_info list with selection metadata, physical total_bytes, free_bytes, and used_bytes, allocator allocated_bytes, allocated_peak_bytes, reserved_bytes, and reserved_peak_bytes, plus reason and any captured error. Unsupported counters are NA_real_ rather than estimated.

Examples

```
cuda_memory_info("cpu")
cuda_memory_info("auto")
```

---

cuda_pca *GPU-aware principal component analysis*

Description

GPU-aware principal component analysis

Usage

```
cuda_pca(
  x,
  n_components = 2L,
  center = TRUE,
  scale. = FALSE,
  device = c("auto", "cuda", "cpu")
)
```

Arguments

x	A matrix or cudasparse object with observations in rows and features in columns.
n_components	Number of components to return.
center	Whether to centre features.
scale.	Whether to scale features to unit variance.
device	One of "auto", "cuda", or "cpu".

Details

A native CUDA cudatensor selected on the same backend is validated on the device and passed directly into preprocessing and cuSOLVER without downloading its input matrix. Float32 and integer tensors are converted to float64 on the device. PCA scores retain shared native storage for direct composition with native distance and kNN operations. Sparse inputs transfer directly from their stable COO mirror. Constant features are scanned from that mirror only when scale. = TRUE; unscaled sparse PCA does not build or scan an intermediate Matrix object.

Value

A cuda_pca object with scores in x, loadings in rotation, standard deviations, centring/scaling values, and actual device. Observation names, feature names, and stable PC1, PC2, ... component names are preserved on every backend.

Examples

```
fit <- cuda_pca(iris[, 1:4], n_components = 2, device = "cpu")
fit
```

---

cuda_provenance *Inspect actual compute provenance*

Description

Returns one row per computation stage. The table prevents an "auto" request, a CUDA-aware kernel, or a hybrid pipeline from being mistaken for end-to-end GPU execution.

Usage

```
cuda_provenance(x)

## Default S3 method:
cuda_provenance(x)
```

Arguments

x	A cudaverse result or a named list of cuda_stage records.

Details

cuda_provenance() is the canonical cudaverse S3 generic. Extension packages can register methods for container classes while ordinary cudaverse results continue through the default method.

Value

A cuda_provenance data frame with columns stage, requested_device, device, backend, selection_reason, fallback, and output_device. Its schema and compute_device attributes contain the contract version and aggregate actual compute device.

Examples

```
x <- cuda_tensor(matrix(1:6, 2, 3), device = "cpu")
cuda_provenance(x)
```

`cuda_select_device` *Select a computation device without hiding fallback*

Description

`"auto"` may select CPU when CUDA is unavailable and records why. Explicit `"cuda"` is strict: it signals a `cudaverse_cuda_unavailable` error instead of silently falling back.

Usage

```
cuda_select_device(device = c("auto", "cuda", "cpu"))
```

Arguments

`device`	Requested device: `"auto"`, `"cuda"`, or `"cpu"`.

Value

A named `cuda_device_selection` list containing the original request, selected device, selection reason, fallback flag, and diagnostics.

Examples

```
cuda_select_device("cpu")
cuda_select_device("auto")
```

`cuda_sparse` *Create a GPU-aware sparse matrix*

Description

Create a GPU-aware sparse matrix

Usage

```
cuda_sparse(
  x,
  format = c("csr", "coo"),
  device = c("auto", "cuda", "cpu"),
  drop_zeros = TRUE
)
```

Arguments

x	A numeric matrix, a sparse matrix from the Matrix package, or a cudasparse object.
format	Logical storage format, "csr" or "coo".
device	One of "auto", "cuda", or "cpu".
drop_zeros	Whether to remove explicitly stored zeros.

Details

Existing cudasparse inputs use their stable sorted COO mirror directly. Same-device format changes share backend storage; transfers and zero filtering do not construct an intermediate Matrix object.

Value

A cudasparse list. Stable public metadata include one-based COO i and j, numeric values, zero-based CSR row_ptr and col_index, integer shape, matrix dimnames, logical format, actual device, and backend. storage is backend-internal and should not be accessed directly.

Examples

```
library(Matrix)
x <- rsparsematrix(5, 4, density = 0.25)
cuda_sparse(x, device = "cpu")
```

---

cuda_stage *Record one compute stage*

Description

cuda_stage() is the shared constructor for cudaverse packages and extensions. It distinguishes the requested device, actual compute device, implementation backend, and device holding the returned value.

Usage

```
cuda_stage(
  requested_device,
  device,
  backend,
  selection_reason,
  fallback = FALSE,
  output_device = device
)
```

Arguments

requested_device	"auto", "cpu", "cuda", "fixed-cpu", or "inherited".
device	Actual compute device, "cpu" or "cuda".
backend	Concrete implementation backend.
selection_reason	Stable reason describing device selection.
fallback	Whether an "auto" request fell back to CPU.
output_device	Device holding the returned value. Defaults to device.

Value

A validated cuda_stage list.

Examples

```
cuda_stage(
  requested_device = "auto",
  device = "cpu",
  backend = "base",
  selection_reason = "cuda_unavailable",
  fallback = TRUE
)
```

---

cuda_svd *GPU-aware singular value decomposition*

Description

GPU-aware singular value decomposition

Usage

```
cuda_svd(
  x,
  nu = min(nrow(x), ncol(x)),
  nv = min(nrow(x), ncol(x)),
  device = c("auto", "cuda", "cpu")
)
```

Arguments

x	A finite numeric matrix or cudatensor.
nu, nv	Number of left and right singular vectors to return.
device	One of "auto", "cuda", or "cpu".

Details

A native CUDA cudatensor selected on the same backend is validated for finite values on the device and passed directly to cuSOLVER. Float32 and integer storage is converted to float64 on the device. Only the requested decomposition results are materialized in R.

Value

A list with d, u, v, and the actual device. Matrix row and column names are retained on the corresponding singular vectors.

Examples

```
cuda_svd(matrix(rnorm(30), 10, 3), device = "cpu")
```

---

cuda_tensor *Create a GPU-aware tensor*

Description

Create a GPU-aware tensor

Usage

```
cuda_tensor(x, device = c("auto", "cuda", "cpu"), dtype = NULL)
```

Arguments

x	Numeric vector, matrix, array, or another cudatensor.
device	One of "auto", "cuda", or "cpu". Auto selects CUDA only when cuda_available() is true.
dtype	One of "float64", "float32", or "integer". Matrix and array dimnames, including names on a one-dimensional input, are retained as R metadata on both CPU and CUDA tensors. Floating dtypes accept IEEE Inf, -Inf, NaN, and R's floating NA; torch backends may normalize NA to NaN. Integer dtype rejects non-finite or fractional values because they have no exact integer representation. When x is already a tensor on the selected device, compatible floating dtype changes use its current backend without materializing the tensor on the host. Conversion to integer still validates exact representability on the host.

Value

A cudatensor object.

Examples

```
x <- cuda_tensor(matrix(1:6, nrow = 2), device = "cpu")
x
```

cuda_tsne *t-SNE embedding*

Description

t-SNE currently uses the CPU Rtsne backend.

Usage

```
cuda_tsne(
  x,
  n_components = 2L,
  perplexity = 30,
  theta = 0.5,
  seed = NULL,
  ...,
  reduced_dim = NULL
)
```

Arguments

x	Numeric observation-by-feature matrix, compatible cudaverse result, or a SingleCellExperiment with a reduced dimension.
n_components	Output dimensions.
perplexity	t-SNE perplexity.
theta	Barnes-Hut accuracy/speed trade-off.
seed	Optional random seed.
...	Additional arguments passed to Rtsne::Rtsne().
reduced_dim	For a SingleCellExperiment, the reduced-dimension name to embed. When NULL, a compatible recorded metadata choice is used first, followed by a uniquely named "PCA". Other names must be selected explicitly.

Value

A cuda_embedding; see cuda_umap() for the stable result fields.

Examples

```
if (requireNamespace("Rtsne", quietly = TRUE)) {
  cuda_tsne(matrix(rnorm(120), 40, 3), perplexity = 5, seed = 1)
}
```

cuda_umap *UMAP embedding*

Description

UMAP currently uses the CPU uwot backend. GPU-aware cudaverse inputs are accepted and their source device is retained in the result metadata.

Usage

```
cuda_umap(
  x,
  n_components = 2L,
  n_neighbors = 15L,
  min_dist = 0.1,
  metric = "euclidean",
  n_epochs = NULL,
  seed = NULL,
  ...,
  reduced_dim = NULL
)
```

Arguments

x	Numeric observation-by-feature matrix, compatible cudaverse result, or a SingleCellExperiment with a reduced dimension.
n_components	Output dimensions.
n_neighbors	Number of nearest neighbours.
min_dist	Minimum UMAP distance.
metric	Distance metric passed to uwot::umap().
n_epochs	Optional training epochs.
seed	Optional random seed.
...	Additional arguments passed to uwot::umap().
reduced_dim	For a SingleCellExperiment, the reduced-dimension name to embed. When NULL, a compatible recorded metadata choice is used first, followed by a uniquely named "PCA". Other names must be selected explicitly.

Value

A cuda_embedding list containing coordinates, method, backend, compute_device, per-stage compute_stages, source metadata, and algorithm parameters.

Examples

```
if (requireNamespace("uwot", quietly = TRUE)) {
  cuda_umap(matrix(rnorm(120), 40, 3), n_neighbors = 5, seed = 1)
}
```

`dimnames.cudasparse` *Inspect sparse matrix dimension labels*

Description

Inspect sparse matrix dimension labels

Usage

```
## S3 method for class 'cudasparse'
dimnames(x)
```

Arguments

`x`	A `cudasparse` matrix.

Value

`NULL` for an unnamed matrix, otherwise its row and column names, following base R `dimnames()` semantics.

`dimnames.cudatensor` *Inspect tensor dimension labels*

Description

Inspect tensor dimension labels

Usage

```
## S3 method for class 'cudatensor'
dimnames(x)
```

Arguments

`x`	A `cudatensor`.

Value

`NULL` for an unnamed tensor, otherwise one character vector (or `NULL`) per tensor dimension, following base R `dimnames()` semantics.

embedding_coordinates *Extract embedding coordinates*

Description

Extract embedding coordinates

Usage

```
embedding_coordinates(x)
```

Arguments

x	A cuda_embedding.

Value

Numeric coordinate matrix.

Examples

```
fit <- cuda_diffusion_map(
  matrix(rnorm(60), 20, 3),
  n_components = 2,
  device = "cpu"
)
embedding_coordinates(fit)
```

predict.cuda_kmeans *Assign observations with a fitted CUDA-aware k-means model*

Description

`predict.cuda_kmeans()` computes Euclidean distances to the fitted centres and returns either the closest-centre assignment or the complete distance matrix. Named features may be supplied in any order and are aligned safely.

Usage

```
## S3 method for class 'cuda_kmeans'
predict(
  object,
  newdata,
  type = c("cluster", "distance"),
  device = c("model", "auto", "cuda", "cpu"),
  ...
)
```

Arguments

object	A fitted cuda_kmeans object.
newdata	A finite numeric matrix or data frame with observations in rows and model features in columns. When omitted and type = "cluster", the training assignments in object$cluster are returned.
type	Return closest-centre "cluster" assignments or the observation-by-centre "distance" matrix.
device	Device used for the distance calculation. "model" reuses the fitted model's actual distance device; "auto", "cuda", and "cpu" follow the usual cudaverse device-selection rules.
...	Must be empty.

Value

For type = "cluster", an integer vector with observation names and, for recomputed assignments, stage-level provenance. For type = "distance", a numeric matrix whose columns identify the fitted centres. Omitting newdata returns validated stored training assignments unchanged and does not create a prediction stage.

See Also

cuda_kmeans()

Examples

```
train <- as.matrix(iris[1:100, 1:4])
fit <- cuda_kmeans(train, centers = 3, seed = 1, device = "cpu")
predict(fit, as.matrix(iris[101:105, 1:4]), device = "cpu")
```

---

predict.cuda_pca *Project observations with a fitted CUDA-aware PCA model*

Description

predict.cuda_pca() applies the fitted centring, scaling, and loadings to new observations. Named features may be supplied in any order and are aligned safely before projection. If the fitted model has feature names, unnamed or mismatched columns are rejected instead of being used in the wrong order.

Usage

```
## S3 method for class 'cuda_pca'
predict(object, newdata, device = c("model", "auto", "cuda", "cpu"), ...)
```

Arguments

object	A fitted cuda_pca object.
newdata	A finite numeric matrix or data frame with observations in rows and the model features in columns. When omitted, the training scores in object$x are returned.
device	Where to compute the projection. "model" reuses the actual device of the fitted model; "auto", "cuda", and "cpu" follow the usual cudaverse device-selection rules.
...	Must be empty.

Value

A numeric matrix of component scores. New observation names and stable component names are retained. A recomputed prediction includes stage-level provenance and is materialized as an R matrix on the CPU. The native backend also retains shared device storage so a subsequent native distance or kNN operation can reuse the scores without uploading them. Omitting newdata returns the validated stored training scores unchanged; that retrieval does not create a prediction stage.

See Also

cuda_pca()

Examples

```
train <- as.matrix(iris[1:100, 1:4])
fit <- cuda_pca(train, n_components = 2, device = "cpu")
predict(fit, as.matrix(iris[101:105, 1:4]), device = "cpu")
```

---

sparse_info *Inspect sparse matrix metadata*

Description

Inspect sparse matrix metadata

Usage

```
sparse_info(x)
```

Arguments

x	A cudasparse matrix.

Value

A named list containing shape, nnz, density, format, actual device, and backend.

Examples

```
sparse_info(cuda_sparse(diag(3), device = "cpu"))
```

`sparse_matmul_dense` *Sparse matrix by dense matrix multiplication*

Description

Sparse matrix by dense matrix multiplication

Usage

```
sparse_matmul_dense(x, y)
```

Arguments

x	A cudasparse matrix.
y	A numeric matrix or cudatensor.

Value

A dense cudatensor. The native backend keeps the result on CUDA; compatibility backends retain their existing portable CPU result.

Examples

```
x <- cuda_sparse(diag(3), device = "cpu")
sparse_matmul_dense(x, matrix(1:6, 3, 2))
```

`sparse_matvec` *Sparse matrix-vector multiplication*

Description

Sparse matrix-vector multiplication

Usage

```
sparse_matvec(x, y)
```

Arguments

x	A cudasparse matrix.
y	A numeric vector.

Value

A numeric vector.

Examples

```
sparse_matvec(cuda_sparse(diag(3), device = "cpu"), 1:3)
```

`sparse_normalize` *Normalize sparse rows or columns without densifying*

Description

Each selected row or column is divided by its sum and multiplied by `scale_factor`. Optionally, `log1p()` is applied to stored non-zero values. The operation preserves sparse structure and dimension labels. The native CUDA backend retains normalized storage on the device and updates the public host COO mirror from metadata already held by the object. It does not download the normalized values or the complete margin-sum vector; only a small device-validation flag crosses back before the result is returned. Native results share immutable sparse index storage with their source while retaining independent value storage and release-safe ownership.

Usage

```
sparse_normalize(
  x,
  margin = c("rows", "columns"),
  scale_factor = 1,
  log1p = FALSE
)
```

Arguments

`x`	A non-negative `cudasparse` matrix.
`margin`	Normalize `"rows"` or `"columns"`.
`scale_factor`	Positive target sum before the optional log transform.
`log1p`	Whether to apply `log1p()` to normalized stored values.

Value

A `cudasparse` matrix on the same device as `x`.

Examples

```
x <- cuda_sparse(matrix(c(1, 0, 3, 2), 2), device = "cpu")
sparse_normalize(x, margin = "rows", scale_factor = 1)
```

sparse_row_sums *Sparse row and column reductions*

Description

Sparse row and column reductions

Usage

```
sparse_row_sums(x)

sparse_col_sums(x)
```

Arguments

x	A cudasparse matrix.

Value

A numeric vector.

Examples

```
x <- cuda_sparse(matrix(1:6, 2), device = "cpu")
sparse_row_sums(x)
sparse_col_sums(x)
```

t.cudasparse *Transpose a GPU-aware sparse matrix*

Description

Swaps sparse rows and columns while preserving stored values, logical format, dimension labels, and the actual device. The native CUDA backend transposes its CSR backing storage on the device. Compatibility backends rebuild same-device storage from the stable public COO metadata.

Usage

```
## S3 method for class 'cudasparse'
t(x)
```

Arguments

x	A cudasparse matrix.

Value

A transposed cudasparse matrix on the same device as x.

Examples

```
x <- cuda_sparse(matrix(c(1, 0, 2, 0, 3, 0), 2), device = "cpu")
t(x)
```

---

tensor_broadcast_to *Broadcast a tensor to a compatible shape*

Description

Broadcast a tensor to a compatible shape

Usage

```
tensor_broadcast_to(x, shape)
```

Arguments

x	A cudatensor.
shape	Target dimensions. Existing dimensions are aligned from the right and must either match or equal one.

Details

Labels are retained on dimensions whose sizes do not change. Labels are dropped from singleton dimensions that are expanded because a single input label cannot identify multiple output positions.

Value

A cudatensor.

Examples

```
x <- cuda_tensor(1:3, device = "cpu")
tensor_broadcast_to(x, c(2, 3))
```

tensor_device *Inspect tensor device and backend*

Description

Inspect tensor device and backend

Usage

```
tensor_device(x)
```

Arguments

x	A cudatensor.

Value

A named character vector.

Examples

```
tensor_device(cuda_tensor(1:3, device = "cpu"))
```

tensor_matmul *Matrix multiplication for tensors*

Description

Matrix multiplication for tensors

Usage

```
tensor_matmul(x, y)
```

Arguments

x, y	Two-dimensional cudatensor objects or numeric matrices.

Details

Row names come from x and column names come from y. When both operands name the contracted dimension, those names must be identical.

Value

A cudatensor.

Examples

```
x <- cuda_tensor(matrix(1:6, 2, 3), device = "cpu")
y <- cuda_tensor(matrix(1:6, 3, 2), device = "cpu")
tensor_matmul(x, y)
```

`tensor_reshape` *Reshape a tensor without changing its values*

Description

Reshape a tensor without changing its values

Usage

```
tensor_reshape(x, shape)
```

Arguments

`x`	A `cudatensor`.
`shape`	Positive whole-number dimensions whose product equals `length(x)`.

Details

The native CUDA backend creates an allocation-free metadata view that shares the source device allocation. The source and reshaped tensor have independent external-pointer lifetimes, and the allocation is freed only after the final view is released. Compatibility backends retain their established reshape behavior.

Value

A `cudatensor` on the same device with the requested shape.

Examples

```
x <- cuda_tensor(1:6, device = "cpu")
tensor_reshape(x, c(2, 3))
```

tensor_shape *Inspect tensor shape*

Description

Inspect tensor shape

Usage

```
tensor_shape(x)
```

Arguments

x	A cudatensor.

Value

An integer vector.

Examples

```
tensor_shape(cuda_tensor(matrix(1:6, 2), device = "cpu"))
```

tensor_sum *Tensor reductions*

Description

Tensor reductions

Usage

```
tensor_sum(x, dim = NULL, keepdim = FALSE)

tensor_mean(x, dim = NULL, keepdim = FALSE)
```

Arguments

x	A cudatensor.
dim	Optional one-based dimensions to reduce.
keepdim	Whether reduced dimensions should be retained with size one.

Details

Labels on dimensions that are not reduced are retained. A reduced dimension kept with size one retains its axis name but not its individual labels. Supplying `integer(0)` performs no reduction and returns the tensor values, shape, device, and dimnames unchanged (with the documented reduction dtype promotion).

Value

A `cudatensor`.

Examples

```
x <- cuda_tensor(matrix(1:6, 2), device = "cpu")
tensor_sum(x)
tensor_mean(x, dim = 1)
```

`to_cpu` *Transfer tensor data to base R*

Description

Transfer tensor data to base R

Usage

```
to_cpu(x)
```

Arguments

`x`	A `cudatensor`.

Value

A base R vector, matrix, or array with the tensor shape.

Examples

```
to_cpu(cuda_tensor(matrix(1:4, 2), device = "cpu"))
```

to_device *Transfer a tensor to a device*

Description

Transfer a tensor to a device

Usage

```
to_device(x, device = c("cpu", "cuda"))
```

Arguments

x	A cudatensor.
device	"cpu" or "cuda".

Value

A cudatensor on the requested device.

Examples

```
x <- cuda_tensor(1:4, device = "cpu")
to_device(x, "cpu")
```

to_dgCMatrix *Convert to an R sparse matrix*

Description

Convert to an R sparse matrix

Usage

```
to_dgCMatrix(x)
```

Arguments

x	A cudasparse matrix.

Value

A Matrix::dgCMatrix.

Examples

```
to_dgCMatrix(cuda_sparse(diag(3), device = "cpu"))
```

Index