Package ‘ggmlR’

January 28, 2026

Type Package

Title GGML Tensor Operations for Machine Learning

Version 0.4.1

Description An R port of the 'GGML' library providing efficient tensor operations for machine learning. Implements core functionality including element-wise arithmetic, tensor manipulation, and multi-backend support. Brings lightweight, performance-oriented design to 'R' for fast numerical computation and model prototyping. See <https://github.com/ggml-org/ggml> for more information about the underlying library.

License MIT + file LICENSE

URL https://github.com/Zabis13/ggmlR

BugReports https://github.com/Zabis13/ggmlR/issues

Encoding UTF-8

SystemRequirements C++17, GNU make

Suggests testthat (>= 3.0.0)

RoxygenNote 7.3.3

Config/testthat/edition 3

NeedsCompilation yes

Author Yuri Baramykov [aut, cre]

Maintainer Yuri Baramykov <lbsbmsu@mail.ru>

R topics documented:

`ggmlR-package` *ggmlR: GGML Tensor Operations for Machine Learning*

Description

An R port of the 'GGML' library providing efficient tensor operations for machine learning. Implements core functionality including element-wise arithmetic, tensor manipulation, and multi-backend support. Brings lightweight, performance-oriented design to 'R' for fast numerical computation and model prototyping. See `https://github.com/ggml-org/ggml` for more information about the underlying library.

Author(s)

Maintainer: Yuri Baramykov `<lbsbmsu@mail.ru>`

See Also

Useful links:

- `https://github.com/Zabis13/ggmlR`
- Report bugs at `https://github.com/Zabis13/ggmlR/issues`

ggml_abs — *Absolute Value (Graph)*

Description

Creates a graph node for element-wise absolute value: |x|

Usage

```
ggml_abs(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the abs operation

ggml_add — *Add tensors*

Description

Creates a graph node for element-wise addition. Must be computed using ggml_build_forward_expand() and ggml_graph_compute().

Usage

```
ggml_add(ctx, a, b)
```



```
ggml_add(ctx, a, b)
```


Arguments

ctx	GGML context
a	First tensor
b	Second tensor (same shape as a)

Value

Tensor representing the addition operation

Tensor representing the addition operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(5, 4, 3, 2, 1))
result <- ggml_add(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(5, 4, 3, 2, 1))
result <- ggml_add(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

---

ggml_add1 *Add Scalar to Tensor (Graph)*

Description

Creates a graph node for adding a scalar (1-element tensor) to all elements of a tensor. This is more efficient than creating a full tensor of the same value.

Usage

```
ggml_add1(ctx, a, b)
```

Arguments

ctx	GGML context
a	Input tensor
b	Scalar tensor (1-element tensor)

Value

Tensor representing the operation a + b (broadcasted)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
scalar <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(scalar, 10)
result <- ggml_add1(ctx, a, scalar)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

---

ggml_are_same_shape   *Compare Tensor Shapes*

Description

Checks if two tensors have the same shape.

Usage

```
ggml_are_same_shape(a, b)
```

Arguments

a	First tensor
b	Second tensor

Value

TRUE if shapes are identical, FALSE otherwise

---

ggml_argmax   *Argmax (Graph)*

Description

Creates a graph node that finds the index of the maximum value. CRITICAL for token generation in LLMs.

Usage

```
ggml_argmax(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor with argmax indices

`ggml_argsort` *Argsort - Get Sorting Indices (Graph)*

Description

Returns indices that would sort the tensor rows. Each row is sorted independently.

Usage

```
ggml_argsort(ctx, a, order = GGML_SORT_ORDER_ASC)
```

Arguments

`ctx`	GGML context
`a`	Input tensor to sort (F32)
`order`	Sort order: GGML_SORT_ORDER_ASC (0) or GGML_SORT_ORDER_DESC (1)

Value

Tensor of I32 indices that would sort each row

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Create tensor with values to sort
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(3, 1, 4, 1, 5))
# Get indices for ascending sort
indices <- ggml_argsort(ctx, a, GGML_SORT_ORDER_ASC)
graph <- ggml_build_forward_expand(ctx, indices)
ggml_graph_compute(ctx, graph)
result <- ggml_get_i32(indices)
# result: [1, 3, 0, 2, 4] (0-indexed positions for sorted order)
ggml_free(ctx)

## End(Not run)
```

`ggml_backend_alloc_ctx_tensors` *Allocate Context Tensors to Backend*

Description

Allocates all tensors in a GGML context to a specific backend. Returns a buffer that must be freed when no longer needed.

Usage

```
ggml_backend_alloc_ctx_tensors(ctx, backend)
```

Arguments

`ctx`	GGML context
`backend`	Backend handle

Value

Backend buffer object

`ggml_backend_buffer_free` *Free Backend Buffer*

Description

Frees a backend buffer and all associated memory.

Usage

```
ggml_backend_buffer_free(buffer)
```

Arguments

`buffer`	Backend buffer object

ggml_backend_buffer_get_size

Get Backend Buffer Size

Description

Returns the total size of a backend buffer.

Usage

```
ggml_backend_buffer_get_size(buffer)
```

Arguments

buffer	Backend buffer object

Value

Size in bytes

ggml_backend_buffer_name

Get Backend Buffer Name

Description

Returns the name/type of a backend buffer.

Usage

```
ggml_backend_buffer_name(buffer)
```

Arguments

buffer	Backend buffer object

Value

Character string with buffer name

`ggml_backend_cpu_init` *Initialize CPU Backend*

Description

Creates a new CPU backend instance for graph computation.

Usage

```
ggml_backend_cpu_init()
```

Value

Backend pointer

`ggml_backend_cpu_set_n_threads` *Set CPU Backend Threads*

Description

Sets the number of threads for CPU backend computation.

Usage

```
ggml_backend_cpu_set_n_threads(backend, n_threads)
```

Arguments

`backend`	CPU backend pointer
`n_threads`	Number of threads

Value

NULL invisibly

ggml_backend_free *Free Backend*

Description

Releases resources associated with a backend.

Usage

```
ggml_backend_free(backend)
```

Arguments

backend	Backend pointer

Value

NULL invisibly

ggml_backend_graph_compute *Compute Graph with Backend*

Description

Executes computation graph using specified backend.

Usage

```
ggml_backend_graph_compute(backend, graph)
```

Arguments

backend	Backend pointer
graph	Graph pointer

Value

Status code (0 = success)

`ggml_backend_name` *Get Backend Name*

Description

Returns the name of the backend (e.g., "CPU").

Usage

```
ggml_backend_name(backend)
```

Arguments

`backend`	Backend pointer

Value

Character string name

`ggml_backend_sched_alloc_graph` *Allocate graph on scheduler*

Description

Allocates memory for a graph across the scheduler's backends. Must be called before computing the graph.

Usage

```
ggml_backend_sched_alloc_graph(sched, graph)
```

Arguments

`sched`	Scheduler pointer
`graph`	Graph pointer

Value

Logical indicating success

ggml_backend_sched_free

Free backend scheduler

Description

Releases resources associated with the backend scheduler.

Usage

```
ggml_backend_sched_free(sched)
```

Arguments

sched	Scheduler pointer from ggml_backend_sched_new()

Value

NULL (invisible)

Examples

```
## Not run:
sched <- ggml_backend_sched_new(list(gpu_backend))
ggml_backend_sched_free(sched)

## End(Not run)
```

ggml_backend_sched_get_backend

Get backend from scheduler

Description

Returns a specific backend from the scheduler by index.

Usage

```
ggml_backend_sched_get_backend(sched, index = 0L)
```

Arguments

sched	Scheduler pointer
index	Backend index (0-based)

Value

Backend pointer

ggml_backend_sched_get_n_backends

Get number of backends in scheduler

Description

Returns the number of backends managed by the scheduler.

Usage

```
ggml_backend_sched_get_n_backends(sched)
```

Arguments

sched	Scheduler pointer

Value

Integer count of backends

ggml_backend_sched_get_n_copies

Get number of tensor copies

Description

Returns the number of tensor copies made in the last computed graph. Copies occur when data needs to be transferred between backends.

Usage

```
ggml_backend_sched_get_n_copies(sched)
```

Arguments

sched	Scheduler pointer

Value

Integer count of copies

`ggml_backend_sched_get_n_splits`
Get number of graph splits

Description

Returns the number of splits in the last computed graph. Higher numbers indicate more distribution across backends.

Usage

```
ggml_backend_sched_get_n_splits(sched)
```

Arguments

`sched`	Scheduler pointer

Value

Integer count of splits

`ggml_backend_sched_get_tensor_backend`
Get tensor backend assignment

Description

Returns which backend a tensor is assigned to.

Usage

```
ggml_backend_sched_get_tensor_backend(sched, tensor)
```

Arguments

`sched`	Scheduler pointer
`tensor`	Tensor pointer

Value

Backend pointer or NULL if not assigned

`ggml_backend_sched_graph_compute`
Compute graph using scheduler

Description

Computes a graph by distributing work across multiple backends. This is the main function for multi-GPU computation.

Usage

```
ggml_backend_sched_graph_compute(sched, graph)
```

Arguments

sched	Scheduler pointer
graph	Graph pointer

Value

Status code (0 = success)

Examples

```
## Not run:
# Multi-GPU example
if (ggml_vulkan_available() && ggml_vulkan_device_count() >= 2) {
  gpu1 <- ggml_vulkan_init(0)
  gpu2 <- ggml_vulkan_init(1)
  sched <- ggml_backend_sched_new(list(gpu1, gpu2))

  ctx <- ggml_init(64 * 1024 * 1024)
  a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10000)
  b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10000)
  ggml_set_f32(a, rnorm(10000))
  ggml_set_f32(b, rnorm(10000))

  c <- ggml_add(ctx, a, b)
  graph <- ggml_build_forward_expand(ctx, c)

  # Reserve memory
  ggml_backend_sched_reserve(sched, graph)

  # Compute using both GPUs
  ggml_backend_sched_graph_compute(sched, graph)

  result <- ggml_get_f32(c)

  cat("Splits:", ggml_backend_sched_get_n_splits(sched), "\n")
```

```
  cat("Copies:", ggml_backend_sched_get_n_copies(sched), "\n")

  ggml_free(ctx)
  ggml_backend_sched_free(sched)
  ggml_vulkan_free(gpu1)
  ggml_vulkan_free(gpu2)
}

## End(Not run)
```

---

ggml_backend_sched_graph_compute_async

Compute graph asynchronously

Description

Computes a graph asynchronously across backends. Use ggml_backend_sched_synchronize() to wait for completion.

Usage

```
ggml_backend_sched_graph_compute_async(sched, graph)
```

Arguments

sched	Scheduler pointer
graph	Graph pointer

Value

Status code (0 = success)

---

ggml_backend_sched_new

Create a new backend scheduler

Description

Creates a scheduler that can distribute computation across multiple backends (GPUs, CPU). A CPU backend is automatically added as a fallback. Backends with lower index have higher priority.

Usage

```
ggml_backend_sched_new(backends, parallel = TRUE, graph_size = 2048)
```

Arguments

`backends`	List of backend pointers (from ggml_vulkan_init() or ggml_backend_cpu_init()). Note: A CPU backend is automatically added, so you only need to specify GPU backends.
`parallel`	Logical, whether to run backends in parallel (default: TRUE)
`graph_size`	Expected maximum graph size (default: 2048)

Value

Scheduler pointer

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() >= 2) {
  # Create two GPU backends (CPU is added automatically)
  gpu1 <- ggml_vulkan_init(0)
  gpu2 <- ggml_vulkan_init(1)

  # Create scheduler with both GPUs + CPU (automatic)
  sched <- ggml_backend_sched_new(list(gpu1, gpu2), parallel = TRUE)

  # The scheduler now has 3 backends: GPU1, GPU2, CPU
  cat("Backends:", ggml_backend_sched_get_n_backends(sched), "\\n")

  # Use scheduler...

  # Cleanup
  ggml_backend_sched_free(sched)
  ggml_vulkan_free(gpu1)
  ggml_vulkan_free(gpu2)
}

## End(Not run)
```

`ggml_backend_sched_reserve` *Reserve memory for scheduler*

Description

Pre-allocates memory based on a measurement graph. This should be called before using the scheduler to compute graphs.

Usage

```
ggml_backend_sched_reserve(sched, graph)
```

Arguments

sched	Scheduler pointer
graph	Graph pointer to measure memory requirements

Value

Logical indicating success

Examples

```
## Not run:
sched <- ggml_backend_sched_new(list(gpu_backend))
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1000)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 1000)
c <- ggml_add(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, c)

# Reserve memory based on this graph
ggml_backend_sched_reserve(sched, graph)

## End(Not run)
```

---

ggml_backend_sched_reset

Reset scheduler

Description

Resets the scheduler, deallocating all tensors. Must be called before changing node backends or allocating a new graph.

Usage

```
ggml_backend_sched_reset(sched)
```

Arguments

sched	Scheduler pointer

Value

NULL (invisible)

`ggml_backend_sched_set_tensor_backend`
Set tensor backend assignment

Description

Manually assigns a specific tensor to run on a specific backend. This overrides automatic scheduling.

Usage

```
ggml_backend_sched_set_tensor_backend(sched, tensor, backend)
```

Arguments

`sched`	Scheduler pointer
`tensor`	Tensor pointer
`backend`	Backend pointer to assign tensor to

Value

NULL (invisible)

`ggml_backend_sched_synchronize`
Synchronize scheduler

Description

Waits for all asynchronous operations to complete.

Usage

```
ggml_backend_sched_synchronize(sched)
```

Arguments

`sched`	Scheduler pointer

Value

NULL (invisible)

ggml_backend_tensor_get_data

Get Tensor Data via Backend

Description

Gets tensor data using the backend API. This works with tensors allocated on any backend, not just CPU.

Usage

```
ggml_backend_tensor_get_data(tensor, offset = 0, n_elements = NULL)
```

Arguments

`tensor`	Tensor pointer
`offset`	Byte offset (default: 0)
`n_elements`	Number of elements to retrieve (NULL for all)

Value

R vector with tensor data

ggml_backend_tensor_set_data

Set Tensor Data via Backend

Description

Sets tensor data using the backend API. This works with tensors allocated on any backend, not just CPU.

Usage

```
ggml_backend_tensor_set_data(tensor, data, offset = 0)
```

Arguments

`tensor`	Tensor pointer
`data`	R vector with data to set
`offset`	Byte offset (default: 0)

ggml_build_forward_expand
Build forward expand

Description

Builds a computation graph from the output tensor, expanding backwards to include all dependencies.

Creates a computation graph by expanding backwards from the output tensor

Usage

```
ggml_build_forward_expand(ctx, tensor)

ggml_build_forward_expand(ctx, tensor)
```

Arguments

ctx	GGML context
tensor	Output tensor of the computation

Value

Graph pointer

Graph object (external pointer)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(5, 4, 3, 2, 1))
result <- ggml_add(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_set_f32(a, 1:10)
ggml_set_f32(b, 11:20)
c <- ggml_add(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, c)
```

```
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(c)
ggml_free(ctx)

## End(Not run)
```

`ggml_ceil` *Ceiling (Graph)*

Description

Creates a graph node for element-wise ceiling: ceil(x)

Usage

```
ggml_ceil(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the ceil operation

`ggml_clamp` *Clamp (Graph)*

Description

Creates a graph node for clamping values to a range: clamp(x, min, max)

Usage

```
ggml_clamp(ctx, a, min_val, max_val)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`min_val`	Minimum value
`max_val`	Maximum value

Value

Tensor with values clamped to [min_val, max_val] `ggml_concat` *Concatenate Tensors (Graph)*

Description

Concatenates two tensors along a specified dimension. CRITICAL for KV-cache operations in transformers.

Usage

```
ggml_concat(ctx, a, b, dim = 0)
```

Arguments

`ctx`	GGML context
`a`	First tensor
`b`	Second tensor (must match a in all dimensions except the concat dim)
`dim`	Dimension along which to concatenate (0-3)

Value

Concatenated tensor

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 3)
b <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 2)
ggml_set_f32(a, rnorm(12))
ggml_set_f32(b, rnorm(8))
# Concatenate along dimension 1: result is 4x5
c <- ggml_concat(ctx, a, b, 1)
graph <- ggml_build_forward_expand(ctx, c)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

ggml_cont — *Make Contiguous (Graph)*

Description

Makes a tensor contiguous in memory. Required after permute/transpose before some operations.

Usage

```
ggml_cont(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Contiguous tensor

ggml_conv_1d — *1D Convolution (Graph)*

Description

Applies 1D convolution to input data.

Usage

```
ggml_conv_1d(ctx, a, b, s0 = 1L, p0 = 0L, d0 = 1L)
```

Arguments

ctx	GGML context
a	Convolution kernel tensor
b	Input data tensor
s0	Stride (default 1)
p0	Padding (default 0)
d0	Dilation (default 1)

Value

Convolved tensor

ggml_conv_2d *2D Convolution (Graph)*

Description

Applies 2D convolution to input data.

Usage

```
ggml_conv_2d(ctx, a, b, s0 = 1L, s1 = 1L, p0 = 0L, p1 = 0L, d0 = 1L, d1 = 1L)
```

Arguments

ctx	GGML context
a	Convolution kernel tensor [KW, KH, IC, OC]
b	Input data tensor [W, H, C, N]
s0	Stride dimension 0 (default 1)
s1	Stride dimension 1 (default 1)
p0	Padding dimension 0 (default 0)
p1	Padding dimension 1 (default 0)
d0	Dilation dimension 0 (default 1)
d1	Dilation dimension 1 (default 1)

Value

Convolved tensor

ggml_conv_transpose_1d *Transposed 1D Convolution (Graph)*

Description

Applies transposed 1D convolution (deconvolution) to input data.

Usage

```
ggml_conv_transpose_1d(ctx, a, b, s0 = 1L, p0 = 0L, d0 = 1L)
```

Arguments

`ctx`	GGML context
`a`	Convolution kernel tensor
`b`	Input data tensor
`s0`	Stride (default 1)
`p0`	Padding (default 0)
`d0`	Dilation (default 1)

Value

Transposed convolved tensor

`ggml_cos` *Cosine (Graph)*

Description

Creates a graph node for element-wise cosine: cos(x)

Usage

```
ggml_cos(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the cos operation

ggml_cpu_add — *Element-wise Addition (CPU Direct)*

Description

Performs element-wise addition of two tensors using direct CPU computation. Returns the result as an R numeric vector. Does NOT use computation graphs.

Usage

```
ggml_cpu_add(a, b)
```

Arguments

a	First tensor (must be F32 type)
b	Second tensor (must be F32 type, same size as a)

Value

Numeric vector containing the element-wise sum

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(5, 4, 3, 2, 1))
ggml_cpu_add(a, b)
ggml_free(ctx)

## End(Not run)
```

ggml_cpu_mul — *Element-wise Multiplication (CPU Direct)*

Description

Performs element-wise multiplication of two tensors using direct CPU computation. Returns the result as an R numeric vector. Does NOT use computation graphs.

Usage

```
ggml_cpu_mul(a, b)
```

Arguments

a	First tensor (must be F32 type)
b	Second tensor (must be F32 type, same size as a)

Value

Numeric vector containing the element-wise product

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(2, 2, 2, 2, 2))
ggml_cpu_mul(a, b)
ggml_free(ctx)

## End(Not run)
```

---

ggml_cpy *Copy Tensor with Type Conversion (Graph)*

Description

Copies tensor a into tensor b, performing type conversion if needed. The tensors must have the same number of elements. CRITICAL for type casting operations (e.g., F32 to F16).

Usage

```
ggml_cpy(ctx, a, b)
```

Arguments

ctx	GGML context
a	Source tensor
b	Destination tensor (defines output type and shape)

Value

Tensor representing the copy operation (returns b with a's data)

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Create F32 tensor
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 100)
ggml_set_f32(a, rnorm(100))
# Create F16 tensor for output
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F16, 100)
# Copy with F32 -> F16 conversion
result <- ggml_cpy(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

---

ggml_cycles *Get CPU Cycles*

Description

Returns the current CPU cycle count. Useful for low-level benchmarking.

Usage

```
ggml_cycles()
```

Value

Numeric value representing CPU cycles

Examples

```
## Not run:
ggml_cycles()

## End(Not run)
```

`ggml_cycles_per_ms` *Get CPU Cycles per Millisecond*

Description

Returns an estimate of CPU cycles per millisecond. Useful for converting cycle counts to time.

Usage

```
ggml_cycles_per_ms()
```

Value

Numeric value representing cycles per millisecond

Examples

```
## Not run:
ggml_cycles_per_ms()

## End(Not run)
```

`ggml_diag` *Diagonal Matrix (Graph)*

Description

Creates a diagonal matrix from a vector. For vector a[n], produces matrix with a on the diagonal.

Usage

```
ggml_diag(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input vector tensor

Value

Diagonal matrix tensor

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 3)
ggml_set_f32(a, c(1, 2, 3))
d <- ggml_diag(ctx, a)  # 3x3 diagonal matrix
graph <- ggml_build_forward_expand(ctx, d)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

---

ggml_diag_mask_inf *Diagonal Mask with -Inf (Graph)*

Description

Creates a graph node that sets elements above the diagonal to -Inf. This is used for causal (autoregressive) attention masking.

Usage

```
ggml_diag_mask_inf(ctx, a, n_past)
```

Arguments

ctx	GGML context
a	Input tensor (typically attention scores)
n_past	Number of past tokens (shifts the diagonal). Use 0 for standard causal masking where position i can only attend to positions <= i.

Details

In causal attention, we want each position to only attend to itself and previous positions. Setting future positions to -Inf ensures that after softmax, they contribute 0 attention weight.

The n_past parameter allows for KV-cache scenarios where the diagonal needs to be shifted to account for previously processed tokens.

Value

Tensor with same shape as input, elements above diagonal set to -Inf

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Create attention scores matrix
scores <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 4)
ggml_set_f32(scores, rep(1, 16))
# Apply causal mask
masked <- ggml_diag_mask_inf(ctx, scores, 0)
graph <- ggml_build_forward_expand(ctx, masked)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

---

ggml_diag_mask_inf_inplace

Diagonal Mask with -Inf In-place (Graph)

Description

In-place version of ggml_diag_mask_inf. Returns a view of the input tensor.

Usage

```
ggml_diag_mask_inf_inplace(ctx, a, n_past)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (will be modified in-place)
`n_past`	Number of past tokens

Value

View of input tensor with elements above diagonal set to -Inf

ggml_diag_mask_zero — *Diagonal Mask with Zero (Graph)*

Description

Creates a graph node that sets elements above the diagonal to 0. Alternative to -Inf masking for certain use cases.

Usage

```
ggml_diag_mask_zero(ctx, a, n_past)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`n_past`	Number of past tokens

Value

Tensor with same shape as input, elements above diagonal set to 0

ggml_div — *Element-wise Division (Graph)*

Description

Creates a graph node for element-wise division.

Usage

```
ggml_div(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	First tensor (numerator)
`b`	Second tensor (denominator, same shape as a)

Value

Tensor representing the division operation (a / b)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(10, 20, 30, 40, 50))
ggml_set_f32(b, c(2, 2, 2, 2, 2))
result <- ggml_div(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

---

ggml_dup *Duplicate Tensor (Graph)*

Description

Creates a graph node that copies a tensor. This is a graph operation that must be computed using ggml_build_forward_expand() and ggml_graph_compute(). Unlike ggml_dup_tensor which just allocates, this creates a copy operation in the graph.

Usage

```
ggml_dup(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the copy operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
b <- ggml_dup(ctx, a)
graph <- ggml_build_forward_expand(ctx, b)
ggml_graph_compute(ctx, graph)
ggml_get_f32(b)
ggml_free(ctx)

## End(Not run)
```

ggml_dup_tensor *Duplicate Tensor*

Description

Creates a copy of a tensor with the same shape and type

Usage

```
ggml_dup_tensor(ctx, tensor)
```

Arguments

ctx	GGML context
tensor	Tensor to duplicate

Value

New tensor pointer with same shape

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 100)
b <- ggml_dup_tensor(ctx, a)
ggml_nelements(b)
ggml_free(ctx)

## End(Not run)
```

ggml_element_size *Get Element Size*

Description

Returns the size of a single element in the tensor.

Usage

```
ggml_element_size(tensor)
```

Arguments

tensor	Tensor pointer

Value

Element size in bytes

`ggml_elu` *ELU Activation (Graph)*

Description

Creates a graph node for ELU (Exponential Linear Unit) activation. ELU(x) = x if x > 0, else alpha * (exp(x) - 1) where alpha = 1.

Usage

```
ggml_elu(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the ELU operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
r <- ggml_elu(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)
ggml_free(ctx)

## End(Not run)
```

`ggml_estimate_memory` *Estimate Required Memory*

Description

Helper function to estimate memory needed for a tensor

Usage

```
ggml_estimate_memory(type = GGML_TYPE_F32, ne0, ne1 = 1, ne2 = 1, ne3 = 1)
```

Arguments

`type`	Tensor type (GGML_TYPE_F32, etc)
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1 (optional)
`ne2`	Size of dimension 2 (optional)
`ne3`	Size of dimension 3 (optional)

Value

Estimated memory in bytes

Examples

```
## Not run:
# For 1000x1000 F32 matrix
ggml_estimate_memory(GGML_TYPE_F32, 1000, 1000)

## End(Not run)
```

`ggml_exp` *Exponential (Graph)*

Description

Creates a graph node for element-wise exponential: exp(x)

Usage

```
ggml_exp(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the exp operation

`ggml_flash_attn_back` *Flash Attention Backward (Graph)*

Description

Backward pass for Flash Attention. Used during training to compute gradients through attention.

Usage

```
ggml_flash_attn_back(ctx, q, k, v, d, masked = TRUE)
```

Arguments

`ctx`	GGML context
`q`	Query tensor (same as forward pass)
`k`	Key tensor (same as forward pass)
`v`	Value tensor (same as forward pass)
`d`	Gradient tensor from upstream (same shape as forward output)
`masked`	Logical: whether causal masking was used in forward pass

Value

Gradient tensor

`ggml_flash_attn_ext` *Flash Attention (Graph)*

Description

Creates a graph node for Flash Attention computation. This is a memory-efficient implementation of scaled dot-product attention.

Usage

```
ggml_flash_attn_ext(
  ctx,
  q,
  k,
  v,
  mask = NULL,
  scale,
  max_bias = 0,
  logit_softcap = 0
)
```

Arguments

`ctx`	GGML context
`q`	Query tensor of shape [head_dim, n_head, n_tokens, batch]
`k`	Key tensor of shape [head_dim, n_head_kv, n_kv, batch]
`v`	Value tensor of shape [head_dim, n_head_kv, n_kv, batch]
`mask`	Optional attention mask tensor (NULL for no mask). For causal attention, use ggml_diag_mask_inf instead.
`scale`	Attention scale factor, typically 1/sqrt(head_dim)
`max_bias`	Maximum ALiBi bias (0.0 to disable ALiBi)
`logit_softcap`	Logit soft-capping value (0.0 to disable). Used by some models like Gemma 2.

Details

Flash Attention computes: softmax(Q * K^T / scale + mask) * V

Key features: - Memory efficient: O(n) instead of O(n^2) memory for attention matrix - Supports grouped-query attention (GQA) when n_head_kv < n_head - Supports multi-query attention (MQA) when n_head_kv = 1 - Optional ALiBi (Attention with Linear Biases) for position encoding - Optional logit soft-capping for numerical stability

Value

Attention output tensor of shape [head_dim, n_head, n_tokens, batch]

Examples

```
## Not run:
ctx <- ggml_init(64 * 1024 * 1024)
head_dim <- 64
n_head <- 8
n_head_kv <- 2  # GQA with 4:1 ratio
seq_len <- 32
q <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, head_dim, n_head, seq_len, 1)
k <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, head_dim, n_head_kv, seq_len, 1)
v <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, head_dim, n_head_kv, seq_len, 1)
ggml_set_f32(q, rnorm(head_dim * n_head * seq_len))
ggml_set_f32(k, rnorm(head_dim * n_head_kv * seq_len))
ggml_set_f32(v, rnorm(head_dim * n_head_kv * seq_len))
# Scale = 1/sqrt(head_dim)
scale <- 1.0 / sqrt(head_dim)
# Compute attention
out <- ggml_flash_attn_ext(ctx, q, k, v, NULL, scale, 0.0, 0.0)
graph <- ggml_build_forward_expand(ctx, out)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

ggml_floor *Floor (Graph)*

Description

Creates a graph node for element-wise floor: floor(x)

Usage

```
ggml_floor(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the floor operation

ggml_free *Free GGML context*

Description

Free GGML context

Usage

```
ggml_free(ctx)
```

Arguments

ctx	Context pointer

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_free(ctx)

## End(Not run)
```

ggml_gallocr_alloc_graph

Allocate Memory for Graph

Description

Allocates memory for all tensors in the computation graph. This must be called before computing the graph.

Usage

```
ggml_gallocr_alloc_graph(galloc, graph)
```

Arguments

galloc	Graph allocator object
graph	Graph object

Value

TRUE on success, FALSE on failure

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
galloc <- ggml_gallocr_new()

# Create graph
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
b <- ggml_relu(ctx, a)
graph <- ggml_build_forward_expand(ctx, b)

# Allocate and compute
ggml_gallocr_alloc_graph(galloc, graph)
ggml_graph_compute(ctx, graph)

ggml_gallocr_free(galloc)
ggml_free(ctx)

## End(Not run)
```

ggml_gallocr_free *Free Graph Allocator*

Description

Frees a graph allocator and all associated buffers.

Usage

```
ggml_gallocr_free(galloc)
```

Arguments

galloc	Graph allocator object

ggml_gallocr_get_buffer_size *Get Graph Allocator Buffer Size*

Description

Returns the size of the buffer used by the graph allocator.

Usage

```
ggml_gallocr_get_buffer_size(galloc, buffer_id = 0L)
```

Arguments

galloc	Graph allocator object
buffer_id	Buffer ID (default: 0 for single-buffer allocator)

Value

Size in bytes

`ggml_gallocr_new` *Create Graph Allocator*

Description

Creates a new graph allocator for efficient memory management. The allocator can automatically allocate and reuse memory for graph tensors.

Usage

```
ggml_gallocr_new()
```

Value

Graph allocator object (external pointer)

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
galloc <- ggml_gallocr_new()

a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
b <- ggml_relu(ctx, a)
graph <- ggml_build_forward_expand(ctx, b)

# Allocate graph
ggml_gallocr_alloc_graph(galloc, graph)

ggml_gallocr_free(galloc)
ggml_free(ctx)

## End(Not run)
```

`ggml_gallocr_reserve` *Reserve Memory for Graph*

Description

Pre-allocates memory for a graph. This is optional but recommended when running the same graph multiple times to avoid reallocation.

Usage

```
ggml_gallocr_reserve(galloc, graph)
```

Arguments

galloc	Graph allocator object
graph	Graph object

Value

TRUE on success, FALSE on failure

---

ggml_geglu *GeGLU (GELU Gated Linear Unit) (Graph)*

Description

Creates a graph node for GeGLU operation. GeGLU uses GELU as the activation function on the first half. CRITICAL for models like GPT-NeoX and Falcon.

Usage

```
ggml_geglu(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor (first dimension must be even)

Details

Formula: output = GELU(x) * gate

Value

Tensor with half the first dimension of input

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 8, 3)
ggml_set_f32(a, rnorm(24))
r <- ggml_geglu(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # Shape: 4x3
ggml_free(ctx)

## End(Not run)
```

`ggml_geglu_quick` *GeGLU Quick (Fast GeGLU) (Graph)*

Description

Creates a graph node for fast GeGLU approximation. Uses faster but less accurate GELU approximation for gating.

Usage

```
ggml_geglu_quick(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (first dimension must be even)

Value

Tensor with half the first dimension of input

`ggml_geglu_split` *GeGLU Split (Graph)*

Description

Creates a graph node for GeGLU with separate input and gate tensors.

Usage

```
ggml_geglu_split(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (the values to be gated)
`b`	Gate tensor (same shape as a)

Details

Formula: output = GELU(a) * b

Value

Tensor with same shape as input tensors

ggml_gelu *GELU Activation (Graph)*

Description

Creates a graph node for GELU (Gaussian Error Linear Unit) activation. CRITICAL for GPT models.

Usage

```
ggml_gelu(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the GELU operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
result <- ggml_gelu(ctx, a)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

ggml_gelu_erf *Exact GELU Activation (Graph)*

Description

Creates a graph node for exact GELU using the error function (erf). GELU(x) = x * 0.5 * (1 + erf(x / sqrt(2))). More accurate than approximate GELU but potentially slower on some backends.

Usage

```
ggml_gelu_erf(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the exact GELU operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
r <- ggml_gelu_erf(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)
ggml_free(ctx)

## End(Not run)
```

`ggml_gelu_quick` *GELU Quick Activation (Graph)*

Description

Creates a graph node for fast approximation of GELU. Faster than standard GELU with minimal accuracy loss.

Usage

```
ggml_gelu_quick(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the GELU quick operation `ggml_get_f32` *Get F32 data*

Description

Get F32 data

Get F32 Data

Usage

```
ggml_get_f32(tensor)

ggml_get_f32(tensor)
```

Arguments

`tensor`	Tensor

Value

Numeric vector with tensor values

Numeric vector

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
tensor <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(tensor, c(1, 2, 3, 4, 5))
ggml_get_f32(tensor)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(t, c(1, 2, 3, 4, 5))
ggml_get_f32(t)
ggml_free(ctx)

## End(Not run)
```

ggml_get_i32 *Get I32 Data*

Description

Gets integer data from an I32 tensor (e.g., from ggml_argmax)

Usage

```
ggml_get_i32(tensor)
```

Arguments

tensor	Tensor of type GGML_TYPE_I32

Value

Integer vector

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
pos <- ggml_new_tensor_1d(ctx, GGML_TYPE_I32, 10)
ggml_set_i32(pos, 0:9)
ggml_get_i32(pos)
ggml_free(ctx)

## End(Not run)
```

ggml_get_max_tensor_size *Get Maximum Tensor Size*

Description

Returns the maximum tensor size that can be allocated in the context

Usage

```
ggml_get_max_tensor_size(ctx)
```

Arguments

ctx	GGML context

Value

Maximum tensor size in bytes

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_get_max_tensor_size(ctx)
ggml_free(ctx)

## End(Not run)
```

---

ggml_get_mem_size *Get Context Memory Size*

Description

Returns the total memory pool size of the context

Usage

```
ggml_get_mem_size(ctx)
```

Arguments

ctx	GGML context

Value

Total memory size in bytes

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_get_mem_size(ctx)
ggml_free(ctx)

## End(Not run)
```

`ggml_get_name` *Get Tensor Name*

Description

Retrieves the name of a tensor.

Usage

```
ggml_get_name(tensor)
```

Arguments

`tensor`	Tensor pointer

Value

Character string name or NULL if not set

`ggml_get_no_alloc` *Get No Allocation Mode*

Description

Check if no-allocation mode is enabled

Usage

```
ggml_get_no_alloc(ctx)
```

Arguments

`ctx`	GGML context

Value

Logical indicating if no_alloc is enabled

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_get_no_alloc(ctx)
ggml_free(ctx)

## End(Not run)
```

ggml_get_n_threads *Get Number of Threads*

Description

Get the current number of threads for GGML operations

Usage

```
ggml_get_n_threads()
```

Value

Number of threads

Examples

```
## Not run:
ggml_get_n_threads()

## End(Not run)
```

ggml_get_rows *Get Rows by Indices (Graph)*

Description

Creates a graph node that extracts rows from a tensor by index. This is commonly used for embedding lookup in LLMs.

Usage

```
ggml_get_rows(ctx, a, b)
```

Arguments

ctx	GGML context
a	Data tensor of shape [n_embd, n_rows, ...] - the embedding table
b	Index tensor (int32) of shape [n_indices] - which rows to extract

Details

This operation is fundamental for embedding lookup in transformers: given a vocabulary embedding matrix and token indices, it retrieves the corresponding embedding vectors.

Value

Tensor of shape [n_embd, n_indices, ...] containing the selected rows

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Create embedding matrix: 10 tokens, 4-dim embeddings
embeddings <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 10)
ggml_set_f32(embeddings, rnorm(40))
# Token indices to look up
indices <- ggml_new_tensor_1d(ctx, GGML_TYPE_I32, 3)
ggml_set_i32(indices, c(0L, 5L, 2L))
# Get embeddings for tokens 0, 5, 2
result <- ggml_get_rows(ctx, embeddings, indices)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

ggml_get_rows_back *Get Rows Backward (Graph)*

Description

Backward pass for ggml_get_rows operation. Accumulates gradients at the original row positions.

Usage

```
ggml_get_rows_back(ctx, a, b, c)
```

Arguments

ctx	GGML context
a	Gradient of get_rows output
b	Index tensor (same as forward pass)
c	Reference tensor defining output shape

Value

Gradient tensor for the embedding matrix ggml_glu *Generic GLU (Gated Linear Unit) (Graph)*

Description

Creates a graph node for GLU operation with specified gating type. GLU splits the input tensor in half along the first dimension, applies an activation to the first half (x), and multiplies it with the second half (gate).

Usage

```
ggml_glu(ctx, a, op, swapped = FALSE)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (first dimension must be even)
`op`	GLU operation type (GGML_GLU_OP_REGLU, GGML_GLU_OP_GEGLU, etc.)
`swapped`	If TRUE, swap x and gate halves (default FALSE)

Details

Formula: output = activation(x) * gate where x and gate are the two halves of the input tensor.

Value

Tensor with shape [n/2, ...] where n is the first dimension of input

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Create tensor with 10 columns (will be split into 5 + 5)
a <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 4)
ggml_set_f32(a, rnorm(40))
# Apply SwiGLU
r <- ggml_glu(ctx, a, GGML_GLU_OP_SWIGLU, FALSE)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # Shape: 5x4
ggml_free(ctx)

## End(Not run)
```

GGML_GLU_OP_REGLU *GLU Operation Types*

Description

Constants for GLU (Gated Linear Unit) operation types. Used with ggml_glu() and ggml_glu_split().

Usage

```
GGML_GLU_OP_REGLU

GGML_GLU_OP_GEGLU

GGML_GLU_OP_SWIGLU

GGML_GLU_OP_SWIGLU_OAI

GGML_GLU_OP_GEGLU_ERF

GGML_GLU_OP_GEGLU_QUICK
```

Format

Integer constants

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

Details

- `GGML_GLU_OP_REGLU`: ReGLU - ReLU gating
- `GGML_GLU_OP_GEGLU`: GeGLU - GELU gating (used in GPT-NeoX, etc.)
- `GGML_GLU_OP_SWIGLU`: SwiGLU - SiLU/Swish gating (used in LLaMA)
- `GGML_GLU_OP_GEGLU_QUICK`: GeGLU with fast approximation

ggml_glu_split — *Generic GLU Split (Graph)*

Description

Creates a graph node for GLU with separate input and gate tensors. Unlike standard GLU which splits a single tensor, this takes two separate tensors.

Usage

```
ggml_glu_split(ctx, a, b, op)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (the values to be gated)
`b`	Gate tensor (same shape as a)
`op`	GLU operation type (GGML_GLU_OP_REGLU, GGML_GLU_OP_GEGLU, etc.)

Value

Tensor with same shape as input tensors

ggml_graph_compute — *Compute graph*

Description

Executes all operations in the computation graph.

Executes the computation graph using CPU backend

Usage

```
ggml_graph_compute(ctx, graph)

ggml_graph_compute(ctx, graph)
```

Arguments

`ctx`	GGML context
`graph`	Graph object created by ggml_build_forward_expand

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
result <- ggml_relu(ctx, a)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_set_f32(a, 1:10)
ggml_set_f32(b, 11:20)
c <- ggml_add(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, c)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(c)
ggml_free(ctx)

## End(Not run)
```

---

ggml_graph_compute_with_ctx

Compute Graph with Context (Alternative Method)

Description

Computes the computation graph using the context-based method. This is an alternative to ggml_graph_compute() that uses ggml_graph_plan() and ggml_graph_compute() internally.

Usage

```
ggml_graph_compute_with_ctx(ctx, graph, n_threads = 0L)
```

Arguments

`ctx`	GGML context
`graph`	Graph object created by ggml_build_forward_expand
`n_threads`	Number of threads to use (0 for auto-detect, default: 0)

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_set_f32(a, 1:10)
c <- ggml_relu(ctx, a)
graph <- ggml_build_forward_expand(ctx, c)
ggml_graph_compute_with_ctx(ctx, graph)
result <- ggml_get_f32(c)
ggml_free(ctx)

## End(Not run)
```

`ggml_graph_dump_dot` *Export Graph to DOT Format*

Description

Exports the computation graph to a DOT file for visualization. The DOT file can be converted to an image using Graphviz tools.

Usage

```
ggml_graph_dump_dot(graph, leafs = NULL, filename)
```

Arguments

`graph`	Graph object
`leafs`	Optional graph with leaf tensors (NULL for none)
`filename`	Output filename (should end with .dot)

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
b <- ggml_relu(ctx, a)
graph <- ggml_build_forward_expand(ctx, b)
ggml_graph_dump_dot(graph, NULL, tempfile(fileext = ".dot"))
ggml_free(ctx)

## End(Not run)
```

`ggml_graph_get_tensor` *Get Tensor from Graph by Name*

Description

Finds a tensor in the computation graph by its name

Usage

```
ggml_graph_get_tensor(graph, name)
```

Arguments

`graph`	Graph object
`name`	Character string with tensor name

Value

Tensor pointer or NULL if not found

`ggml_graph_node` *Get Graph Node*

Description

Gets a specific node (tensor) from the computation graph by index

Usage

```
ggml_graph_node(graph, i)
```

Arguments

`graph`	Graph object
`i`	Node index (0-based, negative indices count from end)

Value

Tensor pointer

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
b <- ggml_add(ctx, a, a)
graph <- ggml_build_forward_expand(ctx, b)
# Get the last node (output)
output <- ggml_graph_node(graph, -1)
ggml_free(ctx)

## End(Not run)
```

---

ggml_graph_n_nodes *Get Number of Nodes in Graph*

Description

Returns the number of computation nodes in the graph

Usage

```
ggml_graph_n_nodes(graph)
```

Arguments

graph	Graph object

Value

Integer number of nodes

---

ggml_graph_overhead *Get Graph Overhead*

Description

Returns the memory overhead required for a computation graph

Usage

```
ggml_graph_overhead()
```

Value

Size in bytes

`ggml_graph_print` *Print Graph Information*

Description

Prints debug information about the computation graph

Usage

```
ggml_graph_print(graph)
```

Arguments

`graph`	Graph object

`ggml_graph_reset` *Reset Graph (for backpropagation)*

Description

Resets the computation graph for a new backward pass. NOTE: This function requires the graph to have gradients allocated (used for training/backpropagation). For inference-only graphs, this function will cause an error.

Usage

```
ggml_graph_reset(graph)
```

Arguments

`graph`	Graph object with gradients allocated

ggml_group_norm — *Group Normalization (Graph)*

Description

Creates a graph node for group normalization. Normalizes along ne0*ne1*n_groups dimensions. Used in Stable Diffusion and other image generation models.

Usage

```
ggml_group_norm(ctx, a, n_groups, eps = 1e-05)
```

Arguments

ctx	GGML context
a	Input tensor
n_groups	Number of groups to divide channels into
eps	Epsilon for numerical stability (default 1e-5)

Value

Tensor representing the group norm operation

ggml_group_norm_inplace — *Group Normalization In-place (Graph)*

Description

Creates a graph node for in-place group normalization.

Usage

```
ggml_group_norm_inplace(ctx, a, n_groups, eps = 1e-05)
```

Arguments

ctx	GGML context
a	Input tensor (will be modified in-place)
n_groups	Number of groups
eps	Epsilon for numerical stability (default 1e-5)

Value

View of input tensor with group norm applied

ggml_hardsigmoid — *Hard Sigmoid Activation (Graph)*

Description

Creates a graph node for Hard Sigmoid activation. HardSigmoid(x) = ReLU6(x + 3) / 6 = min(max(0, x + 3), 6) / 6. A computationally efficient approximation of the sigmoid function.

Usage

```
ggml_hardsigmoid(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the Hard Sigmoid operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-4, -1, 0, 1, 4))
r <- ggml_hardsigmoid(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # [0, 0.333, 0.5, 0.667, 1]
ggml_free(ctx)

## End(Not run)
```

ggml_hardswish — *Hard Swish Activation (Graph)*

Description

Creates a graph node for Hard Swish activation. HardSwish(x) = x * ReLU6(x + 3) / 6 = x * min(max(0, x + 3), 6) / 6. Used in MobileNetV3 and other efficient architectures.

Usage

```
ggml_hardswish(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the Hard Swish operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-4, -1, 0, 1, 4))
r <- ggml_hardswish(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)
ggml_free(ctx)

## End(Not run)
```

---

ggml_im2col *Image to Column (Graph)*

Description

Transforms image data into column format for efficient convolution. This is a low-level operation used internally by convolution implementations.

Usage

```
ggml_im2col(
  ctx,
  a,
  b,
  s0,
  s1,
  p0,
  p1,
  d0,
  d1,
  is_2D = TRUE,
  dst_type = GGML_TYPE_F16
)
```

Arguments

`ctx`	GGML context
`a`	Convolution kernel tensor
`b`	Input data tensor
`s0`	Stride dimension 0
`s1`	Stride dimension 1
`p0`	Padding dimension 0
`p1`	Padding dimension 1
`d0`	Dilation dimension 0
`d1`	Dilation dimension 1
`is_2D`	Whether this is a 2D operation (default TRUE)
`dst_type`	Output type (default GGML_TYPE_F16)

Value

Transformed tensor in column format

`ggml_init` *Initialize GGML context*

Description

Initialize GGML context

Usage

```
ggml_init(mem_size = 16 * 1024 * 1024, no_alloc = FALSE)
```

Arguments

`mem_size`	Memory size in bytes
`no_alloc`	If TRUE, don't allocate memory for tensors (default: FALSE)

Value

GGML context pointer

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_free(ctx)

## End(Not run)
```

ggml_init_auto *Create Context with Auto-sizing*

Description

Creates a context with automatically calculated size based on planned tensors

Usage

```
ggml_init_auto(..., extra_mb = 10, type = GGML_TYPE_F32, no_alloc = FALSE)
```

Arguments

`...`	Named arguments with tensor dimensions
`extra_mb`	Extra megabytes to add (default: 10)
`type`	Tensor type (default: GGML_TYPE_F32)
`no_alloc`	If TRUE, don't allocate memory for tensors (default: FALSE)

Value

GGML context

Examples

```
## Not run:
# For two 1000x1000 matrices
ctx <- ggml_init_auto(mat1 = c(1000, 1000), mat2 = c(1000, 1000))
ggml_free(ctx)

## End(Not run)
```

ggml_is_available *Check if GGML is available*

Description

Check if GGML is available

Usage

```
ggml_is_available()
```

Value

TRUE if GGML library is loaded

Examples

```
## Not run:
ggml_is_available()

## End(Not run)
```

`ggml_is_contiguous` *Check if Tensor is Contiguous*

Description

Returns TRUE if tensor data is stored contiguously in memory

Usage

```
ggml_is_contiguous(tensor)
```

Arguments

`tensor`	Tensor pointer

Value

Logical

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_is_contiguous(t)
ggml_free(ctx)

## End(Not run)
```

`ggml_is_permuted` *Check if Tensor is Permuted*

Description

Returns TRUE if tensor dimensions have been permuted

Usage

```
ggml_is_permuted(tensor)
```

Arguments

tensor Tensor pointer

Value

Logical

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_is_permuted(t)
ggml_free(ctx)

## End(Not run)
```

---

ggml_is_transposed *Check if Tensor is Transposed*

Description

Returns TRUE if tensor has been transposed

Usage

```
ggml_is_transposed(tensor)
```

Arguments

tensor Tensor pointer

Value

Logical

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_is_transposed(t)
ggml_free(ctx)

## End(Not run)
```

ggml_l2_norm *L2 Normalization (Graph)*

Description

Creates a graph node for L2 normalization (unit norm). Normalizes vectors to unit length: x / ||x||_2. Used in RWKV v7 and embedding normalization.

Usage

```
ggml_l2_norm(ctx, a, eps = 1e-05)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`eps`	Epsilon for numerical stability (default 1e-5)

Value

Tensor representing the L2 norm operation

ggml_l2_norm_inplace *L2 Normalization In-place (Graph)*

Description

Creates a graph node for in-place L2 normalization.

Usage

```
ggml_l2_norm_inplace(ctx, a, eps = 1e-05)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (will be modified in-place)
`eps`	Epsilon for numerical stability (default 1e-5)

Value

View of input tensor with L2 norm applied

ggml_leaky_relu *Leaky ReLU Activation (Graph)*

Description

Creates a graph node for Leaky ReLU activation. LeakyReLU(x) = x if x > 0, else negative_slope * x. Unlike standard ReLU, Leaky ReLU allows a small gradient for negative values.

Usage

```
ggml_leaky_relu(ctx, a, negative_slope = 0.01, inplace = FALSE)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`negative_slope`	Slope for negative values (default: 0.01)
`inplace`	If TRUE, operation is performed in-place (default: FALSE)

Value

Tensor representing the Leaky ReLU operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
r <- ggml_leaky_relu(ctx, a, negative_slope = 0.1)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # [-0.2, -0.1, 0, 1, 2]
ggml_free(ctx)

## End(Not run)
```

ggml_log *Natural Logarithm (Graph)*

Description

Creates a graph node for element-wise natural logarithm: log(x)

Usage

```
ggml_log(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the log operation

---

ggml_mean *Mean (Graph)*

Description

Creates a graph node that computes the mean of all elements.

Usage

```
ggml_mean(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Scalar tensor with the mean

---

ggml_mul *Multiply tensors*

Description

Creates a graph node for element-wise multiplication.

Usage

```
ggml_mul(ctx, a, b)

ggml_mul(ctx, a, b)
```

Arguments

ctx	GGML context
a	First tensor
b	Second tensor (same shape as a)

Value

Tensor representing the multiplication operation

Tensor representing the multiplication operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(2, 2, 2, 2, 2))
result <- ggml_mul(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(1, 2, 3, 4, 5))
ggml_set_f32(b, c(2, 2, 2, 2, 2))
result <- ggml_mul(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

---

ggml_mul_mat *Matrix Multiplication (Graph)*

Description

Creates a graph node for matrix multiplication. CRITICAL for LLM operations. For matrices A (m x n) and B (n x p), computes C = A * B (m x p).

Usage

```
ggml_mul_mat(ctx, a, b)
```

Arguments

ctx	GGML context
a	First matrix tensor
b	Second matrix tensor

Value

Tensor representing the matrix multiplication

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
A <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 3)
B <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 4, 2)
ggml_set_f32(A, 1:12)
ggml_set_f32(B, 1:8)
C <- ggml_mul_mat(ctx, A, B)
graph <- ggml_build_forward_expand(ctx, C)
ggml_graph_compute(ctx, graph)
ggml_get_f32(C)
ggml_free(ctx)

## End(Not run)
```

---

ggml_mul_mat_id *Matrix Multiplication with Expert Selection (Graph)*

Description

Indirect matrix multiplication for Mixture of Experts architectures. Selects expert weights based on indices and performs batched matmul.

Usage

```
ggml_mul_mat_id(ctx, as, b, ids)
```

Arguments

ctx	GGML context
as	Stacked expert weight matrices [n_embd, n_ff, n_experts]
b	Input tensor
ids	Expert selection indices tensor (I32)

Value

Output tensor after expert-selected matrix multiplication

Examples

```
## Not run:
ctx <- ggml_init(64 * 1024 * 1024)
# 4 experts, each with 8x16 weights (small for example)
experts <- ggml_new_tensor_3d(ctx, GGML_TYPE_F32, 8, 16, 4)
ggml_set_f32(experts, rnorm(8 * 16 * 4))
input <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 8, 2)
ggml_set_f32(input, rnorm(16))
# Select expert 0 for token 0, expert 2 for token 1
ids <- ggml_new_tensor_1d(ctx, GGML_TYPE_I32, 2)
ggml_set_i32(ids, c(0L, 2L))
output <- ggml_mul_mat_id(ctx, experts, input, ids)
graph <- ggml_build_forward_expand(ctx, output)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

---

ggml_nbytes *Get number of bytes*

Description

Get number of bytes

Get Number of Bytes

Usage

```
ggml_nbytes(tensor)

ggml_nbytes(tensor)
```

Arguments

tensor	Tensor

Value

Integer number of bytes

Integer number of bytes

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
tensor <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_nbytes(tensor)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_nbytes(t)
ggml_free(ctx)

## End(Not run)
```

---

ggml_neg *Negation (Graph)*

Description

Creates a graph node for element-wise negation: -x

Usage

```
ggml_neg(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the negation operation

---

ggml_nelements *Get number of elements*

Description

Get number of elements

Get Number of Elements

Usage

```
ggml_nelements(tensor)

ggml_nelements(tensor)
```

Arguments

tensor	Tensor

Value

Integer number of elements

Integer number of elements

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
tensor <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_nelements(tensor)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_nelements(t)
ggml_free(ctx)

## End(Not run)
```

---

ggml_new_f32 *Create Scalar F32 Tensor*

Description

Creates a 1-element tensor containing a single float value. Useful for scalar operations, learning rates, and other scalar floats.

Usage

```
ggml_new_f32(ctx, value)
```

Arguments

ctx	GGML context
value	Numeric value

Value

Tensor pointer (1-element F32 tensor)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
scalar <- ggml_new_f32(ctx, 3.14)
ggml_get_f32(scalar)
ggml_free(ctx)

## End(Not run)
```

---

ggml_new_i32 *Create Scalar I32 Tensor*

Description

Creates a 1-element tensor containing a single integer value. Useful for indices, counters, and other scalar integer operations.

Usage

```
ggml_new_i32(ctx, value)
```

Arguments

`ctx`	GGML context
`value`	Integer value

Value

Tensor pointer (1-element I32 tensor)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
scalar <- ggml_new_i32(ctx, 42)
ggml_get_i32(scalar)
ggml_free(ctx)

## End(Not run)
```

ggml_new_tensor — *Create Tensor with Arbitrary Dimensions*

Description

Generic tensor constructor for creating tensors with 1-4 dimensions. This is more flexible than the ggml_new_tensor_Nd functions.

Usage

```
ggml_new_tensor(ctx, type = GGML_TYPE_F32, n_dims, ne)
```

Arguments

`ctx`	GGML context
`type`	Data type (GGML_TYPE_F32, etc.)
`n_dims`	Number of dimensions (1-4)
`ne`	Numeric vector of dimension sizes

Value

Tensor pointer

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor(ctx, GGML_TYPE_F32, 3, c(10, 20, 30))
ggml_nelements(t)
ggml_free(ctx)

## End(Not run)
```

ggml_new_tensor_1d — *Create 1D tensor*

Description

Create 1D tensor

Create 1D Tensor

Usage

```
ggml_new_tensor_1d(ctx, type = GGML_TYPE_F32, ne0)

ggml_new_tensor_1d(ctx, type = GGML_TYPE_F32, ne0)
```

Arguments

ctx	GGML context
type	Data type
ne0	Size

Value

Tensor pointer

Tensor pointer

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
tensor <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_nelements(tensor)
ggml_free(ctx)

## End(Not run)
```

---

ggml_new_tensor_2d *Create 2D tensor*

Description

Create 2D tensor

Create 2D Tensor

Usage

```
ggml_new_tensor_2d(ctx, type = GGML_TYPE_F32, ne0, ne1)

ggml_new_tensor_2d(ctx, type = GGML_TYPE_F32, ne0, ne1)
```

Arguments

ctx	GGML context
type	Data type
ne0	Rows
ne1	Columns

Value

Tensor pointer

Tensor pointer

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
tensor <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_nelements(tensor)
ggml_free(ctx)

## End(Not run)
```

`ggml_new_tensor_3d` *Create 3D Tensor*

Description

Create 3D Tensor

Usage

```
ggml_new_tensor_3d(ctx, type = GGML_TYPE_F32, ne0, ne1, ne2)
```

Arguments

`ctx`	GGML context
`type`	Data type (default GGML_TYPE_F32)
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1
`ne2`	Size of dimension 2

Value

Tensor pointer

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_3d(ctx, GGML_TYPE_F32, 10, 20, 30)
ggml_nelements(t)
ggml_free(ctx)

## End(Not run)
```

ggml_new_tensor_4d *Create 4D Tensor*

Description

Create 4D Tensor

Usage

```
ggml_new_tensor_4d(ctx, type = GGML_TYPE_F32, ne0, ne1, ne2, ne3)
```

Arguments

ctx	GGML context
type	Data type (default GGML_TYPE_F32)
ne0	Size of dimension 0
ne1	Size of dimension 1
ne2	Size of dimension 2
ne3	Size of dimension 3

Value

Tensor pointer

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, 8, 8, 3, 2)
ggml_nelements(t)
ggml_free(ctx)

## End(Not run)
```

ggml_norm *Layer Normalization (Graph)*

Description

Creates a graph node for layer normalization.

Usage

```
ggml_norm(ctx, a, eps = 1e-05)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`eps`	Epsilon value for numerical stability (default: 1e-5)

Value

Tensor representing the layer normalization operation

`ggml_norm_inplace` *Layer Normalization In-place (Graph)*

Description

Creates a graph node for in-place layer normalization. Returns a view of the input tensor.

Usage

```
ggml_norm_inplace(ctx, a, eps = 1e-05)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (will be modified in-place)
`eps`	Epsilon value for numerical stability (default: 1e-5)

Value

View of input tensor with layer normalization applied

`ggml_nrows` *Get Number of Rows*

Description

Returns the number of rows in a tensor (product of all dimensions except ne[0]).

Usage

```
ggml_nrows(tensor)
```

Arguments

`tensor`	Tensor pointer

Value

Number of rows

ggml_n_dims *Get Number of Dimensions*

Description

Returns the number of dimensions of a tensor

Usage

```
ggml_n_dims(tensor)
```

Arguments

tensor	Tensor pointer

Value

Integer number of dimensions (1-4)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_n_dims(t)
ggml_free(ctx)

## End(Not run)
```

ggml_out_prod *Outer Product (Graph)*

Description

Computes the outer product of two vectors: C = a * b^T For vectors a[m] and b[n], produces matrix C[m, n].

Usage

```
ggml_out_prod(ctx, a, b)
```

Arguments

ctx	GGML context
a	First vector tensor
b	Second vector tensor

Value

Matrix tensor representing the outer product

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 3)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 4)
ggml_set_f32(a, c(1, 2, 3))
ggml_set_f32(b, c(1, 2, 3, 4))
c <- ggml_out_prod(ctx, a, b)  # Result: 3x4 matrix
graph <- ggml_build_forward_expand(ctx, c)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

---

ggml_pad — *Pad Tensor with Zeros (Graph)*

Description

Pads tensor dimensions with zeros on the right side. Useful for aligning tensor sizes in attention operations.

Usage

```
ggml_pad(ctx, a, p0 = 0L, p1 = 0L, p2 = 0L, p3 = 0L)
```

Arguments

`ctx`	GGML context
`a`	Input tensor to pad
`p0`	Padding for dimension 0 (default 0)
`p1`	Padding for dimension 1 (default 0)
`p2`	Padding for dimension 2 (default 0)
`p3`	Padding for dimension 3 (default 0)

Value

Padded tensor with shape [ne0+p0, ne1+p1, ne2+p2, ne3+p3]

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 5, 3)
ggml_set_f32(a, 1:15)
# Pad to 8x4
b <- ggml_pad(ctx, a, 3, 1)  # Add 3 zeros to dim0, 1 to dim1
graph <- ggml_build_forward_expand(ctx, b)
ggml_graph_compute(ctx, graph)
# Result shape: [8, 4]
ggml_free(ctx)

## End(Not run)
```

---

ggml_permute *Permute Tensor Dimensions (Graph)*

Description

Permutes the tensor dimensions according to specified axes. CRITICAL for attention mechanisms in transformers.

Usage

```
ggml_permute(ctx, a, axis0, axis1, axis2, axis3)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`axis0`	New position for axis 0
`axis1`	New position for axis 1
`axis2`	New position for axis 2
`axis3`	New position for axis 3

Value

Permuted tensor

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Create 4D tensor: (2, 3, 4, 5)
t <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, 2, 3, 4, 5)
# Swap axes 0 and 1: result shape (3, 2, 4, 5)
t_perm <- ggml_permute(ctx, t, 1, 0, 2, 3)
```

```
ggml_free(ctx)

## End(Not run)
```

`ggml_pool_1d` *1D Pooling (Graph)*

Description

Applies 1D pooling operation.

Usage

```
ggml_pool_1d(ctx, a, op, k0, s0 = k0, p0 = 0L)

GGML_OP_POOL_MAX

GGML_OP_POOL_AVG
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`op`	Pool operation: GGML_OP_POOL_MAX (0) or GGML_OP_POOL_AVG (1)
`k0`	Kernel size
`s0`	Stride (default = k0)
`p0`	Padding (default 0)

Format

An object of class `integer` of length 1.

An object of class `integer` of length 1.

Value

Pooled tensor

ggml_pool_2d — *2D Pooling (Graph)*

Description

Applies 2D pooling operation.

Usage

```
ggml_pool_2d(ctx, a, op, k0, k1, s0 = k0, s1 = k1, p0 = 0, p1 = 0)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`op`	Pool operation: GGML_OP_POOL_MAX (0) or GGML_OP_POOL_AVG (1)
`k0`	Kernel size dimension 0
`k1`	Kernel size dimension 1
`s0`	Stride dimension 0 (default = k0)
`s1`	Stride dimension 1 (default = k1)
`p0`	Padding dimension 0 (default 0)
`p1`	Padding dimension 1 (default 0)

Value

Pooled tensor

ggml_print_mem_status — *Print Context Memory Status*

Description

Helper to print memory usage information

Usage

```
ggml_print_mem_status(ctx)
```

Arguments

`ctx`	GGML context

Value

List with total, used, free memory (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_print_mem_status(ctx)
ggml_free(ctx)

## End(Not run)
```

`ggml_print_objects` *Print Objects in Context*

Description

Debug function to print all objects (tensors) in the context

Usage

```
ggml_print_objects(ctx)
```

Arguments

`ctx`	GGML context

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_print_objects(ctx)
ggml_free(ctx)

## End(Not run)
```

ggml_quantize_chunk *Quantize Data Chunk*

Description

Quantizes a chunk of floating-point data to a lower precision format.

Usage

```
ggml_quantize_chunk(type, src, nrows, n_per_row)
```

Arguments

type	Target GGML type (e.g., GGML_TYPE_Q4_0)
src	Source numeric vector (F32 data)
nrows	Number of rows
n_per_row	Number of elements per row

Value

Raw vector containing quantized data

Examples

```
## Not run:
# Quantize 256 floats to Q8_0 (block size 32)
data <- rnorm(256)
quantized <- ggml_quantize_chunk(GGML_TYPE_Q8_0, data, 1, 256)
ggml_quantize_free()  # Clean up

## End(Not run)
```

ggml_quantize_free *Free Quantization Resources*

Description

Frees any memory allocated by quantization. Call at end of program to avoid memory leaks.

Usage

```
ggml_quantize_free()
```

Value

NULL invisibly

ggml_quantize_init *Initialize Quantization Tables*

Description

Initializes quantization tables for a given type. Called automatically by ggml_quantize_chunk, but can be called manually.

Usage

```
ggml_quantize_init(type)
```

Arguments

type	GGML type (e.g., GGML_TYPE_Q4_0)

Value

NULL invisibly

ggml_quantize_requires_imatrix *Check if Quantization Requires Importance Matrix*

Description

Some quantization types require an importance matrix for optimal quality.

Usage

```
ggml_quantize_requires_imatrix(type)
```

Arguments

type	GGML type

Value

TRUE if importance matrix is required

ggml_reglu *ReGLU (ReLU Gated Linear Unit) (Graph)*

Description

Creates a graph node for ReGLU operation. ReGLU uses ReLU as the activation function on the first half.

Usage

```
ggml_reglu(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor (first dimension must be even)

Details

Formula: output = ReLU(x) * gate

Value

Tensor with half the first dimension of input

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 8, 3)
ggml_set_f32(a, rnorm(24))
r <- ggml_reglu(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # Shape: 4x3
ggml_free(ctx)

## End(Not run)
```

`ggml_reglu_split` *ReGLU Split (Graph)*

Description

Creates a graph node for ReGLU with separate input and gate tensors.

Usage

```
ggml_reglu_split(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (the values to be gated)
`b`	Gate tensor (same shape as a)

Details

Formula: output = ReLU(a) * b

Value

Tensor with same shape as input tensors

`ggml_relu` *ReLU Activation (Graph)*

Description

Creates a graph node for ReLU (Rectified Linear Unit) activation: max(0, x)

Usage

```
ggml_relu(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the ReLU operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
result <- ggml_relu(ctx, a)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

---

ggml_repeat *Repeat (Graph)*

Description

Creates a graph node that repeats tensor 'a' to match shape of tensor 'b'.

Usage

```
ggml_repeat(ctx, a, b)
```

Arguments

ctx	GGML context
a	Tensor to repeat
b	Target tensor (defines output shape)

Value

Tensor with repeated values

---

ggml_repeat_back *Repeat Backward (Graph)*

Description

Backward pass for repeat operation - sums repetitions back to original shape. Used for gradient computation during training.

Usage

```
ggml_repeat_back(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (gradients from repeated tensor)
`b`	Target shape tensor (original tensor before repeat)

Value

Tensor with summed gradients matching shape of b

`ggml_reset` *Reset GGML Context*

Description

Clears all tensor allocations in the context memory pool. The context can be reused without recreating it. This is more efficient than free + init for temporary operations.

Usage

```
ggml_reset(ctx)
```

Arguments

`ctx`	GGML context pointer

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 100)
ggml_reset(ctx)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 200)
ggml_free(ctx)

## End(Not run)
```

ggml_reshape_1d *Reshape to 1D (Graph)*

Description

Reshapes tensor to 1D with ne0 elements

Usage

```
ggml_reshape_1d(ctx, a, ne0)
```

Arguments

ctx	GGML context
a	Input tensor
ne0	Size of dimension 0

Value

Reshaped tensor

ggml_reshape_2d *Reshape to 2D (Graph)*

Description

Reshapes tensor to 2D with shape (ne0, ne1)

Usage

```
ggml_reshape_2d(ctx, a, ne0, ne1)
```

Arguments

ctx	GGML context
a	Input tensor
ne0	Size of dimension 0
ne1	Size of dimension 1

Value

Reshaped tensor

`ggml_reshape_3d` *Reshape to 3D (Graph)*

Description

Reshapes tensor to 3D with shape (ne0, ne1, ne2)

Usage

```
ggml_reshape_3d(ctx, a, ne0, ne1, ne2)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1
`ne2`	Size of dimension 2

Value

Reshaped tensor

`ggml_reshape_4d` *Reshape to 4D (Graph)*

Description

Reshapes tensor to 4D with shape (ne0, ne1, ne2, ne3)

Usage

```
ggml_reshape_4d(ctx, a, ne0, ne1, ne2, ne3)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1
`ne2`	Size of dimension 2
`ne3`	Size of dimension 3

Value

Reshaped tensor

ggml_rms_norm *RMS Normalization (Graph)*

Description

Creates a graph node for RMS (Root Mean Square) normalization. CRITICAL for LLaMA models.

Usage

```
ggml_rms_norm(ctx, a, eps = 1e-05)
```

Arguments

ctx	GGML context
a	Input tensor
eps	Epsilon value for numerical stability (default: 1e-5)

Value

Tensor representing the RMS normalization operation

ggml_rms_norm_back *RMS Norm Backward (Graph)*

Description

Creates a graph node for backward pass of RMS normalization. Used in training for computing gradients.

Usage

```
ggml_rms_norm_back(ctx, a, b, eps = 1e-05)
```

Arguments

ctx	GGML context
a	Input tensor (x from forward pass)
b	Gradient tensor (dy)
eps	Epsilon for numerical stability (default 1e-5)

Value

Tensor representing the gradient with respect to input

ggml_rms_norm_inplace *RMS Normalization In-place (Graph)*

Description

Creates a graph node for in-place RMS normalization. Returns a view of the input tensor. CRITICAL for LLaMA models when memory efficiency is important.

Usage

```
ggml_rms_norm_inplace(ctx, a, eps = 1e-05)
```

Arguments

ctx	GGML context
a	Input tensor (will be modified in-place)
eps	Epsilon value for numerical stability (default: 1e-5)

Value

View of input tensor with RMS normalization applied

ggml_rope *Rotary Position Embedding (Graph)*

Description

Creates a graph node for RoPE (Rotary Position Embedding). RoPE is the dominant position encoding method in modern LLMs like LLaMA, Mistral, and many others.

Usage

```
ggml_rope(ctx, a, b, n_dims, mode = 0L)
```

Arguments

ctx	GGML context
a	Input tensor of shape [head_dim, n_head, seq_len, batch]
b	Position tensor (int32) of shape [seq_len] containing position indices
n_dims	Number of dimensions to apply rotation to (usually head_dim)
mode	RoPE mode: GGML_ROPE_TYPE_NORM (0), GGML_ROPE_TYPE_NEOX (2), etc.

Details

RoPE encodes position information by rotating pairs of dimensions in the embedding space. The rotation angle depends on position and dimension index.

Key benefits of RoPE: - Relative position information emerges naturally from rotation - Better extrapolation to longer sequences than absolute embeddings - No additional parameters needed

Value

Tensor with same shape as input, with rotary embeddings applied

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Query tensor: head_dim=8, n_head=4, seq_len=16, batch=1
q <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, 8, 4, 16, 1)
ggml_set_f32(q, rnorm(8 * 4 * 16))
# Position indices
pos <- ggml_new_tensor_1d(ctx, GGML_TYPE_I32, 16)
ggml_set_i32(pos, 0:15)
# Apply RoPE
q_rope <- ggml_rope(ctx, q, pos, 8, GGML_ROPE_TYPE_NORM)
graph <- ggml_build_forward_expand(ctx, q_rope)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

`ggml_rope_ext` *Extended RoPE with Frequency Scaling (Graph)*

Description

Creates a graph node for extended RoPE with frequency scaling parameters. Supports context extension techniques like YaRN, Linear Scaling, etc.

Usage

```
ggml_rope_ext(
  ctx,
  a,
  b,
  c = NULL,
  n_dims,
  mode = 0L,
  n_ctx_orig = 0L,
  freq_base = 10000,
  freq_scale = 1,
```

```
  ext_factor = 0,
  attn_factor = 1,
  beta_fast = 32,
  beta_slow = 1
)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`b`	Position tensor (int32)
`c`	Optional frequency factors tensor (NULL for default)
`n_dims`	Number of dimensions to apply rotation to
`mode`	RoPE mode
`n_ctx_orig`	Original context length the model was trained on
`freq_base`	Base frequency for RoPE (default 10000 for most models)
`freq_scale`	Frequency scale factor (1.0 = no scaling)
`ext_factor`	YaRN extension factor (0.0 to disable)
`attn_factor`	Attention scale factor (typically 1.0)
`beta_fast`	YaRN parameter for fast dimensions
`beta_slow`	YaRN parameter for slow dimensions

Details

This extended version supports various context extension techniques:

- **Linear Scaling**: Set freq_scale = original_ctx / new_ctx - **YaRN**: Set ext_factor > 0 with appropriate beta_fast/beta_slow - **NTK-aware**: Adjust freq_base for NTK-style scaling

Common freq_base values: - LLaMA 1/2: 10000 - LLaMA 3: 500000 - Mistral: 10000 - Phi-3: 10000

Value

Tensor with extended RoPE applied

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
q <- ggml_new_tensor_4d(ctx, GGML_TYPE_F32, 64, 8, 32, 1)
ggml_set_f32(q, rnorm(64 * 8 * 32))
pos <- ggml_new_tensor_1d(ctx, GGML_TYPE_I32, 32)
ggml_set_i32(pos, 0:31)
# Standard RoPE with default freq_base
q_rope <- ggml_rope_ext(ctx, q, pos, NULL,
                        n_dims = 64, mode = 0L,
                        n_ctx_orig = 4096,
```

```
                      freq_base = 10000, freq_scale = 1.0,
                      ext_factor = 0.0, attn_factor = 1.0,
                      beta_fast = 32, beta_slow = 1)
graph <- ggml_build_forward_expand(ctx, q_rope)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

`ggml_rope_ext_back` *RoPE Extended Backward (Graph)*

Description

Backward pass for extended RoPE (Rotary Position Embedding). Used during training to compute gradients through RoPE.

Usage

```
ggml_rope_ext_back(
  ctx,
  a,
  b,
  c = NULL,
  n_dims,
  mode = 0L,
  n_ctx_orig = 0L,
  freq_base = 10000,
  freq_scale = 1,
  ext_factor = 0,
  attn_factor = 1,
  beta_fast = 32,
  beta_slow = 1
)
```

Arguments

`ctx`	GGML context
`a`	Gradient tensor from upstream (gradients of ggml_rope_ext result)
`b`	Position tensor (same as forward pass)
`c`	Optional frequency factors tensor (NULL for default)
`n_dims`	Number of dimensions for rotation
`mode`	RoPE mode
`n_ctx_orig`	Original context length
`freq_base`	Base frequency

freq_scale	Frequency scale factor
ext_factor	Extension factor (YaRN)
attn_factor	Attention factor
beta_fast	YaRN fast beta
beta_slow	YaRN slow beta

Value

Gradient tensor for the input

---

ggml_rope_inplace   *Rotary Position Embedding In-place (Graph)*

Description

In-place version of ggml_rope. Returns a view of the input tensor.

Usage

```
ggml_rope_inplace(ctx, a, b, n_dims, mode = 0L)
```

Arguments

ctx	GGML context
a	Input tensor (will be modified in-place)
b	Position tensor (int32)
n_dims	Number of dimensions to apply rotation to
mode	RoPE mode

Value

View of input tensor with RoPE applied

ggml_round — *Round (Graph)*

Description

Creates a graph node for element-wise rounding: round(x)

Usage

```
ggml_round(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the round operation

ggml_scale — *Scale (Graph)*

Description

Creates a graph node for scaling tensor by a scalar: x * s

Usage

```
ggml_scale(ctx, a, s)
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`s`	Scalar value to multiply by

Value

Tensor representing the scaled values

ggml_set	*Set Tensor Region (Graph)*

Description

Copies tensor b into tensor a at a specified offset. This allows writing to a portion of a tensor.

Usage

```
ggml_set(ctx, a, b, nb1, nb2, nb3, offset)
```

Arguments

`ctx`	GGML context
`a`	Destination tensor
`b`	Source tensor (data to copy)
`nb1`	Stride for dimension 1 (in bytes)
`nb2`	Stride for dimension 2 (in bytes)
`nb3`	Stride for dimension 3 (in bytes)
`offset`	Byte offset in destination tensor

Value

Tensor representing the set operation

ggml_set_1d	*Set 1D Tensor Region (Graph)*

Description

Simplified 1D version of ggml_set. Copies tensor b into tensor a starting at offset.

Usage

```
ggml_set_1d(ctx, a, b, offset)
```

Arguments

`ctx`	GGML context
`a`	Destination tensor
`b`	Source tensor
`offset`	Byte offset in destination tensor

Value

Tensor representing the set operation

ggml_set_2d *Set 2D Tensor Region (Graph)*

Description

Simplified 2D version of ggml_set.

Usage

```
ggml_set_2d(ctx, a, b, nb1, offset)
```

Arguments

`ctx`	GGML context
`a`	Destination tensor
`b`	Source tensor
`nb1`	Stride for dimension 1 (in bytes)
`offset`	Byte offset in destination tensor

Value

Tensor representing the set operation

ggml_set_f32 *Set F32 data*

Description

Set F32 data

Set F32 Data

Usage

```
ggml_set_f32(tensor, data)

ggml_set_f32(tensor, data)
```

Arguments

`tensor`	Tensor
`data`	Numeric vector

Value

NULL (invisible)

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
tensor <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(tensor, c(1, 2, 3, 4, 5))
ggml_get_f32(tensor)
ggml_free(ctx)

## End(Not run)
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(t, c(1, 2, 3, 4, 5))
ggml_get_f32(t)
ggml_free(ctx)

## End(Not run)
```

---

ggml_set_i32 *Set I32 Data*

Description

Sets integer data in an I32 tensor. Used for indices (ggml_get_rows) and position tensors (ggml_rope).

Usage

```
ggml_set_i32(tensor, data)
```

Arguments

tensor	Tensor of type GGML_TYPE_I32
data	Integer vector

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
pos <- ggml_new_tensor_1d(ctx, GGML_TYPE_I32, 10)
ggml_set_i32(pos, 0:9)
ggml_get_i32(pos)
ggml_free(ctx)

## End(Not run)
```

---

ggml_set_name *Set Tensor Name*

Description

Assigns a name to a tensor. Useful for debugging and graph visualization.

Usage

```
ggml_set_name(tensor, name)
```

Arguments

tensor	Tensor pointer
name	Character string name

Value

The tensor (for chaining)

---

ggml_set_no_alloc *Set No Allocation Mode*

Description

When enabled, tensor creation will not allocate memory for data. Useful for creating computation graphs without allocating storage.

Usage

```
ggml_set_no_alloc(ctx, no_alloc)
```

Arguments

ctx	GGML context
no_alloc	Logical, TRUE to disable allocation

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
ggml_set_no_alloc(ctx, TRUE)
ggml_get_no_alloc(ctx)
ggml_set_no_alloc(ctx, FALSE)
ggml_free(ctx)

## End(Not run)
```

---

ggml_set_n_threads *Set Number of Threads*

Description

Set the number of threads for GGML operations

Usage

```
ggml_set_n_threads(n_threads)
```

Arguments

n_threads	Number of threads to use

Value

Number of threads set

Examples

```
## Not run:
# Use 4 threads
ggml_set_n_threads(4)

# Use all available cores
ggml_set_n_threads(parallel::detectCores())

## End(Not run)
```

ggml_set_zero *Set Tensor to Zero*

Description

Sets all elements of a tensor to zero. This is more efficient than manually setting all elements.

Usage

```
ggml_set_zero(tensor)
```

Arguments

tensor	Tensor to zero out

Value

NULL (invisible)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_set_f32(t, 1:10)
ggml_set_zero(t)
ggml_get_f32(t)
ggml_free(ctx)

## End(Not run)
```

ggml_sgn *Sign Function (Graph)*

Description

Creates a graph node for element-wise sign function. sgn(x) = -1 if x < 0, 0 if x == 0, 1 if x > 0

Usage

```
ggml_sgn(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the sign operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -0.5, 0, 0.5, 2))
r <- ggml_sgn(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # c(-1, -1, 0, 1, 1)
ggml_free(ctx)

## End(Not run)
```

---

ggml_sigmoid *Sigmoid Activation (Graph)*

Description

Creates a graph node for sigmoid activation: 1 / (1 + exp(-x))

Usage

```
ggml_sigmoid(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the sigmoid operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
result <- ggml_sigmoid(ctx, a)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

ggml_silu *SiLU Activation (Graph)*

Description

Creates a graph node for SiLU (Sigmoid Linear Unit) activation, also known as Swish. CRITICAL for LLaMA models.

Usage

```
ggml_silu(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the SiLU operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
result <- ggml_silu(ctx, a)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

ggml_silu_back *SiLU Backward (Graph)*

Description

Computes the backward pass for SiLU (Swish) activation. Used during training for gradient computation.

Usage

```
ggml_silu_back(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	Forward input tensor
`b`	Gradient tensor from upstream

Value

Gradient tensor for the input

`ggml_sin` *Sine (Graph)*

Description

Creates a graph node for element-wise sine: sin(x)

Usage

```
ggml_sin(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the sin operation

`ggml_softplus` *Softplus Activation (Graph)*

Description

Creates a graph node for Softplus activation. Softplus(x) = log(1 + exp(x)). A smooth approximation of ReLU.

Usage

```
ggml_softplus(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the Softplus operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
r <- ggml_softplus(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)
ggml_free(ctx)

## End(Not run)
```

---

ggml_soft_max *Softmax (Graph)*

Description

Creates a graph node for softmax operation. CRITICAL for attention mechanisms.

Usage

```
ggml_soft_max(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor

Value

Tensor representing the softmax operation ggml_soft_max_ext *Extended Softmax with Masking and Scaling (Graph)*

Description

Creates a graph node for fused softmax operation with optional masking and ALiBi (Attention with Linear Biases) support. Computes: softmax(a * scale + mask * (ALiBi slope)) CRITICAL for efficient attention computation in transformers.

Usage

```
ggml_soft_max_ext(ctx, a, mask = NULL, scale = 1, max_bias = 0)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (typically attention scores)
`mask`	Optional attention mask tensor (F16 or F32). NULL for no mask. Shape must be broadcastable to input tensor.
`scale`	Scaling factor, typically 1/sqrt(head_dim)
`max_bias`	Maximum ALiBi bias (0.0 to disable ALiBi)

Details

This extended softmax is commonly used in transformer attention: 1. Scale attention scores by 1/sqrt(d_k) for numerical stability 2. Apply attention mask (e.g., causal mask, padding mask) 3. Optionally apply ALiBi position bias 4. Compute softmax

All these operations are fused for efficiency.

Value

Tensor representing the scaled and masked softmax

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
scores <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 10)
ggml_set_f32(scores, rnorm(100))
attn <- ggml_soft_max_ext(ctx, scores, NULL, 1.0, max_bias = 0.0)
graph <- ggml_build_forward_expand(ctx, attn)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

ggml_soft_max_ext_back *Softmax Backward Extended (Graph)*

Description

Backward pass for extended softmax operation.

Usage

```
ggml_soft_max_ext_back(ctx, a, b, scale = 1, max_bias = 0)
```

Arguments

`ctx`	GGML context
`a`	Softmax output tensor (from forward pass)
`b`	Gradient tensor from upstream
`scale`	Scale factor (same as forward pass)
`max_bias`	Maximum ALiBi bias (same as forward pass)

Value

Gradient tensor for the input

ggml_soft_max_inplace *Softmax In-place (Graph)*

Description

Creates a graph node for in-place softmax operation. Returns a view of the input tensor.

Usage

```
ggml_soft_max_inplace(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (will be modified in-place)

Value

View of input tensor with softmax applied

GGML_SORT_ORDER_ASC *Sort Order Constants*

Description

Constants for specifying sort order in argsort operations.

Usage

```
GGML_SORT_ORDER_ASC

GGML_SORT_ORDER_DESC
```

Format

Integer constants

An object of class `integer` of length 1.

Examples

```
## Not run:
GGML_SORT_ORDER_ASC   # 0 - Ascending order
GGML_SORT_ORDER_DESC  # 1 - Descending order

## End(Not run)
```

ggml_sqr *Square (Graph)*

Description

Creates a graph node for element-wise squaring: x^2

Usage

```
ggml_sqr(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the square operation

ggml_sqrt — *Square Root (Graph)*

Description

Creates a graph node for element-wise square root: sqrt(x)

Usage

```
ggml_sqrt(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the sqrt operation

ggml_step — *Step Function (Graph)*

Description

Creates a graph node for element-wise step function. step(x) = 0 if x <= 0, 1 if x > 0 Also known as the Heaviside step function.

Usage

```
ggml_step(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the step operation

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -0.5, 0, 0.5, 2))
r <- ggml_step(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # c(0, 0, 0, 1, 1)
ggml_free(ctx)

## End(Not run)
```

`ggml_sub` *Element-wise Subtraction (Graph)*

Description

Creates a graph node for element-wise subtraction.

Usage

```
ggml_sub(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	First tensor
`b`	Second tensor (same shape as a)

Value

Tensor representing the subtraction operation (a - b)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
b <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(5, 4, 3, 2, 1))
ggml_set_f32(b, c(1, 1, 1, 1, 1))
result <- ggml_sub(ctx, a, b)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

`ggml_sum` *Sum (Graph)*

Description

Creates a graph node that computes the sum of all elements.

Usage

```
ggml_sum(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Scalar tensor with the sum

`ggml_sum_rows` *Sum Rows (Graph)*

Description

Creates a graph node that computes the sum along rows.

Usage

```
ggml_sum_rows(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor with row sums

ggml_swiglu *SwiGLU (Swish/SiLU Gated Linear Unit) (Graph)*

Description

Creates a graph node for SwiGLU operation. SwiGLU uses SiLU (Swish) as the activation function on the first half. CRITICAL for LLaMA, Mistral, and many modern LLMs.

Usage

```
ggml_swiglu(ctx, a)
```

Arguments

ctx	GGML context
a	Input tensor (first dimension must be even)

Details

Formula: output = SiLU(x) * gate

Value

Tensor with half the first dimension of input

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 8, 3)
ggml_set_f32(a, rnorm(24))
r <- ggml_swiglu(ctx, a)
graph <- ggml_build_forward_expand(ctx, r)
ggml_graph_compute(ctx, graph)
result <- ggml_get_f32(r)  # Shape: 4x3
ggml_free(ctx)

## End(Not run)
```

`ggml_swiglu_split` *SwiGLU Split (Graph)*

Description

Creates a graph node for SwiGLU with separate input and gate tensors.

Usage

```
ggml_swiglu_split(ctx, a, b)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (the values to be gated)
`b`	Gate tensor (same shape as a)

Details

Formula: output = SiLU(a) * b

Value

Tensor with same shape as input tensors

`ggml_tanh` *Tanh Activation (Graph)*

Description

Creates a graph node for hyperbolic tangent activation.

Usage

```
ggml_tanh(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor

Value

Tensor representing the tanh operation

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 5)
ggml_set_f32(a, c(-2, -1, 0, 1, 2))
result <- ggml_tanh(ctx, a)
graph <- ggml_build_forward_expand(ctx, result)
ggml_graph_compute(ctx, graph)
ggml_get_f32(result)
ggml_free(ctx)

## End(Not run)
```

---

ggml_tensor_overhead *Get Tensor Overhead*

Description

Returns the memory overhead (metadata) for each tensor in bytes

Usage

```
ggml_tensor_overhead()
```

Value

Size in bytes

Examples

```
## Not run:
ggml_tensor_overhead()

## End(Not run)
```

---

ggml_tensor_shape *Get Tensor Shape*

Description

Returns the shape of a tensor as a numeric vector of 4 elements (ne0, ne1, ne2, ne3)

Usage

```
ggml_tensor_shape(tensor)
```

Arguments

`tensor`	Tensor pointer

Value

Numeric vector of length 4 with dimensions

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 10, 20)
ggml_tensor_shape(t)
ggml_free(ctx)

## End(Not run)
```

`ggml_tensor_type` *Get Tensor Type*

Description

Returns the data type of a tensor as an integer code

Usage

```
ggml_tensor_type(tensor)
```

Arguments

`tensor`	Tensor pointer

Value

Integer type code (0 = F32, 1 = F16, etc.)

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
t <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
ggml_tensor_type(t)
ggml_free(ctx)

## End(Not run)
```

ggml_test — *Test GGML*

Description

Runs GGML library self-test and prints version info.

Usage

```
ggml_test()
```

Value

TRUE if test passed

Examples

```
## Not run:
ggml_test()

## End(Not run)
```

ggml_time_init — *Initialize GGML Timer*

Description

Initializes the GGML timing system. Call this once at the beginning of the program before using ggml_time_ms() or ggml_time_us().

Usage

```
ggml_time_init()
```

Value

NULL (invisible)

Examples

```
## Not run:
ggml_time_init()
start <- ggml_time_ms()
Sys.sleep(0.01)
elapsed <- ggml_time_ms() - start

## End(Not run)
```

`ggml_time_ms` *Get Time in Milliseconds*

Description

Returns the current time in milliseconds since the timer was initialized.

Usage

```
ggml_time_ms()
```

Value

Numeric value representing milliseconds

Examples

```
## Not run:
ggml_time_init()
start <- ggml_time_ms()
Sys.sleep(0.01)
elapsed <- ggml_time_ms() - start

## End(Not run)
```

`ggml_time_us` *Get Time in Microseconds*

Description

Returns the current time in microseconds since the timer was initialized. More precise than ggml_time_ms() for micro-benchmarking.

Usage

```
ggml_time_us()
```

Value

Numeric value representing microseconds

Examples

```
## Not run:
ggml_time_init()
start <- ggml_time_us()
Sys.sleep(0.001)
elapsed <- ggml_time_us() - start

## End(Not run)
```

---

ggml_top_k   *Top-K Indices (Graph)*

Description

Returns the indices of top K elements per row. Useful for sampling strategies in language models (top-k sampling). Note: the resulting indices are in no particular order within top-k.

Usage

```
ggml_top_k(ctx, a, k)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (F32)
`k`	Number of top elements to return per row

Value

Tensor containing I32 indices of top-k elements (not values)

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
# Logits from model output
logits <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 100)
ggml_set_f32(logits, rnorm(100))
# Get top 5 logits for sampling
top5 <- ggml_top_k(ctx, logits, 5)
graph <- ggml_build_forward_expand(ctx, top5)
ggml_graph_compute(ctx, graph)
ggml_free(ctx)

## End(Not run)
```

ggml_transpose — *Transpose (Graph)*

Description

Creates a graph node for matrix transpose operation.

Usage

```
ggml_transpose(ctx, a)
```

Arguments

`ctx`	GGML context
`a`	Input tensor (2D matrix)

Value

Tensor representing the transposed matrix

GGML_TYPE_F32 — *GGML Data Types*

Description

Constants representing different data types supported by GGML.

Usage

```
GGML_TYPE_F32

GGML_TYPE_F16

GGML_TYPE_Q4_0

GGML_TYPE_Q4_1

GGML_TYPE_Q8_0

GGML_TYPE_I32
```

Format

Integer constants

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

Details

- `GGML_TYPE_F32`: 32-bit floating point (default)
- `GGML_TYPE_F16`: 16-bit floating point (half precision)
- `GGML_TYPE_Q4_0`: 4-bit quantization type 0
- `GGML_TYPE_Q4_1`: 4-bit quantization type 1
- `GGML_TYPE_Q8_0`: 8-bit quantization type 0
- `GGML_TYPE_I32`: 32-bit integer

Examples

```
## Not run:
GGML_TYPE_F32
GGML_TYPE_F16
GGML_TYPE_I32

## End(Not run)
```

`ggml_type_size` *Get Type Size in Bytes*

Description

Returns the size in bytes for all elements in a block for a given type.

Usage

```
ggml_type_size(type)
```

Arguments

`type`	GGML type constant (e.g., GGML_TYPE_F32)

Value

Size in bytes

ggml_upscale *Upscale Tensor (Graph)*

Description

Upscales tensor by multiplying ne0 and ne1 by scale factor. Supports different interpolation modes.

Usage

```
ggml_upscale(ctx, a, scale_factor, mode = 0L)

GGML_SCALE_MODE_NEAREST

GGML_SCALE_MODE_BILINEAR

GGML_SCALE_MODE_BICUBIC
```

Arguments

`ctx`	GGML context
`a`	Input tensor
`scale_factor`	Integer scale factor (e.g., 2 = double size)
`mode`	Scale mode: 0 = nearest, 1 = bilinear, 2 = bicubic

Format

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

Value

Upscaled tensor

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
img <- ggml_new_tensor_2d(ctx, GGML_TYPE_F32, 8, 8)
ggml_set_f32(img, rnorm(64))
upscaled <- ggml_upscale(ctx, img, 2, GGML_SCALE_MODE_NEAREST)
graph <- ggml_build_forward_expand(ctx, upscaled)
ggml_graph_compute(ctx, graph)
# Result is 16x16
ggml_free(ctx)

## End(Not run)
```

ggml_used_mem *Get Used Memory*

Description

Returns the amount of memory currently used in the context

Usage

```
ggml_used_mem(ctx)
```

Arguments

ctx	GGML context

Value

Used memory in bytes

Examples

```
## Not run:
ctx <- ggml_init(1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 100)
ggml_used_mem(ctx)
ggml_free(ctx)

## End(Not run)
```

ggml_version *Get GGML version*

Description

Get GGML version

Usage

```
ggml_version()
```

Value

Character string with GGML version

Examples

```
## Not run:
ggml_version()

## End(Not run)
```

---

ggml_view_1d *1D View with Byte Offset (Graph)*

Description

Creates a 1D view of a tensor starting at a byte offset. The view shares memory with the source tensor.

Usage

```
ggml_view_1d(ctx, a, ne0, offset = 0)
```

Arguments

ctx	GGML context
a	Source tensor
ne0	Number of elements in the view
offset	Byte offset from the start of tensor data

Value

View tensor

Examples

```
## Not run:
ctx <- ggml_init(16 * 1024 * 1024)
a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 100)
# View elements 10-19 (offset = 10 * 4 bytes = 40)
v <- ggml_view_1d(ctx, a, 10, 40)
ggml_free(ctx)

## End(Not run)
```

`ggml_view_2d` *2D View with Byte Offset (Graph)*

Description

Creates a 2D view of a tensor starting at a byte offset. The view shares memory with the source tensor.

Usage

```
ggml_view_2d(ctx, a, ne0, ne1, nb1, offset = 0)
```

Arguments

`ctx`	GGML context
`a`	Source tensor
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1
`nb1`	Stride for dimension 1 (in bytes)
`offset`	Byte offset from the start of tensor data

Value

View tensor

`ggml_view_3d` *3D View with Byte Offset (Graph)*

Description

Creates a 3D view of a tensor starting at a byte offset. The view shares memory with the source tensor.

Usage

```
ggml_view_3d(ctx, a, ne0, ne1, ne2, nb1, nb2, offset = 0)
```

Arguments

`ctx`	GGML context
`a`	Source tensor
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1
`ne2`	Size of dimension 2
`nb1`	Stride for dimension 1 (in bytes)
`nb2`	Stride for dimension 2 (in bytes)
`offset`	Byte offset from the start of tensor data

Value

View tensor

`ggml_view_4d` *4D View with Byte Offset (Graph)*

Description

Creates a 4D view of a tensor starting at a byte offset. The view shares memory with the source tensor. CRITICAL for KV-cache operations in transformers.

Usage

```
ggml_view_4d(ctx, a, ne0, ne1, ne2, ne3, nb1, nb2, nb3, offset = 0)
```

Arguments

`ctx`	GGML context
`a`	Source tensor
`ne0`	Size of dimension 0
`ne1`	Size of dimension 1
`ne2`	Size of dimension 2
`ne3`	Size of dimension 3
`nb1`	Stride for dimension 1 (in bytes)
`nb2`	Stride for dimension 2 (in bytes)
`nb3`	Stride for dimension 3 (in bytes)
`offset`	Byte offset from the start of tensor data

Value

View tensor

ggml_view_tensor *View Tensor*

Description

Creates a view of the tensor (shares data, no copy)

Usage

```
ggml_view_tensor(ctx, src)
```

Arguments

ctx	GGML context
src	Source tensor

Value

View tensor (shares data with src)

ggml_vulkan_available *Check if Vulkan support is available*

Description

Returns TRUE if the package was compiled with Vulkan support. To enable Vulkan, reinstall with: install.packages(..., configure.args = "–with-vulkan")

Usage

```
ggml_vulkan_available()
```

Value

Logical indicating if Vulkan is available

Examples

```
ggml_vulkan_available()
```

`ggml_vulkan_backend_name` *Get Vulkan backend name*

Description

Returns the name of the Vulkan backend (includes device info).

Usage

```
ggml_vulkan_backend_name(backend)
```

Arguments

`backend`	Vulkan backend pointer

Value

Character string with backend name

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  backend <- ggml_vulkan_init(0)
  print(ggml_vulkan_backend_name(backend))
  ggml_vulkan_free(backend)
}

## End(Not run)
```

`ggml_vulkan_device_count` *Get number of Vulkan devices*

Description

Returns the number of available Vulkan-capable GPU devices.

Usage

```
ggml_vulkan_device_count()
```

Value

Integer count of Vulkan devices (0 if Vulkan not available)

Examples

```
## Not run:
if (ggml_vulkan_available()) {
  ggml_vulkan_device_count()
}

## End(Not run)
```

---

ggml_vulkan_device_description *Get Vulkan device description*

Description

Returns a human-readable description of the specified Vulkan device.

Usage

```
ggml_vulkan_device_description(device = 0L)
```

Arguments

device	Device index (0-based)

Value

Character string with device description

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  ggml_vulkan_device_description(0)
}

## End(Not run)
```

ggml_vulkan_device_memory *Get Vulkan device memory*

Description

Returns free and total memory for the specified Vulkan device.

Usage

```
ggml_vulkan_device_memory(device = 0L)
```

Arguments

device	Device index (0-based)

Value

Named list with 'free' and 'total' memory in bytes

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  mem <- ggml_vulkan_device_memory(0)
  cat("Free:", mem$free / 1e9, "GB\n")
  cat("Total:", mem$total / 1e9, "GB\n")
}

## End(Not run)
```

ggml_vulkan_free *Free Vulkan backend*

Description

Releases resources associated with the Vulkan backend.

Usage

```
ggml_vulkan_free(backend)
```

Arguments

backend	Vulkan backend pointer from ggml_vulkan_init()

Value

NULL (invisible)

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  backend <- ggml_vulkan_init(0)
  ggml_vulkan_free(backend)
}

## End(Not run)
```

---

ggml_vulkan_init   *Initialize Vulkan backend*

Description

Creates a Vulkan backend for the specified device. The backend must be freed with ggml_vulkan_free() when done.

Usage

```
ggml_vulkan_init(device = 0L)
```

Arguments

device	Device index (0-based, default 0)

Value

Vulkan backend pointer

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  backend <- ggml_vulkan_init(0)
  print(ggml_vulkan_backend_name(backend))
  ggml_vulkan_free(backend)
}

## End(Not run)
```

`ggml_vulkan_is_backend`

Check if backend is Vulkan

Description

Returns TRUE if the given backend is a Vulkan backend.

Usage

```
ggml_vulkan_is_backend(backend)
```

Arguments

backend	Backend pointer

Value

Logical indicating if backend is Vulkan

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  vk_backend <- ggml_vulkan_init(0)
  cpu_backend <- ggml_backend_cpu_init()

  ggml_vulkan_is_backend(vk_backend)  # TRUE
  ggml_vulkan_is_backend(cpu_backend) # FALSE

  ggml_vulkan_free(vk_backend)
  ggml_backend_free(cpu_backend)
}

## End(Not run)
```

`ggml_vulkan_list_devices`

List all Vulkan devices

Description

Returns detailed information about all available Vulkan devices.

Usage

```
ggml_vulkan_list_devices()
```

Value

List of device information (index, name, memory)

Examples

```
## Not run:
if (ggml_vulkan_available() && ggml_vulkan_device_count() > 0) {
  devices <- ggml_vulkan_list_devices()
  print(devices)
}

## End(Not run)
```

---

ggml_vulkan_status *Print Vulkan status*

Description

Prints information about Vulkan availability and devices.

Usage

```
ggml_vulkan_status()
```

Value

NULL (invisible), prints status to console

Examples

```
ggml_vulkan_status()
```

---

ggml_with_temp_ctx *Execute with Temporary Context*

Description

Creates a temporary context, executes code, and frees it automatically. Useful when you need to create large temporary tensors.

Usage

```
ggml_with_temp_ctx(mem_size, expr)
```

Arguments

`mem_size`	Context memory size in bytes
`expr`	Expression to evaluate with the temporary context

Value

Result of the expression

Examples

```
## Not run:
# Create tensors in temporary context
result <- ggml_with_temp_ctx(1024 * 1024, {
  a <- ggml_new_tensor_1d(ctx, GGML_TYPE_F32, 10)
  ggml_set_f32(a, 1:10)
  ggml_get_f32(a)
})

## End(Not run)
```

`rope_types` *RoPE Mode Constants*

Description

Constants for RoPE (Rotary Position Embedding) modes.

Usage

```
GGML_ROPE_TYPE_NORM

GGML_ROPE_TYPE_NEOX

GGML_ROPE_TYPE_MROPE

GGML_ROPE_TYPE_VISION
```

Format

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

Details

- GGML_ROPE_TYPE_NORM (0): Standard RoPE as in original paper - GGML_ROPE_TYPE_NEOX (2): GPT-NeoX style RoPE (interleaved differently) - GGML_ROPE_TYPE_MROPE (8): Multi-RoPE for models like Qwen2-VL - GGML_ROPE_TYPE_VISION (24): Vision model RoPE

Index