

Package ‘opencltools’

July 11, 2026

Type Package

Title 'OpenCL' Tools for R Package Developers

Version 0.8.2

Date 2026-07-11

Description Runtime 'OpenCL' support for R package developers: probe hardware and drivers, load and concatenate kernel sources, and manage dependency-annotated '.cl' libraries, so packages like 'nmathopencl' and other ported libraries can offer GPU acceleration without each re-implementing the same plumbing. Vignettes illustrate integration with suggested packages 'nmathopencl' and 'glmbytes'; production kernels for those applications ship in those packages rather than here.

License GPL-2

URL <https://github.com/knygren/opencltools>,
<https://knygren.r-universe.dev/opencltools>

BugReports <https://github.com/knygren/opencltools/issues>

Imports stats, Rcpp (>= 1.1.1), RcppParallel, Rdpack (>= 0.11-0),
jsonlite

RdMacros Rdpack

LinkingTo Rcpp, RcppArmadillo, RcppParallel

Depends R (>= 3.5.0)

Suggests glmbytes (>= 0.9.3), nmathopencl, testthat (>= 3.0.0),
spelling, knitr, rmarkdown

VignetteBuilder knitr

SystemRequirements Optional 'OpenCL' support. If available, GPU acceleration will be used; otherwise, computation runs on CPU.

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

Language en-US

NeedsCompilation yes

Author Kjell Nygren [aut, cre],

The R Core Team [ctb, cph] (R Mathlib sources, R stats modeling code,
and derived/adapted routines),

The R Foundation [cph] (Portions of R Mathlib and R source code),

Ross Ihaka [ctb, cph] (R Mathlib and original R modeling
infrastructure),

Robert Gentleman [ctb, cph] (Portions of R Mathlib),

Simon Davies [ctb] (Original R glm implementation),

Morten Welinder [ctb, cph] (Portions of R Mathlib),

Martin Maechler [ctb] (Portions of R Mathlib)

Maintainer Kjell Nygren <kjell.a.nygren@gmail.com>

Repository CRAN

Date/Publication 2026-07-11 15:20:02 UTC

Contents

openclttools-package	3
add_to_path	4
attach_cross_library_tags	5
attach_kernel_call_tags	7
attach_kernel_dependency_tags	10
diagnose_glmbyes	11
extract_library_subset	15
get_openccl_core_count	17
kernel_lib_subset_printing	18
load_kernel_source	20
load_library_for_kernel	27
load_library_for_kernel_cross_package	29
load_program_preload	30
openclttoolsLdFlags	31
port_to_openccl_configure	32
print.openccl_dependency_tags	34
read_program_preload_manifest	35
stage_kernel_dependency_sort	36
use_openccl_configure	37
write_kernel_dependency_index	39
write_program_preload_manifest	40

Index	43
--------------	-----------

openclttools-package *openclttools: OpenCL Tools for R Package Developers*

Description

openclttools provides runtime OpenCL support for R: probe hardware and drivers, load and concatenate kernel sources, and manage dependency-annotated .cl libraries. Downstream packages such as **nmathopencl** ship ported kernel trees and call [load_kernel_library](#) / [load_library_for_kernel](#) from this package rather than re-implementing the same plumbing.

Details

Typical workflow for a library author:

1. Annotate .cl shards with @depends / @provides.
2. Use [load_library_for_kernel](#) or [extract_library_subset](#) to assemble minimal subsets.
3. Probe the workstation with [has_opencl](#), [verify_opencl_runtime](#), and [diagnose_glbayes](#).
4. For a new downstream package, call [use_opencl_configure](#) or [port_to_opencl_configure](#) to install CRAN-safe configure scripts.

Citation

Use `citation("openclttools")` for BibTeX. The first entry cites this package as software; the second cites Stone *et al.* (2010) when reporting OpenCL GPU execution. Cite **nmathopencl** for ported Mathlib kernels and **glmbayes** for the Bayesian GLM application layer. Likelihood sub-gradient methodology (Nygren and Nygren, 2006) applies to **glmbayes** envelope sampling, not to citing this loader/runtime layer alone.

Author(s)

Kjell Nygren

References

Stone JE, Gohara D, Shi G (2010). *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*. Computing in Science and Engineering **12**(3), 66–72. doi:10.1109/MCSE.2010.69

See Also

[load_kernel_library](#), [load_kernel_source](#), [has_opencl](#), [opencl_device_info](#).

add_to_path

Add Directories to PATH or LD_LIBRARY_PATH

Description

These helper functions allow you to add missing directories to the PATH or library search environment variables in a permanent way, minimizing manual editing.

Usage

```
add_to_path_windows(dirs)
```

```
add_to_path_linux(dirs)
```

```
add_to_libpath_linux(dirs)
```

Arguments

dirs Character vector of directories to add.

Details

- On **Windows**, updates the user-level PATH via PowerShell.
- On **Linux/WSL**, appends export lines to ~/.bashrc for PATH or LD_LIBRARY_PATH.

Value

No return value; called for side effects.

See Also

[Sys.getenv](#), [Sys.setenv](#)

Examples

```
##### Start of add_to_path example #####

if (interactive()) {
  add_to_path_windows(c("C:/OpenCL/bin"))
  add_to_path_linux(c("/usr/local/cuda/bin"))
  add_to_libpath_linux(c("/usr/local/cuda/lib64"))
}

#####
## End of add_to_path example
#####
```

attach_cross_library_tags

Attach Cross-Library Dependency Tags to Kernel Files

Description

Given a set of user-facing kernel .cl files and a library directory, computes the full transitive dependency list for each kernel (using the library's pre-built dependency index) and writes the results back into the kernel files as annotation tags.

Usage

```
attach_cross_library_tags(
  kernel_paths,
  library_dir,
  depends_tag = "depends_nmath",
  index = NULL,
  dry_run = FALSE
)
```

Arguments

kernel_paths	Character vector of paths to kernel .cl files.
library_dir	Path to the library directory containing kernel_dependency_index.rds and the library .cl files.
depends_tag	Name of the annotation tag in the kernel files that lists the direct library entry-point stems (e.g. "depends_nmath"). The function reads @{depends_tag} and writes @all_{depends_tag} and @all_{depends_tag}_count.
index	Optional dependency index (write_kernel_dependency_index , load via readRDS). If NULL, reads file.path(library_dir, "kernel_dependency_index.rds") with message().
dry_run	Logical; if TRUE, compute tags but do not write any files.

Details

Cross-library analogue of [attach_kernel_dependency_tags](#): it targets kernels that depend on a library through a @{depends_tag} tag (entry-point stems), instead of expanding @depends purely inside one library tree.

Typical usage for kernels that call nmath functions:

```
nmath_dir <- system.file(
  "cl", "nmath", package = "opencltools")
idx <- readRDS(file.path(nmath_dir, "kernel_dependency_index.rds"))

attach_cross_library_tags(
```

```

    kernel_paths = list.files("inst/cl/src", "\\\\.cl$", full.names = TRUE),
    library_dir  = nmath_dir,
    depends_tag  = "depends_nmath",
    index       = idx
  )

```

This writes @all_depends_nmath_count and @all_depends_nmath into each kernel file that carries a @depends_nmath annotation.

Value

A data frame (returned invisibly) with one row per kernel file and columns:

file Basename of the kernel file.

direct_stems Comma-separated direct entry-point stems read from @{depends_tag}.

all_depends_count Number of library files in the full transitive closure.

all_depends Comma-separated full transitive dependency list in load order.

changed TRUE if the file was (or would be, under dry_run) modified.

See Also

[attach_kernel_dependency_tags](#) [write_kernel_dependency_index](#) [load_library_for_kernel](#)

Examples

```

##### Start of kernel_tagging_workflow example #####

lib_dir <- system.file("cl/nmath_small", package = "opencltools")
kernels <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")

# Step 1: scan a launcher kernel for library calls (read-only dry run)
step1 <- attach_kernel_call_tags(
  kernel_paths = kernels,
  library_dir  = lib_dir,
  library_tag  = "nmath",
  dry_run      = TRUE
)
step1

# Step 2: expand transitive closure (small nmath library; runs on CRAN check)
tagged <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")
idx_small <- write_kernel_dependency_index(library_dir = lib_dir, write = FALSE)

step2_small <- attach_cross_library_tags(
  kernel_paths = tagged,
  library_dir  = lib_dir,
  depends_tag  = "depends_nmath",
  index       = idx_small,
  dry_run      = TRUE
)

```

```

nrow(step2_small)

# Step 2: full nmath (slow)
nmath_dir <- system.file("cl/nmath", package = "opencltools")
idx      <- write_kernel_dependency_index(library_dir = nmath_dir, write = FALSE)

step2 <- attach_cross_library_tags(
  kernel_paths = tagged,
  library_dir  = nmath_dir,
  depends_tag  = "depends_nmath",
  index        = idx,
  dry_run      = TRUE
)
step2

#####
## End of kernel_tagging_workflow example
#####

```

attach_kernel_call_tags

Attach Library Call Tags to Kernel Files

Description

Scans kernel .cl source files for calls to functions provided by a pre-annotated library, then writes the discovered dependencies as annotation tags directly into the kernel files.

Usage

```

attach_kernel_call_tags(
  kernel_paths,
  library_dir,
  library_tag,
  overwrite_existing = FALSE,
  dry_run = FALSE
)

```

Arguments

kernel_paths	Character vector of paths to kernel .cl files.
library_dir	Path to the pre-annotated library directory. Each .cl file in this directory must carry a @provides annotation listing the symbols it exports.
library_tag	String tag suffix used for annotation names, e.g. "nmath". Must not contain spaces or regex special characters.

`overwrite_existing` Logical; if FALSE (default), skip files that already carry a `@calls_<library_tag>` annotation. Set to TRUE to re-scan and overwrite (also clears any existing `@all_depends_<tag>` so [attach_cross_library_tags](#) re-computes it cleanly).

`dry_run` Logical; if TRUE, compute tags but do not write any files.

Details

For a library such as **nmathopencil**, each .cl shard carries a `@provides` annotation listing the symbols it defines. This function builds a **provides map** from those annotations (symbol → shard stem), scans each kernel's source (with comments and string literals stripped) for matching function calls, and writes four annotation tags at the top of each kernel file:

`@library_deps: <library_tag>` Library name (written if absent).

`@calls_<library_tag>: sym1, sym2` Symbols from the library actually called in this kernel.

`@depends_<library_tag>: stem1, stem2` Library shard stems that define the called symbols.

`@calls_opencil_builtin: sym | (none)` Detected OpenCL work-item and synchronization builtins (standard math builtins excluded).

Two-step tagging workflow:

```
# Step 1 – this function: infer direct library calls from source
nmath_dir <- system.file("cl/nmath_small", package = "openciltools")
attach_kernel_call_tags(
  kernel_paths = system.file("cl/src/dnorm_kernel.cl", package = "openciltools"),
  library_dir = nmath_dir,
  library_tag = "nmath"
)

# Step 2 – expand to full transitive closure
attach_cross_library_tags(
  kernel_paths = system.file("cl/src/dnorm_kernel.cl", package = "openciltools"),
  library_dir = nmath_dir,
  depends_tag = "depends_nmath"
)
```

Value

A data frame (returned invisibly) with one row per kernel file and columns:

`file` Basename of the kernel file.

`calls` Comma-separated library symbols detected in source.

`depends` Comma-separated shard stems for the detected symbols.

`opencil_builtins` Comma-separated OpenCL builtins detected, or empty string if none.

`changed` TRUE if the file was (or would be) modified. NA means the file was skipped (already tagged).

See Also

[attach_cross_library_tags](#), [attach_kernel_dependency_tags](#)

Examples

```
##### Start of kernel_tagging_workflow example #####

lib_dir <- system.file("cl/nmath_small", package = "opencltools")
kernels <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")

# Step 1: scan a launcher kernel for library calls (read-only dry run)
step1 <- attach_kernel_call_tags(
  kernel_paths = kernels,
  library_dir = lib_dir,
  library_tag = "nmath",
  dry_run = TRUE
)
step1

# Step 2: expand transitive closure (small nmath library; runs on CRAN check)
tagged <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")
idx_small <- write_kernel_dependency_index(library_dir = lib_dir, write = FALSE)

step2_small <- attach_cross_library_tags(
  kernel_paths = tagged,
  library_dir = lib_dir,
  depends_tag = "depends_nmath",
  index = idx_small,
  dry_run = TRUE
)
nrow(step2_small)

# Step 2: full nmath (slow)
nmath_dir <- system.file("cl/nmath", package = "opencltools")
idx <- write_kernel_dependency_index(library_dir = nmath_dir, write = FALSE)

step2 <- attach_cross_library_tags(
  kernel_paths = tagged,
  library_dir = nmath_dir,
  depends_tag = "depends_nmath",
  index = idx,
  dry_run = TRUE
)
step2

#####
## End of kernel_tagging_workflow example
#####
```

attach_kernel_dependency_tags

Attach Dependency Tags to a Dependency-Ordered Kernel Library

Description

Compute and attach derived tags to each .cl file in a library: @load_order, @all_depends, and @all_depends_count.

Usage

```
attach_kernel_dependency_tags(library_dir, dry_run = FALSE)
```

Arguments

library_dir	Directory containing .cl files with @depends tags.
dry_run	Logical; if TRUE, compute tags and reports without writing.

Details

Tags are written only if dependency sorting fully succeeds. If unresolved files remain, no files are modified and a cycle report is returned so source refactoring can be prioritized.

Value

A list with:

ok Logical success flag.

message Human-readable summary.

sorted Sorted records data frame.

unresolved Unresolved records data frame (empty on success).

cycles Cycle report data frame (empty on success).

tags Per-file tag data frame with source_origin, source_type, includes, depends, provides, load_order, all_depends, and all_depends_count (present on success).

header_functions Functions declared in header files without attribute_hidden, including declaring header, inferred definition file, declaration signature, define_alias, and all_depends for the definition file.

An S3 class "opencl_dependency_tags" is attached for use with `print()`.

Examples

```
##### Start of attach_kernel_dependency_tags example #####

lib_dir <- system.file("cl/nmath_small", package = "opencltools")
res <- attach_kernel_dependency_tags(lib_dir, dry_run = TRUE)
res$ok
print(res)

#####
## End of attach_kernel_dependency_tags example
#####
```

diagnose_glmbytes	<i>GPU and OpenCL Diagnostics for glmbytes</i>
-------------------	--

Description

A collection of tools for detecting GPU hardware, verifying OpenCL availability, checking driver installation, validating environment configuration, and diagnosing whether **glmbytes** can use GPU acceleration. These functions provide both high-level diagnostic summaries and low-level checks of system components such as PATH, library directories, OpenCL headers, and the ICD loader.

The diagnostic workflow is centered around `diagnose_glmbytes()`, which orchestrates all other checks and prints a detailed, human-readable report. Lower-level helpers can be called individually for programmatic inspection or automated testing.

Usage

```
diagnose_glmbytes()

detect_environment_and_gpus()

gpu_names()

detect_or_install_gpu_drivers(info)

detect_compute_runtimes(info)

has_opengl()

opengl_device_info(force = FALSE, details = FALSE)

opengl_fp64_available(force = FALSE)

opengl_reset_device_selection()

verify_opengl_runtime(lib_dirs = NULL)

check_runtime_env(runtime_info)
```

Arguments

info	A list returned by <code>detect_environment_and_gpus()</code> . The list must contain the following elements: environment One of "windows", "msys2", "linux", "wsl", or "unknown". nvidia A list with elements present (logical) and names (character). amd A list with elements present (logical) and names (character). intel A list with elements present (logical) and names (character).
force	If TRUE, rerun discovery even when a previous selection is cached.
details	If TRUE, include a candidates list describing every platform/device pair (extension flag and probe result per device).
lib_dirs	A list of OpenCL directories
runtime_info	The structured list returned by <code>detect_compute_runtimes()</code> .

Details

GPU acceleration speeds up embarrassingly parallel kernel workloads when a downstream package passes `use_opengl = TRUE` to its own OpenCL-backed entry points (for example envelope evaluation in **glmbytes**). OpenCL is **vendor-neutral** (NVIDIA, AMD, Intel); CPU-only builds remain valid when no OpenCL stack is present.

Practical setup (summary). Prebuilt binaries from CRAN or R-universe are often built **without** OpenCL GPU support; enabling the GPU path usually requires installing the consuming package **from source** on a machine with OpenCL **headers**, a linkable **OpenCL library / ICD loader**, and a working **vendor runtime** (GPU driver).

What this help page checks. A usable OpenCL environment requires:

1. OpenCL headers (e.g., `CL/cl.h`) at compile time,
2. the OpenCL ICD loader (e.g., `libOpenCL.so.1`) at runtime,
3. correct PATH and library search paths (especially on Linux/WSL),
4. a functional OpenCL platform and device (driver installed).

The functions here inspect these pieces. On Linux and WSL, `verify_opengl_runtime()` tries to create a platform, device, context, queue, and compile a minimal kernel. On Windows, that probe is skipped because platform-creation failures are often uninformative; rely on `diagnose_glmbytes()` and driver/runtime detection instead.

Start with `diagnose_glmbytes()` for a single readable report; use `has_opengl()` for a quick boolean when scripting.

Value

`diagnose_glmbytes()` Called for side effects (prints a report). Invisibly returns a named list with `environment_info`, `driver_status`, `runtime_status`, `env_diag`, `opengl_runtime_probe` (logical or NA), and `opengl_enabled` (logical or NA).

`detect_environment_and_gpus()` Named list: `environment` (character: windows, linux, wsl, msys2, or unknown); `nvidia`, `amd`, and `intel` each a list(present, names).

`gpu_names()` character GPU name(s) from `nvidia-smi` on Linux; on other platforms a one-element informational character vector.

`detect_or_install_gpu_drivers(info)` `list(environment, drivers)` with per-vendor installed, issues, and actions character vectors.

`detect_compute_runtimes(info)` `list(environment, runtimes)` with nested CUDA/OpenCL path lists and `headers_present`, `runtime_present`, and installed flags per vendor.

`check_runtime_env(runtime_info)` `list(environment, diagnostics)` comparing detected bin/lib dirs to current `PATH` and `LD_LIBRARY_PATH`: `found_path_dirs`, `missing_path_dirs`, `found_lib_dirs`, `missing_lib_dirs`, and `include_dirs` per vendor/runtime.

`has_opengl()` Logical scalar: TRUE if this **build** of **opencitools** was compiled with OpenCL (`USE_OPENGL`), not whether a GPU is usable.

`opengl_device_info(force, details)` `list` with `ok`, `reason`, `platform/device` indices and name strings, `extension_cl_khr_fp64`, `probe_fp64_ok`, `selection_policy`; optional candidates if `details = TRUE`. On CPU-only builds, `ok = FALSE` with reason "OpenCL not compiled in".

`opengl_fp64_available(force)` Logical scalar: TRUE if the cached device passes the `cl_khr_fp64` extension and compile probe.

`opengl_reset_device_selection()` No return value, called for side effects (clears the device-selection cache). Invisibly returns `NULL`.

`verify_opengl_runtime(lib_dirs)` Logical scalar: TRUE if a temporary program linked against `libOpenCL` and `clGetPlatformIDs` succeeded; FALSE if libraries or compile/run failed.

`get_opengl_core_count()` Integer scalar: total GPU compute units across platforms when OpenCL is compiled in; 1 on CPU-only builds (stub).

Functions

- `opengl_device_info()`: Cached OpenCL device selection for double-precision (`cl_khr_fp64`) kernels: enumerates platforms and devices, prefers GPU, checks the extension token, then verifies with a tiny `clBuildProgram` probe. Override with environment variables `NMATHOPENCL_PLATFORM_INDEX` and/or `NMATHOPENCL_DEVICE_INDEX` (0-based; device index is within the platform's device list). Use `opengl_reset_device_selection()` to clear the cache (e.g. after driver changes).
- `opengl_fp64_available()`: Returns TRUE if a cached OpenCL device passes the `cl_khr_fp64` extension check and build probe used for double kernels.
- `opengl_reset_device_selection()`: Clears the process-local OpenCL device selection cache so the next kernel or `opengl_device_info()` run re-enumerates devices.

High-level diagnostic

- `diagnose_glmbytes()` — full GPU/OpenCL diagnostic report.

Environment and hardware detection

- `detect_environment_and_gpus()` — detect OS and GPU vendor.
- `gpu_names()` — enumerate available GPU device names.
- `detect_compute_runtimes()` — detect CUDA/OpenCL runtimes.

OpenCL availability and runtime checks

- `has_opengl()` — quick check for OpenCL support.
- `opengl_fp64_available()`, `opengl_device_info()` — double-precision OpenCL device selection (cached probe).
- `verify_opengl_runtime()` — probe OpenCL platform/device availability.
- `check_runtime_env()` — validate PATH and library directories.

Driver installation helpers

- `detect_or_install_gpu_drivers()` — detect driver presence and issues.

PATH and library path utilities

These are optional helpers used by the diagnostic pipeline.

- `add_to_path_windows()`
- `add_to_path_linux()`
- `add_to_libpath_linux()`

See Also

[diagnose_glmbytes](#), [detect_environment_and_gpus](#), [detect_compute_runtimes](#), [verify_opengl_runtime](#), [has_opengl](#).

Examples

```
##### Start of gpu_diagnostics example #####

## Host-side inventory (no package USE_OPENGL .cpp)
info <- detect_environment_and_gpus()
info$environment
gpu_names()

runtime <- detect_compute_runtimes(info)
check_runtime_env(runtime)

## OpenCL device probes (stubs when !has_opengl(); full probes when NOT_CRAN)
## Inlined for CheckExEnv (unexported helpers are not visible during R CMD check).
if (!has_opengl() || identical(Sys.getenv("NOT_CRAN"), "true")) {
  opengl_fp64_available()
  opengl_device_info()
  get_opengl_core_count()
  verify_opengl_runtime()
} else {
  has_opengl()
}

# Full diagnostic report (prints; returns list invisibly)
diag <- diagnose_glmbytes()
names(diag)
```

```
#####
## End of gpu_diagnostics example
#####
```

extract_library_subset

Extract a Minimal Library Subset for a Set of Kernels

Description

Given one or more kernel .cl files and a library directory, determines the minimal set of library files required (union of all kernels' dependency annotations) and copies them — in dependency order — to a destination directory.

Usage

```
extract_library_subset(
  kernel_paths,
  library_dir,
  dest_dir,
  depends_tag = "all_depends",
  index = NULL,
  overwrite = FALSE
)
```

Arguments

kernel_paths	Character vector of paths to kernel .cl files. Each file is scanned for the annotation tag given by depends_tag.
library_dir	Path to the source library directory.
dest_dir	Path where files would be written. Must already exist for any copying to occur. If absent, a warning is issued, no directories are created, and nothing is copied; the returned data.frame still describes the planned subset (sources, intended destinations, copied = FALSE). In addition to the .cl files, kernel_dependency_index.rds is copied here when copying runs so the extracted subset can be used with a pre-loaded index.
depends_tag	Name of the annotation tag listing library file stems. Defaults to "all_depends". Pass "all_depends_nmath" for kernels that annotate their nmath dependencies with that tag.
index	Pre-loaded dependency index. If NULL, read from file.path(library_dir, "kernel_dependency_index.rds") with a message() nudging toward the recommended pattern.
overwrite	Logical; if FALSE (default) existing files in dest_dir are not overwritten — the copy is skipped and copied = FALSE in the returned data frame.

Details

Use this to populate a project-local copy containing only the library files actually needed by your kernels. The result can then be committed alongside your kernel files, removing a runtime dependency on the full library.

Bundled libraries such as `inst/cl/nmath` ship `kernel_dependency_index.rds` next to their `.cl` shards. Pass `index =` when you pass a pre-loaded index object to avoid redundant reads; regenerate the files with [write_kernel_dependency_index](#), for example after porting via `nmathtools/port_inst_cl_nmath_from_src` in the `openclport` package source tree.

```
lib_dir <- system.file("cl/nmath", package = "opencltools")
kernel_paths <- system.file(
  c("cl/src/dnorm_kernel.cl", "cl/src/pnorm_kernel.cl"),
  package = "opencltools"
)
dest_dir <- tempfile("ex_subset"); dir.create(dest_dir)
## on.exit(unlink(dest_dir, recursive = TRUE), add = TRUE)
result <- extract_library_subset(
  kernel_paths, lib_dir, dest_dir,
  depends_tag = "all_depends_nmath")
```

`extract_library_subset()` evaluates `inst/extdata/opencl_known_failures.json` against the union of `depends_tag` annotations and launcher paths.

Value

A `nmathopencl_lib_extract_df` subclass of `data.frame` with one row per library shard (`.cl` files in dependency order, followed by companion index files when planned or copied) and columns:

`stem` Stem name (filename without `.cl`), or index filenames.

`source` Full path to the source file under `library_dir`.

`dest` Intended destination path under `dest_dir`.

`copied` TRUE if copied; otherwise FALSE including when `dest_dir` is missing (`copied = FALSE` for every row), a source path is missing, or an existing destination was skipped.

See Also

[printing methods](#)

[load_library_for_kernel](#)

[write_kernel_dependency_index](#)

Other OpenCL kernel library subsets: [load_library_for_kernel\(\)](#), [load_library_for_kernel_cross_package\(\)](#), [write_kernel_dependency_index\(\)](#)

Examples

```
##### Start of extract_library_subset example #####
## Small library (fast; runs on CRAN check)
```

```

lib_small <- system.file("cl/nmath_small", package = "opencltools")
kpath <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")
idx_small <- write_kernel_dependency_index(library_dir = lib_small, write = FALSE)
dest_small <- file.path(tempdir(), "opencltools_extract_small")
if (dir.exists(dest_small)) unlink(dest_small, recursive = TRUE)
dir.create(dest_small, recursive = TRUE)
on.exit(unlink(dest_small, recursive = TRUE), add = TRUE)
df_small <- extract_library_subset(
  kpath, lib_small, dest_small,
  depends_tag = "all_depends_nmath",
  index = idx_small
)
sum(df_small$copied)

## Full nmath (slow)
lib_dir <- system.file("cl/nmath", package = "opencltools")
kernel_paths <- system.file(
  c("cl/src/dnorm_kernel.cl", "cl/src/pnorm_kernel.cl"),
  package = "opencltools"
)
idx <- write_kernel_dependency_index(library_dir = lib_dir, write = FALSE)
dest_dir <- file.path(tempdir(), "opencltools_extract_example")
if (dir.exists(dest_dir)) unlink(dest_dir, recursive = TRUE)
dir.create(dest_dir, recursive = TRUE)
on.exit(unlink(dest_dir, recursive = TRUE), add = TRUE)
df <- extract_library_subset(
  kernel_paths, lib_dir, dest_dir,
  depends_tag = "all_depends_nmath",
  index = idx
)
print(df)
sum(df$copied)

#####
## End of extract_library_subset example
#####

```

get_opengl_core_count *Get the number of available OpenCL compute units*

Description

Returns the number of compute units (cores) available on the default OpenCL device. This can be useful for diagnostics or performance tuning.

Usage

```
get_opengl_core_count()
```

Value

Integer scalar: total GPU compute units across OpenCL platforms when this build has OpenCL support; 1 on CPU-only builds. See [gpu_diagnostics](#) for full return details.

Examples

```
##### Start of gpu_diagnostics example #####

## Host-side inventory (no package USE_OPENCL .cpp)
info <- detect_environment_and_gpus()
info$environment
gpu_names()

runtime <- detect_compute_runtimes(info)
check_runtime_env(runtime)

## OpenCL device probes (stubs when !has_opengl(); full probes when NOT_CRAN)
## Inlined for CheckExEnv (unexported helpers are not visible during R CMD check).
if (!has_opengl() || identical(Sys.getenv("NOT_CRAN"), "true")) {
  opengl_fp64_available()
  opengl_device_info()
  get_opengl_core_count()
  verify_opengl_runtime()
} else {
  has_opengl()
}

# Full diagnostic report (prints; returns list invisibly)
diag <- diagnose_glmbyes()
names(diag)

#####
## End of gpu_diagnostics example
#####
```

kernel_lib_subset_printing

Printing results from minimal kernel-library subset loaders

Description

[load_library_for_kernel](#) returns a concatenated OpenCL-sources string with classes `nmathopengl_concatenated_lib`, extending character, with attributes listing requested and loaded stems in dependency order.

Usage

```
## S3 method for class 'nmathopengl_concatenated_lib'
print(x, ..., max_provides_symbols_per_shard = 80L)
```

```
## S3 method for class 'nmathopenc1_lib_extract_df'
print(x, ...)
```

Arguments

x From [load_library_for_kernel](#) or [extract_library_subset](#).

... Reserved; the extract data.frame printer does not forward to print.data.frame by default (use `print(as.data.frame(x), ...)` for a full preview).

max_provides_symbols_per_shard
For `print.nmathopenc1_concatenated_lib`: maximum `//@provides` symbols printed per algorithm-style shard (default 80). For shards `dpq`, `Rmath`, `nmath`, and `refactored`, only a one-line count is shown (header / internals-style `//@provides`). Use `NA_integer_` or a negative value for no cap on algorithm shards.

Details

[extract_library_subset](#) returns classes `nmathopenc1_lib_extract_df`, extending `data.frame`, with paths and tag metadata attached.

Value

Both S3 print methods are called for side effects. They write a structured summary to the console and invisibly return `x`.

`print.nmathopenc1_concatenated_lib` `x` is a character with class `nmathopenc1_concatenated_lib` (concatenated OpenCL sources from [load_library_for_kernel](#)). Printing lists requested/loaded library stems, kernel entry points, sampled `//@provides` symbols, and concatenated byte size—not the full source.

`print.nmathopenc1_lib_extract_df` `x` is a `data.frame` with class `nmathopenc1_lib_extract_df` (from [extract_library_subset](#)). Printing lists extraction/manifest rows and copy status per `stem.cl`.

Examples

```
##### Start of kernel_lib_subset_printing example #####

## Small library (fast; runs on CRAN check)
lib_small <- system.file("cl/nmath_small", package = "openc1tools")
kpath <- system.file("cl/src/dnorm_kernel.cl", package = "openc1tools")
idx_small <- write_kernel_dependency_index(library_dir = lib_small, write = FALSE)
src_small <- load_library_for_kernel(
  kpath, lib_small,
  depends_tag = "all_depends_nmath",
  index = idx_small
)
length(attr(src_small, "stems_loaded"))

## Full nmath (slow; verbose print)
```

```

lib_dir <- system.file("cl/nmath", package = "opencltools")
idx      <- write_kernel_dependency_index(library_dir = lib_dir, write = FALSE)

src <- load_library_for_kernel(
  kpath, lib_dir,
  depends_tag = "all_depends_nmath",
  index       = idx
)
print(src)

dest_dir <- file.path(tempdir(), "opencltools_subset_print_example")
if (dir.exists(dest_dir)) unlink(dest_dir, recursive = TRUE)
dir.create(dest_dir, recursive = TRUE)
on.exit(unlink(dest_dir, recursive = TRUE), add = TRUE)
df <- extract_library_subset(
  kpath, lib_dir, dest_dir,
  depends_tag = "all_depends_nmath",
  index       = idx
)
print(df)

#####
## End of kernel_lib_subset_printing example
#####

```

load_kernel_source	<i>Load OpenCL Kernel Source Files</i>
--------------------	--

Description

These functions provide a user-facing interface for loading OpenCL kernel source files and kernel libraries from the package's `cl/` directory. They call internal C++ routines that perform file lookup, dependency resolution, and concatenation of kernel sources.

Usage

```

load_kernel_source(relative_path, package = "opencltools")

load_kernel_library(subdir, package = "opencltools", verbose = FALSE)

```

Arguments

<code>relative_path</code>	A file path inside the package's <code>cl/</code> directory. Used by <code>load_kernel_source()</code> to load a single <code>.cl</code> file.
<code>package</code>	Name of the package containing the OpenCL sources. Defaults to "glmbytes".
<code>subdir</code>	A subdirectory inside <code>cl/</code> containing a set of <code>.cl</code> files annotated with <code>@provides</code> and <code>@depends</code> tags. Used by <code>load_kernel_library()</code> to construct a dependency-resolved kernel library.

verbose Logical; print diagnostic information during dependency resolution (default: FALSE).

Details

Reading and concatenating ‘.cl’ files does not require a GPU or OpenCL runtime. These functions work on all builds of **openclttools**. Use [has_opencil](#) to see whether the package was compiled with OpenCL device/runtime support for downstream GPU execution.

If no text could be read, an empty string is returned and a message is emitted.

Value

load_kernel_source() Character scalar (length 1) with the contents of one ‘.cl’ file (plain character, not an S3 class). Throws if the path is missing; returns "" with a message() if no text was read. Does not require [has_opencil](#).

load_kernel_library() Character scalar: all shards in subdir concatenated in dependency order (plain character). Same empty-string message() behavior. Use verbose = TRUE for dependency-sort diagnostics. For a subclass with stem metadata, see [load_library_for_kernel](#).

How These Functions Assemble an OpenCL Program

The functions load_kernel_source() and load_kernel_library() are the fundamental tools used by **glmbayes** to construct complete OpenCL programs from modular components. OpenCL kernels in this package are not stored as monolithic .cl files. Instead, they are built dynamically by concatenating several layers of source code, each serving a distinct purpose in the final GPU program.

A typical OpenCL program used by **glmbayes** is assembled in the following order:

1. **Global configuration header** The file `OPENCL.cl` defines global extensions, IEEE constants, helper macros, and device-side utilities. It plays a role analogous to a C/C++ header file and must appear at the very top of every combined kernel module. It enables features such as double-precision arithmetic (`cl_khr_fp64`) and device-side debugging (`cl_khr_printf`).
2. **Mathematical library modules** Subdirectories such as `"rmath"`, `"dpq"`, and `"nmath"` contain collections of .cl files implementing mathematical functions used throughout the GLM likelihood and gradient computations.
Each file may declare `@provides` and `@depends` tags. `load_kernel_library()` reads all files in a subdirectory, parses these annotations, performs a dependency-aware topological sort, and concatenates the files so upstream functions appear before downstream callers.
This mechanism parallels a sequence of `#include` statements in C/C++, but resolves dependencies automatically.
3. **Model-specific helper functions** Some kernels require additional device-side utilities that are not part of the shared libraries. These are typically loaded using `load_kernel_source()` and appended after the library modules.
4. **Final kernel entry function** Device kernels compile last.
Helper sources follow bundled `f2_f3_*` kernels.
Dispatch wrappers (like `f2_f3_opencil()`) stitch layers safely.

The resulting program is a single, syntactically valid OpenCL source string that is passed directly to the OpenCL compiler (e.g., via `clBuildProgram`). The ordering performed by `load_kernel_library()` is essential for successful compilation and ensures that the GPU kernels used by **glmbayes** are reproducible, modular, and maintainable.

Mirrors assembly order:\cr shards, sorted dependency libraries,
then kernels for GPU submits.

Preparing a Global Configuration Header

Most OpenCL programs begin with a small configuration header that enables device extensions, defines IEEE constants, and provides utility macros used throughout the kernel code. The file `OPENCL.cl` included in the examples illustrates one such header, but users are free to design their own.

A configuration header typically includes:

- **Extension declarations** (e.g. enable `cl_khr_fp64` with a short `#pragma OPENCL EXTENSION` line).
- **IEEE constants** (e.g., `ML_NAN`, `ML_POSINF`), which many statistical kernels rely on.
- **Utility macros** such as `INLINE` or `R_UNUSED` to improve readability and suppress warnings.
- **Optional helper definitions** such as work-item macros, typedefs, or device-side debugging tools.

This header should appear at the very top of every combined OpenCL program. The function `load_kernel_source()` is typically used to load it.

Library-Level Header Files

In addition to individual `.cl` files that define functions, most OpenCL libraries also require a *library-level header file*. This file contains constants, macros, error-handling definitions, and other shared utilities that apply to the entire library. It is conceptually similar to a C/C++ header such as `<math.h>` or `Rmath.h`.

A library-level header file:

- defines numerical constants (e.g., `ML_NAN`, `ML_POSINF`),
- provides OpenCL-safe versions of R's `mathlib` macros,
- stubs out error and warning hooks for device-side execution,
- defines validation macros such as `ISNAN` and `R_FINITE`,
- declares error codes and error-handling helpers, and
- provides any library-wide constants (e.g., `WILCOX_MAX`).

This file is typically named after the library itself (e.g., `"nmath.cl"`) and is placed in the same directory as the other library files. It should be loaded *before* any of the function-level files in the library, because those files may rely on macros or constants defined here.

A simplified example of such a header is shown below:

```

// nmath.cl - OpenCL math constants, macros & remaps for GPU kernels

// Stub out R's error/warning hooks for OpenCL
#ifndef MATHLIB_ERROR
    #define MATHLIB_ERROR(fmt, ...) /* no-op */
#endif

// Numerical constants
#define ML_POSINF (1.0 / 0.0)
#define ML_NEGINF (-1.0 / 0.0)
#define ML_NAN    (0.0 / 0.0)

// Error codes
#define ME_DOMAIN    1
#define ME_RANGE     2
#define ME_NOCONV    4
#define ME_PRECISION 8

// Validation macros
#define ISNAN(x)      (isnan(x))
#define R_FINITE(x)   (isfinite(x))

// Error-handling macros
#define ML_ERR_return_NAN \
    do { ML_ERROR(ME_DOMAIN, fname); return ML_NAN; } while(0)

// Library-wide constants
#define WILCOX_MAX 50

```

When `load_kernel_library()` processes a library directory, it first loads this header file (if present) and then loads the remaining files in dependency-correct order. This ensures that all macros and constants are available before any function-level code is compiled.

Library-level headers are optional but strongly recommended. They provide a clean, centralized place to define constants and macros that would otherwise need to be duplicated across multiple files. This structure is general and applies to any OpenCL project, not only to statistical or mathematical libraries.

Preparing Library Files

A library file is a single `.cl` source file containing one or more device-side functions. To allow `load_kernel_library()` to assemble these files in a dependency-correct order, each file should begin with a small annotation block describing what the file provides and what it depends on.

A typical library file begins with three comment lines:

```

// @provides: symbol1, symbol2, ...
// @depends:  fileA, fileB, ...
// @includes: library_name

```

@provides Comma-separated symbols defined here (resolver matches them to other files).
Helps map requirements to supplying stems.

@depends Comma-separated stems (omit .cl) listing *direct* prerequisites.
If file A calls B and B calls C, A lists only B (not transitive C).
Include the shared library header stem so macros load before bodies.

@includes Library name metadata (e.g., "nmath"); does *not* drive sort order.
load_kernel_library() prints diagnostics using this grouping.

The following example illustrates the recommended format:

```
// expm1.cl - OpenCL adaptation of expm1.c
// @provides: expm1
// @depends:  nmath, log1p
// @includes: nmath

#ifdef HAVE_EXPM1
double expm1(double x) {
    ...
}
#endif
```

The dependency resolver in load_kernel_library() reads the @provides and @depends tags, constructs a directed graph of file dependencies, and performs a topological sort to ensure that:

- all required files are loaded before they are referenced,
- files with no dependencies (typically the library header) are loaded first,
- files depending on others are loaded later, and
- circular or missing dependencies are detected and reported.

The @includes tag is not used for dependency resolution; it simply identifies the library grouping to which the file belongs.

This annotation format is general and applies to any OpenCL project. It allows users to port existing C/C++ mathematical libraries into OpenCL simply by translating the functions into .cl files and adding the appropriate @provides and @depends tags.

Dependency Resolution, Circular Logic, and Verbose Output

The load_kernel_library() function performs a dependency-aware topological sort of all .cl files in the specified library directory. Each file is promoted into the sorted load order only when all of the file names listed in its @depends tag have already been promoted. Files with an empty @depends list (typically the library header file) are promoted first.

If, during the sorting process, no additional files can be promoted, the function concludes that the remaining files have either:

- circular dependencies (e.g., file A depends on file B, and file B depends on file A), or
- missing dependencies (e.g., a file lists a dependency that does not correspond to any .cl file in the directory).

In such cases, the function throws a runtime error:

```
"Dependency sort failed: unresolved dependencies remain."
```

This error prevents the construction of an invalid or incomplete OpenCL program.

When `verbose = TRUE`, `load_kernel_library()` prints detailed diagnostic information describing each stage of the dependency-resolution process. The verbose output includes:

- the list of all `.cl` files discovered in the library,
- the initial set of files with no dependencies,
- the set of unsorted files and their dependency counts,
- each pass of the while-loop used for topological sorting,
- for each file, whether each dependency has already been satisfied,
- which files are promoted on each pass, and
- a final summary of the sorted load order.

If circular or missing dependencies are detected while `verbose = TRUE`, the function prints the list of files that could not be promoted before raising the error. This makes it straightforward for users to identify incorrect or incomplete `@depends` tags. In rare cases, files may need to be split or functions rewritten to eliminate genuine circular dependencies.

Writing Kernel Files

After preparing a configuration header and any required libraries, users typically write one or more OpenCL *kernel* files. A kernel is the entry point executed on the device and is usually designed for tasks that are embarrassingly parallel.

A kernel file should:

- begin with any required `#pragma OPENCL EXTENSION` lines,
- include any constants or local macros needed by the computation,
- define a `__kernel` void function that operates on global buffers, and
- avoid dependencies on host-side libraries (OpenCL C is a restricted subset of C99).

The example below illustrates a kernel that computes a quadratic form and gradient for each grid point in parallel. It demonstrates typical OpenCL idioms such as:

- using `get_global_id(0)` to index work-items,
- reading from `__global` buffers,
- accumulating results in local arrays, and
- writing results back to global memory.

Kernel files are usually loaded with `load_kernel_source()` and placed at the end of the assembled program so that all helper functions and library routines are defined before the kernel is compiled.

Kernel Runners and Kernel Wrappers

Although not strictly required, it is strongly recommended to separate OpenCL execution into two layers: a *kernel runner* and a *kernel wrapper*. This is the design used throughout the **glmbayes** implementation and provides a clean, modular structure for preparing inputs, launching OpenCL kernels, and post-processing results.

Kernel Runners: A kernel runner is a low-level C++ function that interacts directly with the OpenCL runtime. It is responsible for:

- selecting the OpenCL platform and device,
- creating the context and command queue,
- compiling the combined program source,
- creating device buffers and transferring data,
- setting kernel arguments,
- launching the kernel via `clEnqueueNDRangeKernel()`,
- reading results back to host memory, and
- releasing all OpenCL resources.

Kernel runners contain *no R-specific logic*. They operate entirely on flattened numeric arrays and primitive types. This makes them reusable across many kernels and easy to test in isolation.

In **glmbayes**, the function `f2_f3_kernel_runner()` is the primary example of a kernel runner. It takes fully prepared inputs, executes the OpenCL kernel, and returns raw numeric outputs.

Kernel Wrappers: A kernel wrapper is an R-facing function (typically exported via `[[Rcpp::export]]`) that prepares inputs for the runner and performs any necessary post-processing. A wrapper is responsible for:

- validating R inputs and extracting dimensions,
- flattening matrices and vectors into contiguous arrays,
- selecting the appropriate kernel name and kernel file based on model family and link function,
- assembling the full OpenCL program source by concatenating the core OpenCL header, library sources, and the model-specific kernel,
- invoking the kernel runner with the prepared inputs, and
- reshaping or converting the runner's outputs into R-friendly structures (e.g., `NumericVector`, `arma::mat`).

Kernel wrappers contain all R-level logic and none of the OpenCL plumbing. They provide a stable, user-facing API while delegating GPU execution to the runner.

In **glmbayes**, the function `f2_f3_openc1()` is the kernel wrapper corresponding to `f2_f3_kernel_runner()`. It flattens inputs, selects the correct kernel based on the GLM family and link, assembles the program source, calls the runner, and returns the results as R objects.

Why Separate Runners and Wrappers?: This two-layer design is recommended because it:

- isolates OpenCL resource management from R-level logic,
- makes kernel runners reusable across many models,
- simplifies debugging by separating data preparation from GPU execution,
- allows wrappers to evolve independently of the low-level runner,
- enables consistent program assembly across kernels, and

- keeps exported R functions clean, readable, and easy to maintain.

While users are free to design their own structure, adopting this pattern generally leads to clearer code and more maintainable OpenCL integrations.

Examples

```
##### Start of load_kernel_source example #####

src <- load_kernel_source("nmath/bd0.cl")
lib <- load_kernel_library("nmath")
nchar(src)
nchar(lib)
nchar(load_kernel_library("libR_shims"))

#####
## End of load_kernel_source example
#####
```

load_library_for_kernel

Load a Minimal OpenCL Library Subset for a Single Kernel

Description

Given a single kernel .cl file and a library directory (with an associated dependency index), reads the annotation tag that lists needed library files and returns their source code concatenated in the correct dependency order.

Usage

```
load_library_for_kernel(
  kernel_path,
  library_dir,
  depends_tag = "all_depends",
  index = NULL
)
```

Arguments

kernel_path	Path to a single .cl kernel file. The file is scanned for the annotation tag given by depends_tag.
library_dir	Path to the library directory containing the .cl source files (e.g. system.file("cl/nmath", package = "opencltools")).
depends_tag	Name of the annotation tag in the kernel file that lists the required library file stems. Defaults to "all_depends". For kernels annotated with @all_depends_nmath, pass depends_tag = "all_depends_nmath".
index	Optional RDS list; NULL triggers lazy reads.

Details

For repeated calls in R code, loading `kernel_dependency_index.rds` once and passing `index =` avoids redundant disk reads. The bundled `cl/nmath` directory ships `kernel_dependency_index.rds` beside the `.cl` files; use [write_kernel_dependency_index](#) to regenerate it after porting (for example via `nmathtools/port_inst_cl_nmath_from_src.R` in the `openclport` package source tree).

```
lib_dir <- system.file("cl/nmath", package = "opencltools")
kpath <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")
src <- load_library_for_kernel(
  kpath, lib_dir,
  depends_tag = "all_depends_nmath")
```

Subsets come only from each launcher file's transitive `@all_depends_nmath`-style annotations (use [attach_cross_library_tags](#) where dependency graphs span libraries). Deliberately loading every `.cl` under `cl/nmath` remains available via [load_kernel_library](#)(..., "nmath") where appropriate.

When `inst/extdata/opencl_known_failures.json` matches the launcher path or the declared / loaded stems, [warning](#)(...) points to fragile ports.

Value

A character vector subclass `nmathopencl_concatenated_lib` holding concatenated sources (often length 1; blank annotations yield length-zero concatenation). Attachments describe requested and loaded library stems, paths, and byte size; see [print methods](#).

See Also

[extract_library_subset](#)

[printing methods](#)

[write_kernel_dependency_index](#)

Other OpenCL kernel library subsets: [extract_library_subset\(\)](#), [load_library_for_kernel_cross_package\(\)](#), [write_kernel_dependency_index\(\)](#)

Examples

```
##### Start of load_library_for_kernel example #####

## Small library (fast; runs on CRAN check)
lib_small <- system.file("cl/nmath_small", package = "opencltools")
kpath <- system.file("cl/src/dnorm_kernel.cl", package = "opencltools")
idx_small <- write_kernel_dependency_index(library_dir = lib_small, write = FALSE)
src_small <- load_library_for_kernel(
  kpath, lib_small,
  depends_tag = "all_depends_nmath",
  index = idx_small
)
nzchar(src_small)
```

```
## Full nmath (slow; rebuilds index over all shards)
lib_dir <- system.file("cl/nmath", package = "opencltools")
idx <- write_kernel_dependency_index(library_dir = lib_dir, write = FALSE)
src <- load_library_for_kernel(
  kpath, lib_dir,
  depends_tag = "all_depends_nmath",
  index       = idx
)
print(src)
nzchar(src)

#####
## End of load_library_for_kernel example
#####
```

load_library_for_kernel_cross_package

Load a minimal OpenCL library subset across packages

Description

Cross-package variant of [load_library_for_kernel](#): the launcher kernel and the annotated library tree may live in different installed packages. Paths are resolved with `system.file("cl", ...)` on each package, matching the C++ helper `openclPort::load_library_for_kernel_cross_package()` used by **glmbayes** when assembling likelihood-subgradient programs.

Usage

```
load_library_for_kernel_cross_package(
  kernel_relative_path,
  kernel_package,
  library_subdir,
  library_package,
  depends_tag = "all_depends_nmath"
)
```

Arguments

kernel_relative_path	Path to the launcher .cl file relative to inst/cl/ in kernel_package (for example "src/f2_f3_poisson.cl").
kernel_package	Installed package containing the launcher kernel.
library_subdir	Library subdirectory relative to inst/cl/ in library_package (for example "nmath").
library_package	Installed package containing the annotated library tree and kernel_dependency_index.tsv.
depends_tag	Annotation tag scanned in the launcher kernel (for example "all_depends_nmath").

Value

A character vector with class `nmathopencl_concatenated_lib` holding concatenated library sources. Attributes record the resolved kernel and library paths, packages, dependency tag, and byte size.

See Also

[load_library_for_kernel](#), [load_program_preload](#), [load_kernel_source](#)

Other OpenCL kernel library subsets: [extract_library_subset\(\)](#), [load_library_for_kernel\(\)](#), [write_kernel_dependency_index\(\)](#)

Examples

```
##### Start of cross-package loader example #####

## Launcher kernel and library tree both live in openclttools (same API as
## cross-package use in downstream packages).
src <- load_library_for_kernel_cross_package(
  kernel_relative_path = "src/dnorm_kernel.cl",
  kernel_package = "openclttools",
  library_subdir = "nmath_small",
  library_package = "openclttools",
  depends_tag = "all_depends_nmath"
)
print(src)

## End of cross-package loader example
```

`load_program_preload` *Concatenate a program preload from a manifest*

Description

Loads the fixed OpenCL prelude listed in a program preload manifest: single `.cl` files via [load_kernel_source](#) and annotated library directories via [load_kernel_library](#). This mirrors the C++ helper `openclPort::load_program_preload` used by downstream packages such as **glmbayes** when assembling likelihood-subgradient programs.

Usage

```
load_program_preload(
  manifest_relative_path = "program_preload_manifest.tsv",
  source_package = "nmathopencl",
  verbose = FALSE,
  manifest = NULL
)
```

Arguments

manifest_relative_path	Path relative to inst/cl/ inside source_package.
source_package	Installed package that owns the manifest and prelude sources.
verbose	Passed to load_kernel_library for library rows.
manifest	Optional pre-validated manifest data . frame from read_program_preload_manifest . When supplied, assembly runs in R without re-reading the manifest from disk.

Value

A length-one character vector with class opencltools_program_preload holding the concatenated OpenCL C source text.

See Also

[read_program_preload_manifest](#), [load_library_for_kernel_cross_package](#), [load_kernel_source](#)

Other OpenCL program preload: [read_program_preload_manifest\(\)](#), [write_program_preload_manifest\(\)](#)

Examples

```
##### Start of load_program_preload example #####

manifest <- read_program_preload_manifest(source_package = "opencltools")
print(manifest)

preload <- load_program_preload(source_package = "opencltools")
cat("Preload bytes:", attr(preload, "nbytes_concatenated"), "\n")

## End of load_program_preload example
```

opencltoolsLdFlags *Linker flags for downstream LinkingTo: opencltools packages*

Description

Returns a PKG_LIBS fragment so a downstream package can resolve openclPort:: symbols from this package's shared library instead of duplicating C++ sources such as 'kernel_loader.cpp'.

Usage

```
opencltoolsLdFlags()
```

Details

Use together with [openclttools](#) headers (`#include <openclttools/opencltPort.h>`) and your own ‘configure’ logic for OpenCL SDK `-I` paths when `USE_OPENCL` is defined.

Typical ‘src/Makevars’ usage (Unix):

```
OPENCLTOOLS_LIBS = $(shell $(R_HOME)/bin/Rscript -e "openclttools::openclttoolsLdFlags()")
PKG_LIBS = $(OPENCLTOOLS_LIBS) $(PKG_LIBS)
```

On Windows, add the same `OPENCLTOOLS_LIBS` line to ‘src/Makevars.win’ (or merge into the ‘Makevars’ block written by ‘configure.win’).

Value

A character scalar of linker flags (`-L... -lopenclttools`).

Examples

```
##### Start of openclttoolsLdFlags example #####

flags <- openclttoolsLdFlags()
flags
grepl("-lopenclttools", flags)

#####
## End of openclttoolsLdFlags example
#####
```

port_to_opengl_configure

Port an existing static src/Makevars to use OpenCL configure scripts

Description

Migrates a package that already has a static committed `src/Makevars` (and optionally `src/Makevars.win`) to the configure-script pattern required for CRAN-safe OpenCL support.

The function renames the existing `src/Makevars` to `src/Makevars.in` (the maintained source template) and generates `configure` (Linux/macOS) and `configure.win` (Windows) scripts that read `src/Makevars.in` at R CMD INSTALL time, run OpenCL detection, and write the final `src/Makevars` with OpenCL flags merged in – or copied verbatim from `src/Makevars.in` for CPU-only builds.

The generated scripts **always succeed**: if no OpenCL SDK is found the package installs cleanly as CPU-only. This is the property that makes packages safe for CRAN submission on build machines without a GPU SDK.

For packages with no existing `src/Makevars`, use [use_opengl_configure](#) instead. This function is for *migrating* an existing static `Makevars`.

Usage

```
port_to_opengl_configure(path = ".", backup = TRUE, overwrite = FALSE)
```

Arguments

path	Character. Root directory of the target package. Defaults to the current working directory (" . ").
backup	Logical. If TRUE (default), rename src/Makevars to src/Makevars.in before writing the configure scripts. If FALSE, only the configure scripts are written (the existing src/Makevars is left in place).
overwrite	Logical. If TRUE, overwrite existing configure scripts. Defaults to FALSE.

Value

Invisibly returns a character vector of the file paths written.

src/Makevars.in workflow

After porting, **maintain** src/Makevars.in for your base build flags (OpenMP, **RcppParallel**, LAPACK, etc.). The configure script reads it at install time and appends (or omits) the OpenCL flags. The generated src/Makevars is a build artifact – add it to .gitignore and never commit it. To update base flags, edit src/Makevars.in and reinstall.

configure → USE_OPENCL → has_opengl()

```
configure / configure.win
  -> reads src/Makevars.in for base flags
  -> detects CL/cl.h + libOpenCL (+ runtime probe on Linux)
  -> writes -DUSE_OPENCL into Makevars (or copies .in verbatim)

#ifdef USE_OPENCL in C++ source
  -> guards all GPU code; package compiles cleanly either way

has_opengl() in R
  -> mirrors the compile-time flag; TRUE only if USE_OPENCL was set
```

Limitations

- += (append) assignments in src/Makevars are detected and trigger a warning; review the generated configure carefully.
- If src/Makevars looks like a generated file (contains absolute paths, -lOpenCL, or -DUSE_OPENCL), the function warns. Run on the static committed file, not a build artifact.
- Packages that already have configure or configure.win are refused unless overwrite = TRUE. Users with existing configure scripts should integrate the OpenCL block manually; see system.file("configure-templates", "README.md", package = "opengltools").

See Also

[use.openc1_configure](#) for packages without an existing src/Makevars. `vignette("Chapter-02", package = "nmathopenc1")` in **nmathopenc1** for a full guide when building on the ported nmath kernel library.

Examples

```
##### Start of port_to_openc1_configure example #####

tmp <- tempfile("port_openc1_pkg")
dir.create(tmp)
dir.create(file.path(tmp, "src"))
on.exit(unlink(tmp, recursive = TRUE), add = TRUE)

writeLines(
  c(
    "PKG_CXXFLAGS = $(SHLIB_OPENMP_CXXFLAGS)",
    "PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS)"
  ),
  file.path(tmp, "src", "Makevars")
)

written <- port_to_openc1_configure(path = tmp, backup = TRUE)
written
file.exists(file.path(tmp, "configure"))
file.exists(file.path(tmp, "src", "Makevars.in"))

## Regenerate configure scripts after editing Makevars.in (same temp tree)
written2 <- port_to_openc1_configure(path = tmp, backup = FALSE, overwrite = TRUE)
length(written2)

#####
## End of port_to_openc1_configure example
#####
```

```
print.openc1_dependency_tags
```

Print Dependency Tag Attachment Results

Description

Print output from [attach_kernel_dependency_tags](#).

Usage

```
## S3 method for class 'openc1_dependency_tags'
print(x, max_rows = 50, ...)
```

Arguments

x	Result of attach_kernel_dependency_tags .
max_rows	Maximum printed rows per table section.
...	Forwarded to <code>print.data.frame</code> .

Value

Called for side effects. Prints tag tables (and optional cycle/unresolved reports) from [attach_kernel_dependency_tags](#). Invisibly returns x, a list with S3 class "openc1_dependency_tags" (same structure as `attach_kernel_dependency_tags`, `ok`, `message`, `sorted`, `unresolved`, `cycles`, `tags`, ...).

Examples

```
##### Start of attach_kernel_dependency_tags example #####

lib_dir <- system.file("cl/nmath_small", package = "openc1tools")
res <- attach_kernel_dependency_tags(lib_dir, dry_run = TRUE)
res$ok
print(res)

#####
## End of attach_kernel_dependency_tags example
#####
```

```
read_program_preload_manifest
```

Read a program preload manifest

Description

Reads the tab-separated manifest that lists the fixed OpenCL prelude (single files and annotated library directories) in load order. Production manifests ship with downstream packages such as **nmathopenc1**; this package includes a small teaching manifest at `inst/cl/program_preload_manifest.tsv` for examples and tests.

Usage

```
read_program_preload_manifest(
  manifest_relative_path = "program_preload_manifest.tsv",
  source_package = "nmathopenc1",
  manifest_path = NULL
)
```

Arguments

`manifest_relative_path` Path relative to `inst/cl/` inside `source_package`. Defaults to `"program_preload_manifest.tsv"`.

`source_package` Installed package that owns the manifest and prelude sources. Defaults to `"nmathopenc1"`.

`manifest_path` Optional absolute path to a `.tsv` manifest. When supplied, `source_package` is still recorded in metadata but `system.file()` is not used.

Details

When a companion `program_preload_manifest.rds` exists beside the `.tsv` file (see [write_program_preload_manifest](#)), the RDS copy is preferred for repeated R-side reads.

Value

A `data.frame` with columns `rank`, `kind` (`"file"` or `"library"`), and `rel_path`, sorted by `rank`.

See Also

[load_program_preload](#), [write_program_preload_manifest](#)

Other OpenCL program preload: [load_program_preload\(\)](#), [write_program_preload_manifest\(\)](#)

Examples

```
##### Start of load_program_preload example #####

manifest <- read_program_preload_manifest(source_package = "openc1tools")
print(manifest)

preload <- load_program_preload(source_package = "openc1tools")
cat("Preload bytes:", attr(preload, "nbytes_concatenated"), "\n")

## End of load_program_preload example
```

stage_kernel_dependency_sort

Stage Kernel Library Dependency Sort Results

Description

Run the file-level @depends sort without requiring full success. Files that can be sorted are copied in order, and files blocked by unresolved dependencies are copied into a separate folder with CSV reports.

Usage

```
stage_kernel_dependency_sort(library_dir, output_dir, overwrite = FALSE)
```

Arguments

library_dir Directory containing .cl files with @depends tags.
 output_dir Directory where sorted and unresolved files/reports should be written.
 overwrite Logical; remove and recreate output_dir if it exists.

Value

A named list with:

sorted Data frame of files placed in dependency order (order, pass, file, source_type, depends).

unresolved Data frame of files blocked by missing dependencies.

sorted_dir, unresolved_dir Output directory paths; CSV reports 'sorted_files.csv' and 'unresolved_files.csv' are written under output_dir.

Examples

```
##### Start of stage_kernel_dependency_sort example #####

lib_dir    <- system.file("cl/nmath_small", package = "opengltools")
output_dir <- tempfile("stage_sort")
on.exit(unlink(output_dir, recursive = TRUE), add = TRUE)

res <- stage_kernel_dependency_sort(lib_dir, output_dir, overwrite = TRUE)
nrow(res$sorted)
length(list.files(output_dir, recursive = TRUE))

#####
## End of stage_kernel_dependency_sort example
#####
```

use_opengl_configure *Set up OpenCL configure scripts in a downstream R package*

Description

Copies generic OpenCL configure and configure.win scripts to the root directory of a package. The scripts detect CL/cl.h and libOpenCL at compile time and generate src/Makevars (Linux / macOS) or src/Makevars.win (Windows) with or without -DUSE_OPENCL, depending on what is found.

The scripts **always succeed**. When no OpenCL SDK is present they produce a CPU-only Makevars with no -lOpenCL. This is the key property that makes packages safe for CRAN submission without requiring a GPU SDK on the build machine.

Usage

```
use_opengl_configure(path = ".", overwrite = FALSE)
```

Arguments

path	Character. Root directory of the target package. Defaults to the current working directory (".").
overwrite	Logical. If TRUE, overwrite existing configure scripts. Defaults to FALSE to avoid accidentally replacing a customized script.

Value

Invisibly returns a character vector of the file paths that were written (empty if all files were skipped).

Why configure scripts are necessary

A package that references `-lOpenCL` or `CL/cl.h` in a static `src/Makevars` will **fail to compile** on 'CRAN' build machines (which have no GPU SDK installed), and no binary will be produced. The configure scripts here avoid this by probing for the SDK at install time and falling back to a CPU-only build when it is absent. The relationship is:

```
configure / configure.win
-> detects CL/cl.h + libOpenCL
-> writes -DUSE_OPENCL into Makevars    (or omits it)

#ifdef USE_OPENCL in C++ source
-> guards all GPU code; package compiles cleanly either way

has_opengl() in R
-> mirrors the compile-time flag; returns TRUE only if USE_OPENCL was set
```

See Also

[port_to_opengl_configure](#) for packages with an existing static `src/Makevars`. [opengltoolsLdFlags](#) for linking against this package from C++. `vignette("Chapter-02", package = "nmathopengl")` in **nmathopengl** for a full downstream package guide when using the ported nmath kernel library. Template source: `system.file("configure-templates", package = "opengltools")`.

Examples

```
##### Start of use_opengl_configure example #####

tmp <- tempfile("use_opengl_pkg")
dir.create(tmp)
on.exit(unlink(tmp, recursive = TRUE), add = TRUE)

written <- use_opengl_configure(path = tmp)
written
file.exists(file.path(tmp, "configure"))
file.exists(file.path(tmp, "configure.win"))

## Overwrite path (same temp tree; safe when run.dontest = TRUE)
written2 <- use_opengl_configure(path = tmp, overwrite = TRUE)
```

```
length(written2)

#####
## End of use_opengl_configure example
#####
```

```
write_kernel_dependency_index
```

Build and save a kernel dependency index

Description

Writes two companion index files next to the .cl files in the kernel library directory:

Usage

```
write_kernel_dependency_index(
  library_dir = NULL,
  tags = NULL,
  output_path = NULL,
  write = TRUE,
  verbose = FALSE
)
```

Arguments

library_dir	Library root with annotated .cl files. When tags is supplied, must agree with tags\$library_dir.
tags	Optional attachment result from attach_kernel_dependency_tags . Must have ok = TRUE when reused to skip resorting.
output_path	Path for the RDS file. Defaults to file.path(<library_dir>, "kernel_dependency_index.rds"). The .tsv file is always written to the same directory with the .tsv extension.
write	If FALSE, builds the index object and returns it without writing either file.
verbose	If TRUE, emits short messages with the output paths.

Details

- RDS helper shard for loaders such as [load_library_for_kernel](#).
- Tab-separated (.tsv) stem map for C++ (rows stem<TAB>dependencies, pre-sorted).

Both files encode the same information and are always written together so they remain in sync.

Value

Invisibly, the index `list()` (version schema):

- version: integer schema version.
- generated_at: timestamp from `Sys.time()`.
- library_dir, library_name: resolved library path / basename.
- stems_ordered: stems in global load order.
- load_order: named integer vector stem -> rank.
- depends: named list stem -> character() (direct @depends).
- all_depends: named list stem -> character() (transitive dependencies, order consistent with global load order).
- n_files: file count.

See Also

[load_library_for_kernel](#)

[extract_library_subset](#)

Other OpenCL kernel library subsets: [extract_library_subset\(\)](#), [load_library_for_kernel\(\)](#), [load_library_for_kernel_cross_package\(\)](#)

Examples

```
##### Start of write_kernel_dependency_index example #####

lib_dir <- system.file("cl/nmath_small", package = "opencltools")
idx <- write_kernel_dependency_index(library_dir = lib_dir, write = FALSE)
names(idx)
length(idx$stems_ordered)
idx$n_files

#####
## End of write_kernel_dependency_index example
#####
```

```
write_program_preload_manifest
```

Build and save a program preload manifest

Description

Writes companion manifest files next to the prelude sources under `inst/cl/`:

Usage

```
write_program_preload_manifest(
  manifest = NULL,
  manifest_path = NULL,
  source_package = "nmathopenc1",
  manifest_relative_path = "program_preload_manifest.tsv",
  write = TRUE,
  verbose = FALSE
)
```

Arguments

<code>manifest</code>	Optional manifest data.frame. When NULL, the function reads <code>manifest_path</code> or resolves <code>system.file("cl", manifest_relative_path, package = source_package)</code> .
<code>manifest_path</code>	Optional path to an existing .tsv manifest.
<code>source_package</code>	Package name recorded in RDS metadata.
<code>manifest_relative_path</code>	Relative path under <code>inst/cl/</code> used when resolving the default manifest location.
<code>write</code>	If FALSE, validates and returns the manifest object without writing files.
<code>verbose</code>	If TRUE, reports output paths.

Details

- tab-separated `program_preload_manifest.tsv` for C++ loaders;
- `program_preload_manifest.rds` for fast R-side reuse.

Value

Invisibly, a `list()` with `schema version = 1L`, `generated_at`, `source_package`, `manifest_relative_path`, `manifest_path`, and `entries`.

See Also

[read_program_preload_manifest](#), [load_program_preload](#)

Other OpenCL program preload: [load_program_preload\(\)](#), [read_program_preload_manifest\(\)](#)

Examples

```
## write_program_preload_manifest example

manifest <- read_program_preload_manifest(source_package = "opencltools")
obj <- write_program_preload_manifest(
  manifest = manifest,
  source_package = "opencltools",
  write = FALSE
)
names(obj)
```

```
nrow(obj$entries)
```

```
## End
```

Index

- * **OpenCL kernel library subsets**
 - extract_library_subset, [15](#)
 - load_library_for_kernel, [27](#)
 - load_library_for_kernel_cross_package, [29](#)
 - write_kernel_dependency_index, [39](#)
- * **OpenCL program preload**
 - load_program_preload, [30](#)
 - read_program_preload_manifest, [35](#)
 - write_program_preload_manifest, [40](#)
- * **diagnostics**
 - diagnose_glbayes, [11](#)
- * **environment**
 - diagnose_glbayes, [11](#)
- * **gpu**
 - diagnose_glbayes, [11](#)
- * **opengl**
 - diagnose_glbayes, [11](#)
- add_to_libpath_linux (add_to_path), [4](#)
- add_to_path, [4](#)
- add_to_path_linux (add_to_path), [4](#)
- add_to_path_windows (add_to_path), [4](#)
- attach_cross_library_tags, [5](#), [8](#), [9](#), [28](#)
- attach_kernel_call_tags, [7](#)
- attach_kernel_dependency_tags, [5](#), [6](#), [9](#), [10](#), [34](#), [35](#), [39](#)
- check_runtime_env (diagnose_glbayes), [11](#)
- detect_compute_runtimes, [14](#)
- detect_compute_runtimes (diagnose_glbayes), [11](#)
- detect_environment_and_gpus, [14](#)
- detect_environment_and_gpus (diagnose_glbayes), [11](#)
- detect_or_install_gpu_drivers (diagnose_glbayes), [11](#)
- diagnose_glbayes, [3](#), [11](#), [12](#), [14](#)
- extract_library_subset, [3](#), [15](#), [19](#), [28](#), [30](#), [40](#)
- get_opengl_core_count, [17](#)
- gpu_diagnostics, [18](#)
- gpu_diagnostics (diagnose_glbayes), [11](#)
- gpu_names (diagnose_glbayes), [11](#)
- has_opengl, [3](#), [12](#), [14](#), [21](#)
- has_opengl (diagnose_glbayes), [11](#)
- kernel_lib_subset_printing, [18](#)
- load_kernel_library, [3](#), [28](#), [30](#), [31](#)
- load_kernel_library (load_kernel_source), [20](#)
- load_kernel_source, [3](#), [20](#), [30](#), [31](#)
- load_library_for_kernel, [3](#), [6](#), [16](#), [18](#), [19](#), [21](#), [27](#), [29](#), [30](#), [39](#), [40](#)
- load_library_for_kernel_cross_package, [16](#), [28](#), [29](#), [31](#), [40](#)
- load_program_preload, [30](#), [30](#), [36](#), [41](#)
- opengl_device_info, [3](#), [13](#)
- opengl_device_info (diagnose_glbayes), [11](#)
- opengl_fp64_available (diagnose_glbayes), [11](#)
- opengl_reset_device_selection, [13](#)
- opengl_reset_device_selection (diagnose_glbayes), [11](#)
- opengltools, [32](#)
- opengltools (opengltools-package), [3](#)
- opengltools-package, [3](#)
- opengltoolsLdFlags, [31](#), [38](#)
- port_to_opengl_configure, [3](#), [32](#), [38](#)
- print methods, [28](#)
- print(), [10](#)

`print.nmathopencl_concatenated_lib`
 (`kernel_lib_subset_printing`),
 18

`print.nmathopencl_lib_extract_df`
 (`kernel_lib_subset_printing`),
 18

`print.opencl_dependency_tags`, 34

printing methods, 16, 28

`read_program_preload_manifest`, 31, 35,
 41

`stage_kernel_dependency_sort`, 36

`Sys.getenv`, 4

`Sys.setenv`, 4

`Sys.time()`, 40

`use_opencl_configure`, 3, 32, 34, 37

`verify_opencl_runtime`, 3, 14

`verify_opencl_runtime`
 (`diagnose_glmbyes`), 11

warning, 28

`write_kernel_dependency_index`, 5, 6, 16,
 28, 30, 39

`write_program_preload_manifest`, 31, 36,
 40