

Package ‘ardea’

October 9, 2026

Type Package

Title Infrastructure for Interacting with GPUs

Version 0.1.1

Description Compiling, managing, and dispatching functions to GPUs. Will compile tooling for successfully detected frameworks, currently limited to: 'OpenCL' (<<https://www.khronos.org/opencl/resources>>), 'CUDA' (<<https://docs.nvidia.com/cuda/>>), and 'Metal' (<<https://developer.apple.com/documentation/metal>>).

Imports methods

Suggests knitr, rmarkdown

VignetteBuilder knitr

License GPL-3

SystemRequirements OpenCL, GNU make

Depends R (>= 4.5.0)

OS_type unix

NeedsCompilation yes

ByteCompile true

Encoding UTF-8

Author Nicholas Cooley [aut, cre] (ORCID:
<<https://orcid.org/0000-0002-6029-304X>>)

Maintainer Nicholas Cooley <npcooley@gmail.com>

URL <https://github.com/npcooley/ardea>

BugReports <https://github.com/npcooley/ardea/issues>

Repository CRAN

Date/Publication 2026-10-09 06:30:02 UTC

Contents

cuda_devices_exist	2
cuda_device_information	3
cuda_is_available	4
cuda_make_context	5
cuda_make_kernelptr	6
cuda_make_program	8
metal_compiler_is_available	9
metal_devices_exist	11
metal_device_information	11
metal_is_available	13
metal_make_context	14
metal_make_kernelptr	15
metal_make_program	16
opengl_devices_exist	19
opengl_device_information	20
opengl_is_available	21
opengl_make_context	22
opengl_make_kernelptr	23
opengl_make_program	24
simple_wrapper	26
supported_types_by_context	32
Index	34

cuda_devices_exist	<i>Check For Metal Devices</i>
--------------------	--------------------------------

Description

Report whether at least one CUDA compliant device is available on the current system.

Usage

```
cuda_devices_exist()
```

Details

This function is a thin wrapper for the C function `cuda_exposed_device_count`.

Value

A single logical value: TRUE if one or more CUDA devices were found, FALSE if no devices were discovered.

See Also

[cuda_is_available](#) [cuda_device_information](#)

Examples

```
if (cuda_is_available()) {
    cuda_devices_exist()
}
```

cuda_device_information

List Available CUDA Devices and Their Properties

Description

Enumerate all visible CUDA devices on the current system. Returns a list with one entry per device describing the device's properties.

Usage

```
cuda_device_information()
```

Details

All devices visible to `cudaGetDeviceCount()/cudaGetDeviceProperties()` on the current system are enumerated.

`cuda_is_available` is available as a sentinel function that returns TRUE if this package's configure script was able to successfully compile and link a CUDA test program.

Unlike `metal_device_information`, there is no `device_ptr` field here, and no `is_default` field – CUDA identifies devices by a plain integer ordinal rather than an opaque handle, and has no inherent default-device concept the way `MTLCreateSystemDefaultDevice()` does (`cudaGetDevice()` only reports whichever device happens to be current at the moment of the call, which is arbitrary at enumeration time, not a hardware property worth reporting).

Value

When available devices are present, an object of class `c("device_list", "list")`, where each element is itself of class `c("alternative_device", "list")` and describes a single CUDA compliant device with the following components:

`index` Integer: The device's ordinal, starting at 0, as reported by `cudaGetDeviceCount()`. Identical to `device_index`.

`device_index` Integer: The device ordinal used to select this device elsewhere in the package, e.g. by `cuda_make_context`. Identical to `index`, provided under this name for consistency with the `device_ptr`-style field other frameworks expose.

`name` Character: The device name, see `cudaDeviceProp::name`.

`total_global_mem_bytes` Numeric: Total global memory size in bytes, see `cudaDeviceProp::totalGlobalMem`.

`multiprocessor_count` Integer: The number of streaming multiprocessors on the device, see `cudaDeviceProp::multiProcessorCount`.

`max_threads_per_block` Integer: The device-level ceiling on threads per block, see `cudaDeviceProp::maxThreadsPerBlock`. This is a *per-device* value; the *per-kernel* ceiling actually used for dispatch is queried and cached separately by `cuda_make_kernelptr`.

`warp_size` Integer: The device's warp size, see `cudaDeviceProp::warpSize`.

`compute_capability_major` Integer: Major compute capability version, see `cudaDeviceProp::major`.

`compute_capability_minor` Integer: Minor compute capability version, see `cudaDeviceProp::minor`.

`framework` Character: In the case of this function, just populated with `cuda`.

`type` Character: Always GPU for CUDA devices.

`max_compute_units` Integer: Populated from `multiprocessor_count`, provided under this name for consistency with the field `opencl_device_information/metal_device_information` expose.

If no CUDA compliant devices are found, this function returns `NULL`.

Note

This R function is a relatively thin wrapper for the C-level `cuda_available_devices`. If no CUDA devices are detected on the system, i.e. `cuda_is_available` returns `FALSE`, this function returns `NULL`.

See Also

`cuda_is_available`, `cuda_make_context`

Examples

```
devices <- cuda_device_information()
str(devices)
```

<code>cuda_is_available</code>	<i>Check Whether CUDA Support Was Compiled Into This Build</i>
--------------------------------	--

Description

Report whether ardea was compiled with CUDA support on this system.

Usage

```
cuda_is_available()
```

Details

This function is an R wrapper for ardea's C function `cuda_sentinel`. Like `metal_is_available`, and unlike `opencl_is_available`, this reflects a compile-time decision made once, during package installation.

A `TRUE` result would mean the functions in ardea that depend on CUDA were compiled and are callable; it would not by itself guarantee that a CUDA-compliant GPU is currently attached. Until CUDA detection is implemented, this function exists so that code checking for CUDA support has a stable interface to call, without needing to change once CUDA support is added.

Value

Logical: TRUE if CUDA support was detected and compiled in at installation time, FALSE otherwise. CUDA detection is not yet implemented in ardea's configure script, so this function currently returns FALSE unconditionally, on every platform.

See Also

[opencl_is_available](#), [metal_is_available](#)

Examples

```
cuda_is_available()
```

cuda_make_context	<i>Create a CUDA Context for a Device</i>
-------------------	---

Description

Construct a CUDA context for a specific device, given a device entry returned by [cuda_device_information](#).

Usage

```
cuda_make_context(device,
                  use_default_stream = TRUE)
```

Arguments

device	alternative_device: a single device entry from the list created by cuda_device_information. Specifically required device properties currently include: - framework, must be cuda in this case - device_index, an integer ordinal for the device (0..cudaGetDeviceCount()-1)
use_default_stream	logical: a single non-NA value. If TRUE (the default), CUDA's implicit default stream is used. If FALSE, an explicit stream is created and owned by the returned context, released when it is garbage collected.

Details

Unlike [metal_make_context/opencl_make_context](#), there is no external device pointer because CUDA identifies devices by a plain integer ordinal. device\$device_index fills the same structural role device\$device_ptr fills for the other two frameworks. Taking an alternative_device object here (rather than a bare integer) is a deliberate choice to keep the calling convention consistent across all supported frameworks; the index is re-validated against a live device count on the C side regardless, since nothing guarantees the supplied object isn't stale (e.g. built in an earlier session, or a device reconfigured since).

CUDA has no object that "is" a context the way cl_context/MTLDevice do – cudaSetDevice() sets which GPU is current for the calling thread, and every subsequent CUDA call to a given context

implicitly targets whatever was last set. The object returned here is a thin wrapper around that device index (and, optionally, a stream); every ardea function that consumes it re-activates the correct device internally before doing any work.

Value

An external pointer of class "cuda_context". This is consumed by other ardea functions that require a context, such as [cuda_make_program](#) and [cuda_make_kernelptr](#).

See Also

[cuda_device_information](#), [cuda_make_program](#), [cuda_make_kernelptr](#)

Examples

```
if (cuda_is_available() &
    cuda_devices_exist()) {
  devices <- cuda_device_information()
  ctx <- cuda_make_context(devices[[1]])
}
```

cuda_make_kernelptr *Create Dispatch-Ready Kernel Pointers from a CUDA Module*

Description

Given a loaded CUDA module and the context it should be paired with, extract each named kernel function and return them as a named list of external pointers.

Usage

```
cuda_make_kernelptr(program,
                    context,
                    kernel_names)
```

Arguments

program	cuda_module: a module created by cuda_make_program .
context	cuda_context: the context created by cuda_make_context , used to determine which device the kernel(s) should be associated with. Should be the same context (or at least the same underlying device) that program was compiled/loaded with.
kernel_names	character: a vector of length 1 or greater, naming <code>__global__</code> functions declared in the .cu source program was compiled from.

Details

Unlike [metal_make_kernelptr](#), the underlying CUfunction handle has no reference counting at all – unloading its parent module instantly invalidates every function derived from it, with no protection from the CUDA API itself. To guard against this, each kernel pointer returned here carries its parent program object as an attribute, which keeps program (and therefore its module) reachable, and thus un-collectable, for as long as any kernel built from it is still reachable – even if the caller does not keep their own separate reference to program.

Each kernel’s maximum threads-per-block is queried once, at creation time, and cached – this is a *per-function* limit, not the per-device limit [cuda_device_information](#) reports, and is not re-queried on every dispatch.

The returned list is named using kernel_names, in the order supplied.

Value

A named list of external pointers of class "cuda_kernel", one per element of kernel_names.

See Also

[cuda_make_context](#), [cuda_make_program](#)

Examples

```
cuda_mm_naive <- '
extern "C" __global__ void cuda_mm_naive(float* output,
                                         const long long M,
                                         const long long K,
                                         const long long N,
                                         const float* A,
                                         const float* B)
{
    long long row = blockIdx.x * blockDim.x + threadIdx.x;
    long long col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (long long k = 0; k < K; k++) {
            sum += A[row + k * M] * B[k + col * K];
        }
        output[row + col * M] = sum;
    }
}
'

tmp_cu <- tempfile(fileext = ".cu")
writeLines(text = cuda_mm_naive,
           con = tmp_cu)

if (cuda_is_available() &
    cuda_devices_exist()) {
    devices <- cuda_device_information()
```

```

ctx <- cuda_make_context(devices[[1]])
mod <- cuda_make_program(tmp_cu, ctx)
kernels <- cuda_make_kernelptr(mod, ctx, c("cuda_mm_naive"))
}

```

code cuda_make_program

Compile a CUDA Source File into a Loaded Module

Description

Compile a .cu source file to PTX via nvcc, and load the result against a [cuda_make_context](#) context, returning it as an external pointer ready to be passed to [cuda_make_kernelptr](#).

Usage

```

cuda_make_program(cuda_file,
                  context,
                  ptx_file = NULL,
                  nvcc_flags = NULL)

```

Arguments

cuda_file	character: a length-1 path to a .cu source file. Must exist.
context	cuda_context: a context created by cuda_make_context .
ptx_file	character or NULL: a length-1 output path for the compiled .ptx file. If NULL (the default), a temporary file is created via <code>tempfile(fileext = ".ptx")</code> .
nvcc_flags	character or NULL: additional flags passed through to nvcc. Each element is treated as a single shell word, matching <code>system2()</code> 's own args convention – a flag that takes a value, e.g. <code>-I</code> , must be supplied as two separate elements (<code>c("-I", "/path/to/headers")</code>), not one space-joined string. May not include <code>-o</code> , since this function already supplies that flag from <code>ptx_file</code> .

Details

Unlike [metal_make_program](#), there is no in-process runtime-compile path here – no NVRTC integration currently exists, so this always shells out to nvcc to produce a .ptx file (`nvcc <cuda_file> -ptx -o <ptx_file>`), then loads it via `cuModuleLoad()`. This requires that nvcc is resolvable in the PATH of the R session (or otherwise locatable via `Sys.which("nvcc")`), which in turn requires a working CUDA toolkit install on the machine running R – a stricter requirement than a working CUDA driver alone.

Arguments are passed to `system2()` as individually `shQuote()`'d elements of a vector, not a single pasted string – `system2()` still joins them with spaces internally before the shell sees the line, so `shQuote()` on each path-bearing element is what actually protects against spaces or shell metacharacters, not the vector structure by itself. The nvcc exit status is checked explicitly, and the expected .ptx file's existence is verified before this function ever calls into C.

Value

An external pointer of class "cuda_module" wrapping the loaded module.

See Also

[cuda_make_context](#), [cuda_make_kernelptr](#)

Examples

```
cuda_mm_naive <- '
extern "C" __global__ void cuda_mm_naive(float* output,
                                         const long long M,
                                         const long long K,
                                         const long long N,
                                         const float* A,
                                         const float* B)
{
    long long row = blockIdx.x * blockDim.x + threadIdx.x;
    long long col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (long long k = 0; k < K; k++) {
            sum += A[row + k * M] * B[k + col * K];
        }
        output[row + col * M] = sum;
    }
}
'

tmp_cu <- tempfile(fileext = ".cu")
writeLines(text = cuda_mm_naive,
           con = tmp_cu)

if (cuda_is_available() &
    cuda_devices_exist()) {
    devices <- cuda_device_information()
    ctx <- cuda_make_context(devices[[1]])

    mod <- cuda_make_program(tmp_cu, ctx)
}
```

metal_compiler_is_available

Check Whether the Offline Metal Compiler Is Installed

Description

Report whether the offline Metal compiler (xcrun metal) can be run on this system. This compiler is required only when [metal_make_program](#) is called with a non-NULL metallib_file.

Usage

```
metal_compiler_is_available()
```

Details

ardea offers two ways to compile Metal source with [metal_make_program](#):

- With `metallib_file = NULL` (the default), source is compiled at runtime through the Metal API. This uses the shader compiler built into macOS and does not need any developer tools.
- With `metallib_file` supplied, source is compiled ahead of time by `xcrun metal` into a `.metallib` file. The metal compiler ships with full Xcode only. It is not part of the Xcode Command Line Tools, and recent Xcode versions provide it as a separate Metal Toolchain component that can be installed with `xcodebuild -downloadComponent MetalToolchain`.

Because of this, a system can return `TRUE` from both [metal_is_available](#) and [metal_devices_exist](#) while still returning `FALSE` here. Call this function before supplying `metallib_file` to [metal_make_program](#).

The check runs the compiler rather than only searching for it, because some Xcode versions include a metal placeholder that exists on disk but fails when the Metal Toolchain is not installed. The compiler's output is suppressed. Unlike [metal_is_available](#), which reflects a decision made when the package was installed, this check is performed live each time the function is called.

Value

A single logical value: `TRUE` if `xcrun -sdk macosx metal --version` ran and exited with status 0, `FALSE` otherwise. `FALSE` is always returned on non-macOS platforms, and when `xcrun` cannot be found on the `PATH`.

See Also

[metal_make_program](#), [metal_is_available](#), [metal_devices_exist](#)

Examples

```
metal_compiler_is_available()

if (metal_is_available() &&
    metal_devices_exist() &&
    metal_compiler_is_available()) {
  print("ahead-of-time Metal compilation via 'metallib_file' is supported")
}
```

metal_devices_exist	<i>Check For Metal Devices</i>
---------------------	--------------------------------

Description

Report whether at least one Metal compliant device is available on the current system.

Usage

```
metal_devices_exist()
```

Details

This function is a thin wrapper for the C function `exposed_metal_device_count`.

Value

A single logical value: TRUE if one or more Metal devices were found, FALSE if no devices were discovered.

See Also

[metal_is_available](#) [metal_device_information](#)

Examples

```
if (metal_is_available()) {  
    metal_devices_exist()  
}
```

metal_device_information

List Available Metal Devices and Their Properties

Description

Enumerate all Metal devices on the current system. Returns a list with one entry per device describing the device's properties.

Usage

```
metal_device_information()
```

Details

All devices on the current system conforming to the modern Metal specification will be enumerated.

`metal_is_available` is available as a sentinel function that returns TRUE if this package's configure script was able to successfully compile a Metal test program.

Value

When available devices are present, an object of class `c("device_list", "list")`, where each element is itself is of class `c("alternative_device", "list")` and describes a single OpenCL compliant device with the following components:

`index` Integer: The flat device index, starting at 0, assigned in platform-major order. This index is used to select a device elsewhere in the package.

`name` Character: The device name, see `CL_DEVICE_NAME`.

`is_default` Logical: Indicating whether this device is the OS's default device.

`has_unified_memory` Logical: Indicating whether this device used unified memory, i.e. an M4 or later Silicon device.

`is_low_power` Logical: TBA.

`is_removable` Logical: TBA.

`type` Character: an interpretation of `CL_DEVICE_TYPE`, currently one of the following.

- CPU
- GPU
- ACCELERATOR
- CUSTOM
- UNKNOWN – returned when the queried bitfield doesn't match a recognized device type constant.

`max_compute_units` Integer: For now this is a sentinel NA, as this isn't a query-able quality in the Metal specification?

`recommended_max_working_set_size_bytes` Numeric: Global memory size in bytes.

`max_buffer_length_bytes` Numeric: Maximum work group size.

`max_threads_per_threadgroup` Numeric: A vector indicating the max threads per threadgroup by dimension.

`max_threadgroup_memory_length_bytes` Numeric: Indicates the largest single allocation the device permits.

`registry_id` Numeric: device registry id.

`location` Numeric: device location.

`device_ptr` externalptr: pointer classed as `metal_device` wrapping the underlying Metal device handle.

`framework` Character: In the case of this function, just populated with `metal`.

If no OpenCL compliant devices are found, this function should return NULL.

Note

This R function is a relatively thin wrapper for the C-level `metal_available_devices`. If no Metal devices are detected on the system, i.e. the system is not Darwin, returns NULL

See Also

[metal_is_available](#)

Examples

```
devices <- metal_device_information()
str(devices)
```

metal_is_available	<i>Check Whether Metal Support Was Compiled Into This Build</i>
--------------------	---

Description

Report whether ardea was compiled with Metal support on this system.

Usage

```
metal_is_available()
```

Details

This function is an R wrapper for ardea's C function `metal_sentinel`. Unlike [opencl_is_available](#), which performs a live enumeration of OpenCL devices on the current system, `metal_is_available` reflects a compile-time decision made once, during package installation.

A TRUE result means the functions in ardea that depend on Metal were compiled and are callable; it does not by itself guarantee that a Metal-compliant GPU is currently attached, only that one was detected at install time. A FALSE result means underlying functions supporting interaction with Metal were not compiled into the local build of the package.

Value

Logical: TRUE if Metal support was detected and compiled in at installation time, FALSE otherwise. FALSE is the expected result on any non-macOS platform, and is also expected on a macOS system where Metal detection failed during installation.

See Also

[opencl_is_available](#), [cuda_is_available](#)

Examples

```
metal_is_available()
```

metal_make_context	Create a Metal Context for a Device
--------------------	-------------------------------------

Description

Construct a Metal context – a command queue paired with a specific device – given a device entry returned by [metal_device_information](#).

Usage

```
metal_make_context(device)
```

Arguments

device	alternative_device: a single device entry from the list created by metal_device_information . Specifically required device properties currently include: • framework, must be metal in this case • device_ptr, a ptr for the device
--------	---

Details

A command queue is created for the supplied device under the hood, and bundled together with the device into a single object so callers don't need to manage two separate pointers. The device is retained independently of the device_ptr external pointer passed in, so that pointer may be garbage collected without affecting the returned context.

Metal contexts are structurally scoped to a single device – unlike OpenCL, where a context can span multiple devices.

The returned context is reference-counted by the underlying Metal runtime and is released automatically when the R external pointer is garbage collected.

Value

An external pointer of class "metal_context" wrapping the created device/queue pair. This is consumed by other ardea functions that require a context, such as [metal_make_program](#) and [metal_make_kernelptr](#).

See Also

[metal_device_information](#), [metal_make_program](#), [metal_make_kernelptr](#)

Examples

```
if (metal_is_available() &  
    metal_devices_exist()) {  
  devices <- metal_device_information()  
  ctx <- metal_make_context(devices[[1]])  
}
```

metal_make_kernelptr *Create Dispatch-Ready Kernel Pointers from a Metal Library*

Description

Given a compiled Metal library and the context it should be paired with, build a compute pipeline for each named kernel function and return them as a named list of external pointers.

Usage

```
metal_make_kernelptr(program,  
                     context,  
                     kernel_names)
```

Arguments

program	metal_library: a library created by metal_make_program .
context	metal_context: the context created by metal_make_context , used to determine which device the pipeline(s) should be compiled against. Should be the same context (or at least the same underlying device) that program was compiled with.
kernel_names	character: a vector of length 1 or greater, naming kernel void functions declared in the .metal source program was compiled from.

Details

For each name in kernel_names, the corresponding function is extracted from program and used to build a compute pipeline state. The intermediate function object is discarded immediately after the pipeline is built – it is not retained or returned.

Unlike [opencl_make_kernelptr](#), which returns a handle (cl_kernel) that still needs binding to a device-specific execution context at dispatch time, each pointer returned here already bundles the compiled pipeline together with the identity of the device it was built against. It is dispatch-ready as returned: no separate pipeline-creation step remains to be performed per-call.

Building a compute pipeline is a relatively expensive operation. This function is intended to be called once per kernel and the result reused across many dispatches, rather than rebuilding a pipeline on every call.

The returned list is named using kernel_names, in the order supplied.

Value

A named list of external pointers of class "metal_pipeline", one per element of kernel_names.

See Also

[metal_make_context](#), [metal_make_program](#)

Examples

```
metal_mm_naive <- '
kernel void metal_mm_naive(device float* output [[buffer(0)]],
                           constant uint& M [[buffer(1)]],
                           constant uint& K [[buffer(2)]],
                           constant uint& N [[buffer(3)]],
                           device const float* A [[buffer(4)]],
                           device const float* B [[buffer(5)]],
                           uint2 id [[thread_position_in_grid]])
{
    uint row = id.x;
    uint col = id.y;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (uint k = 0; k < K; k++) {
            sum += A[row + k * M] * B[k + col * K];
        }
        output[row + col * M] = sum;
    }
}
'

tmp_metal <- tempfile(fileext = ".metal")
tmp_metallib <- tempfile(fileext = ".metallib")
writeLines(text = metal_mm_naive,
           con = tmp_metal)

if (metal_is_available() &
    metal_devices_exist()) {
    devices <- metal_device_information()
    ctx <- metal_make_context(devices[[1]])
    lib <- metal_make_program(tmp_metal, ctx)
    kernels <- metal_make_kernelptr(lib, ctx, c("metal_mm_naive"))
}
```

metal_make_program	<i>Compile a Metal Source File into a Library</i>
--------------------	---

Description

Compile a .metal source file against a [metal_make_context](#) context, and return the resulting library as an external pointer, ready to be passed to [metal_make_kernelptr](#).

Usage

```
metal_make_program(metal_file,
                  context,
                  metallib_file = NULL,
                  xcrun_flags = NULL,
```

```
fast_math = FALSE,
language_version = NULL)
```

Arguments

metal_file	character: a length-1 path to a .metal source file. Must exist.
context	metal_context: a context created by metal_make_context .
metallib_file	character or NULL: if supplied, a length-1 output path for a compiled .metallib. Selects the xcrun-based compilation path described in Details, rather than the default runtime API path.
xcrun_flags	character or NULL: additional flags passed through to xcrun metal on the metallib_file compilation path. Ignored (with a warning if non-NULL) when metallib_file is NULL. Each element is treated as a single shell word, matching system2()'s own args convention – a flag that takes a value, e.g. -D, must be supplied as two separate elements (c("-D", "TILE_SIZE=32")), not one space-joined string. May not include "-o", since this function already supplies that flag from metallib_file.
fast_math	logical: a single non-NA value. Enables Metal's fast-math compiler option on the runtime API compilation path. Ignored (with a warning if non-default) when metallib_file is supplied.
language_version	character or NULL: a length-1 Metal Shading Language version string (e.g. "2.4", "3.0"), or NULL to use the platform default. Applies to the runtime API compilation path only; ignored (with a warning if non-NULL) when metallib_file is supplied. Availability of specific version strings depends on the SDK this package was built against.

Details

This function supports two compilation paths, selected by whether metallib_file is supplied:

Runtime API path (metallib_file = NULL, the default): source is compiled in-process via Metal's `newLibraryWithSource:options:error:`, using `fast_math` and `language_version` to configure the compile options. This requires the Metal compiler toolchain (normally installed alongside Xcode or the Xcode Command Line Tools) to be present on the machine running R – a stricter requirement than simply having a Mac with a supported GPU. Because compilation is from an in-memory source string with no filesystem search-path context, `#include` of separate header files is not reliably supported on this path.

xcrun path (metallib_file supplied): `metal_file` is compiled to a .metallib at `metallib_file` by shelling out to `xcrun -sdk macosx metal`, and the compiled library is then loaded from disk. This path does not share the runtime path's `#include` limitation, since xcrun compiles a real file with real filesystem context. Use `xcrun_flags` to reach xcrun metal's own compiler flags (include paths, preprocessor defines, and so on) on this path; `fast_math`/`language_version` have no effect here.

Both paths return an object of the same class – which compiler produced the library is not visible to [metal_make_kernelptr](#) or anything downstream of it.

opengl_devices_exist *Check For OpenCL Compliant Devices*

Description

Report whether at least one OpenCL compliant device is available on the current system, across all detected platforms.

Usage

```
opengl_devices_exist()
```

Details

This function is an R wrapper for ardea's C function `exposed_device_count`. If `exposed_device_count` returns a value greater than zero, this function returns `TRUE`. Device enumeration should hypothetically include openCL compliant CPUs enabled by POCL-like runtimes, but this has not been tested. This means systems with OpenCL compliant CPUs, but no GPUs will return `TRUE`.

Platforms that fail to return valid information are skipped, meaning broken or misconfigured ICDs will remove a device from availability.

Value

A single logical value: `TRUE` if one or more OpenCL devices were found on any platform, `FALSE` if no devices were discovered. If this package installed successfully, `FALSE` should indicate that there are no OpenCL compliant **devices** present. As successful package installation requires working OpenCL headers and shared libraries.

See Also

[opengl_is_available](#) [opengl_device_information](#)

Examples

```
if (opengl_is_available()) {
  opengl_devices_exist()
}
```

opencl_device_information

List Available OpenCL Devices and Their Properties

Description

Enumerate all OpenCL devices found across all platforms on the current system. Returns a list with one entry per device describing the device's properties.

Usage

```
opencl_device_information()
```

Details

All devices on the current system (see `clGetPlatformIDs` and `clGetDeviceIDs` in the OpenCL specification) will be enumerated, including POCL enabled CPUs. A platform that fails to enumerate its devices is skipped rather than treated as a fatal error.

[opencl_is_available](#) is available as a lightweight check of whether any device is present at all, before attempting retrieval of full details with this function.

Value

When available devices are present, an object of class `c("device_list", "list")`, where each element is itself is of class `c("alternative_device", "list")` and describes a single OpenCL compliant device with the following components:

`index` Integer: The flat device index, starting at 0, assigned in platform-major order. This index is used to select a device elsewhere in the package.

`name` Character: The device name, see `CL_DEVICE_NAME`.

`vendor` Character: The device vendor, see `CL_DEVICE_VENDOR`.

`version` Character: The OpenCL version supported by the device, see `CL_DEVICE_VERSION`.

`driver_version` Character: The vendor driver version, see `CL_DRIVER_VERSION`.

`type` Character: an interpretation of `CL_DEVICE_TYPE`, currently one of the following.

- CPU
- GPU
- ACCELERATOR
- CUSTOM
- UNKNOWN – returned when the queried bitfield doesn't match a recognized device type constant.

`max_compute_units` Integer: The number of parallel compute units on the device, see `CL_DEVICE_MAX_COMPUTE_UNITS`.

`global_mem_size` Numeric: Global memory size in bytes, see `CL_DEVICE_GLOBAL_MEM_SIZE`.

`max_work_group_size` Numeric: Maximum work group size, see `CL_DEVICE_MAX_WORK_GROUP_SIZE`.

max_work_group_dimensions Integer: The number of work dimensions the device is capable of, see CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS.

max_work_item_sizes Numeric: A vector the length of the max_work_group_dimensions, see CL_DEVICE_MAX_WORK_ITEM_SIZES.

max_mem_alloc_size Numeric: Indicates the largest single allocation the device permits, see CL_DEVICE_MAX_MEM_ALLOC_SIZE.

local_mem_size Numeric: See CL_DEVICE_LOCAL_MEM_SIZE.

local_mem_type Character: Describes the memory type of the device, see CL_DEVICE_LOCAL_MEM_TYPE.

available Logical: A signal whether the device is available for function dispatch, i.e. not powered down, see CL_DEVICE_AVAILABLE.

extensions Character: A space separated vector enumerating device extensions, see CL_DEVICE_EXTENSIONS.

device_id externalptr: pointer classed as opengl_device_id wrapping the underlying OpenGL device handle.

platform_id externalptr: pointer classed as opengl_platform_id wrapping the underlying OpenGL platform handle.

framework Character: In the case of this function, just populated with opengl.

If no OpenGL compliant devices are found, this function should return NULL.

Note

This R function is a relatively thin wrapper for the C-level opengl_available_devices, which will return an empty list if no OpenGL compliant devices are present. That empty list is signaled to NULL in R.

See Also

[opengl_is_available](#)

Examples

```
if (opengl_is_available()) {
  devices <- opengl_device_information()
  str(devices)
}
```

opengl_is_available *Check For OpenCL Compilation Success*

Description

Report whether OpenCL compilation probes were successful.

Usage

```
opengl_is_available()
```

Details

This function is an R wrapper for ardea's C function `opengl_sentinel`.

Value

A single logical value: TRUE if compilation of OpenCL capability probes were successful. If this package installed successfully FALSE indicates the compilation probe for OpenCL was not successful.

See Also

[opengl_device_information](#) [opengl_devices_exist](#)

Examples

```
opengl_is_available()
```

<code>opengl_make_context</code>	<i>Create an OpenCL Context for a Device</i>
----------------------------------	--

Description

Construct an OpenCL context scoped to a single device, given a device entry returned by [opengl_device_information](#).

Usage

```
opengl_make_context(device)
```

Arguments

<code>device</code>	<code>alternative_device</code>: a single device entry from the list created by opengl_device_information. Specifically required device properties currently include: • <code>framework</code>, must be <code>opengl</code> in this case • <code>device_id</code>, a ptr for the device • <code>platform_id</code>, a ptr for the platform
---------------------	---

Details

Context is created under the hood via `clCreateContext`, explicitly scoped to the supplied device entry's device and platform. This avoids ambiguity on systems with more than one OpenCL platform (ICD) installed, where omitting the platform property can produce implementation-defined behavior.

Only a single device is included in the context created by this function. OpenCL itself permits a context to span multiple devices unlike CUDA or Metal, both of which are structurally limited to one device per context. That capability is not currently exposed in this package.

The returned context is reference-counted by the OpenCL runtime and should be released automatically (`clReleaseContext`) when the R external pointer is garbage collected.

Value

An external pointer of class "opengl_context" wrapping the created OpenCL context. This is consumed by other ardea functions that require a context, such as [opengl_make_program](#).

See Also

[opengl_device_information](#), [opengl_make_program](#)

Examples

```
if (opengl_is_available() &
    opengl_devices_exist()) {
  devices <- opengl_device_information()
  ctx <- opengl_make_context(devices[[1]])
}
```

opengl_make_kernelptr *Extract Kernel Pointers from an OpenCL Program*

Description

Given a built OpenCL program and one or more kernel function names, extract callable kernel handles for each name.

Usage

```
opengl_make_kernelptr(program,
                      kernel_names)
```

Arguments

program	An external pointer, created by opengl_make_program . Must represent a successfully built program.
kernel_names	A character vector of length 1 or greater, giving the name of each __kernel-qualified function to extract from program.

Details

A single OpenCL program can define multiple kernel functions; this function may be called once to extract several of them at once, rather than requiring one call per kernel.

clCreateKernel is called once per name in kernel_names. Kernel function names that cannot be resolved will cause the function to error and return a message identifying the offending name.

Each returned kernel handle is reference-counted by the OpenCL runtime and should be released automatically (clReleaseKernel) when its R external pointer is garbage collected.

Value

A named list, with one element per entry in `kernel_names`, in the same order. Each element is an external pointer wrapping the corresponding compiled kernel. List position names are inferred from the supplied `kernel_names`.

See Also

[opengl_make_program](#), [opengl_make_context](#)

Examples

```
if (opengl_is_available() &
    opengl_devices_exist()) {
  kernel_source <- "
    __kernel void add_one(__global float *data) {
      int gid = get_global_id(0);
      data[gid] = data[gid] + 1.0f;
    }
  "
  tmp01 <- tempfile(fileext = ".cl")
  writeLines(kernel_source, tmp01)

  devices <- opengl_device_information()
  ctx <- opengl_make_context(devices[[1]])

  program <- opengl_make_program(tmp01, ctx)
  kernels <- opengl_make_kernelptr(program, "add_one")
  unlink(tmp01)
}
```

`opengl_make_program` *Compile an OpenCL Program from Source*

Description

Reads a `.cl` source file, compiles it against a given OpenCL context, and returns the built program.

Usage

```
opengl_make_program(cl_file,
                    context,
                    build_options = NULL)
```

Arguments

<code>cl_file</code>	Character: vector of length 1 giving the path to a <code>.cl</code> OpenCL C source file. Must exist.
<code>context</code>	An external pointer, as returned by opengl_make_context . The program is compiled for the device(s) associated with this context.

build_options Character: An optional vector of length 1 containing OpenCL C compiler options to be passed to `clBuildProgram`. Defaults to NULL signifying default options, see Details.

Details

Source is read from `cl_file` and compiled via `clCreateProgramWithSource` followed by `clBuildProgram`. Unlike CUDA and Metal, OpenCL's compiler is part of the runtime API so we are not invoking `system2` to communicate with an external executable.

If compilation fails, the OpenCL build log should be printed via a warning message before an error is raised, so the underlying OpenCL C compiler error (missing semicolon, unknown type, etc.) is visible to the user rather than only a generic OpenCL error code.

`build_options` corresponds directly to the `options` argument of `clBuildProgram`; see "**What build_options is for**" below for its purpose and common values.

Value

An external pointer wrapping the built OpenCL program. This is consumed by `opencl_make_kernelptr` to extract callable kernel handles.

What build_options is for

`build_options` is a single string of space-separated flags passed to the OpenCL C compiler at build time. This is analogous to flags that would be passed to `gcc`, `clang`, `nvcc`, or `xcrun`. It lets a user influence how the `.cl` source is compiled without editing the source file itself. Common categories:

- **Preprocessor defines**, e.g. `"-DBLOCK_SIZE=256"`. Sets a preprocessor macro used in the kernel source, letting the same `.cl` file be recompiled with different constants (tile sizes, loop bounds, feature toggles) without maintaining multiple source files.
- **Include paths**, e.g. `"-I/path/to/headers"`. Needed if the OpenCL file includes non-standard headers.
- **Language version selection**, e.g. `"-cl-std=CL2.0"`. Set a specific OpenCL C language version, if the source relies on features not present in the device's default version.
- **Optimization and numerical-precision trade-offs**, e.g. `"-cl-fast-relaxed-math"` or `"-cl-mad-enable"`. Relax IEEE-754 conformance in exchange for faster generated code. Appropriate when a kernel's numerical tolerance allows it, but not a safe default for every workload.
- **Warnings-as-errors**, e.g. `"-Werror"`. Catch compiler warnings that would otherwise be silently ignored.

Multiple flags can be combined in the one string, since `clBuildProgram` expects a single options string, e.g.:

```
opencl_make_program("kernels.cl", ctx,
    build_options = "-DBLOCK_SIZE=256 -cl-fast-relaxed-math")
```

Most users compiling straightforward kernels can leave `build_options` as NULL. It exists for cases where compile-time configuration or a specific numerical/language mode is needed, not as a required argument.

See Also

[openc1_make_context](#), [openc1_make_kernelptr](#)

Examples

```
if (openc1_is_available() &
    openc1_devices_exist()) {
  kernel_source <- "
  __kernel void add_one(__global float *data) {
    int gid = get_global_id(0);
    data[gid] = data[gid] + 1.0f;
  }
  "
  tmp01 <- tempfile(fileext = ".cl")
  writeLines(kernel_source, tmp01)

  devices <- openc1_device_information()
  ctx <- openc1_make_context(devices[[1]])

  program <- openc1_make_program(tmp01, ctx)
  unlink(tmp01)
}
```

simple_wrapper

Perform Simple Kernel Function Dispatch for a Selected Framework

Description

Dispatch a single compiled kernel function call to the selected GPU framework's underlying runner, using a common set of arguments. Currently supports "openc1", "metal", and "cuda".

Usage

```
simple_wrapper(framework = c("openc1",
                             "metal",
                             "cuda"),
              context_ptr,
              kernel_ptr,
              arg_types,
              arg_list,
              problem_dims,
              group_dims,
              workers_per = NULL)
```

Arguments

framework	Character: One of "openc1", "metal", or "cuda", partial matching via match.arg is supported.
-----------	--

context_ptr	Externalptr: Classed to match the option passed to framework.
kernel_ptr	Externalptr: Classed to match the option passed to framework.
arg_types	Character: vector the same length as arg_list, naming the device-side type of each corresponding argument (e.g. "float", "double", "int", "long"). The first element describes the output.
arg_list	List: A list the same length as arg_types. The first element is the output template: its length and declared type (per arg_types[1]) define the result buffer. Remaining elements are the input arguments – length-1 elements are bound as scalar kernel arguments, longer elements are staged into device buffers.
problem_dims	Numeric: A vector of length 3 giving the logical (global) problem size in up to three dimensions – e.g. c(nrow, ncol, 1) for a 2D problem. Unused trailing dimensions should be set to 1.
group_dims	Numeric: A vector of length 3 giving the local work-group size per dimension, or NULL to let the underlying runner derive a default from the device's reported limits and workers_per.
workers_per	Numeric: An optional single value giving the target number of work-items per work-group, used only when group_dims is NULL to derive a default work-group size. Defaults to NULL (the underlying the device's default is queries when applicable).

Details

This function serves as a consistent dispatch interface across frameworks for simple use cases.

Argument	OpenCL constructor	CUDA constructor	Metal constructor
context_ptr	opengl_make_context	cuda_make_context	metal_make_context
kernel_ptr	opengl_make_kernelptr	cuda_make_kernelptr	metal_make_kernelptr

Argument	OpenCL concept	Metal concept	CUDA concept
problem_dims	global size	threads per grid	grid size (total threads)
group_dims	work-group size	threadgroup size	block size
workers_per	threads per threadgroup	threads per threadgroup	threads per block

Validation of device availability and argument compatibility should all occur before dispatch.

Value

A numeric or integer vector (matching the type declared in arg_types[1]) of length length(arg_list[[1]]), containing the dispatched kernel's output.

See Also

[opengl_make_context](#), [opengl_make_program](#), [opengl_make_kernelptr](#), [cuda_make_context](#), [cuda_make_program](#), [cuda_make_kernelptr](#), [metal_make_context](#), [metal_make_program](#), [metal_make_kernelptr](#)

Examples

```
# write out a character vector to a tempfile for opencl, metal, and cuda
opencl_mm_naive <- '
__kernel void opencl_mm_naive(__global float* output,
                             const long M,
                             const long K,
                             const long N,
                             __global const float* A,
                             __global const float* B)
{
    long row = get_global_id(0);
    long col = get_global_id(1);

    if (row < M && col < N) {
        float sum = 0.0f;
        for (long k = 0; k < K; k++) {
            sum += A[row + k * M] * B[k + col * K];
        }
        output[row + col * M] = sum;
    }
}
',
tmp_cl <- tempfile(fileext = ".cl")
writeLines(text = opencl_mm_naive,
           con = tmp_cl)

cuda_mm_naive <- '
extern "C" __global__ void cuda_mm_naive(float* output,
                                         const long long M,
                                         const long long K,
                                         const long long N,
                                         const float* A,
                                         const float* B)
{
    long long row = blockIdx.x * blockDim.x + threadIdx.x;
    long long col = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (long long k = 0; k < K; k++) {
            sum += A[row + k * M] * B[k + col * K];
        }
        output[row + col * M] = sum;
    }
}
',
tmp_cu <- tempfile(fileext = ".cu")
tmp_ptx <- tempfile(fileext = ".ptx")
writeLines(text = cuda_mm_naive,
           con = tmp_cu)

metal_mm_naive <- '

```

```

kernel void metal_mm_naive(device float* output [[buffer(0)]],
                           constant uint& M [[buffer(1)]],
                           constant uint& K [[buffer(2)]],
                           constant uint& N [[buffer(3)]],
                           device const float* A [[buffer(4)]],
                           device const float* B [[buffer(5)]],
                           uint2 id [[thread_position_in_grid]])
{
    uint row = id.x;
    uint col = id.y;

    if (row < M && col < N) {
        float sum = 0.0f;
        for (uint k = 0; k < K; k++) {
            sum += A[row + k * M] * B[k + col * K];
        }
        output[row + col * M] = sum;
    }
}
',
tmp_metal <- tempfile(fileext = ".metal")
tmp_metallib <- tempfile(fileext = ".metallib")
writeLines(text = metal_mm_naive,
           con = tmp_metal)

# generic inputs, regardless of framework
var00 <- vector(mode = "numeric",
                length = 250000)
var01 <- matrix(rnorm(250000),
                nrow = 500,
                ncol = 500)
var02 <- matrix(rnorm(250000),
                nrow = 500,
                ncol = 500)
dim01 <- nrow(var01)
dim02 <- ncol(var01)
# dim03 <- nrow(var02)
dim04 <- ncol(var02)

arg_list <- list(var00, # our expected output vector, we've just filled it with zeros
                 dim01,
                 dim02,
                 dim04,
                 as.vector(var01),
                 as.vector(var02))

opengl_arg_types <- c("float",
                      "long",
                      "long",
                      "long",
                      "float",
                      "float")
metal_arg_types <- c("float",

```

```

        "uint",
        "uint",
        "uint",
        "float",
        "float")
cuda_arg_types <- c("float",
                    "long",
                    "long",
                    "long",
                    "float",
                    "float")

print("Builtin implementation:")
system.time(res01 <- var01 %*% var02)

if (opencl_is_available() &
    opencl_devices_exist()) {
  print("OpenCL is available on this system!")
  cl_dvcs <- opencl_device_information()
  cl_ctx <- opencl_make_context(device = cl_dvcs[[1]])
  cl_program <- opencl_make_program(cl_file = tmp_cl,
                                   context = cl_ctx)
  cl_knl <- opencl_make_kernelptr(program = cl_program,
                                 kernel_names = "opencl_mm_naive")
  print("OpenCL implementation:")
  print(system.time(res02 <- simple_wrapper(
    framework = "opencl",
    context_ptr = cl_ctx,
    kernel_ptr = cl_knl$opencl_mm_naive,
    arg_types = opencl_arg_types,
    arg_list = arg_list,
    problem_dims = as.integer(c(dim01,
                                dim04,
                                1L)),
    group_dims = NULL,
    workers_per = NULL)))

  plot(as.vector(res01),
       res02,
       pch = 46,
       xlab = "builtin",
       ylab = "OpenCL")
}
if (cuda_is_available() &
    cuda_devices_exist()) {
  print("CUDA is available on this system!")
  cu_dvcs <- cuda_device_information()
  cu_ctx <- cuda_make_context(device = cu_dvcs[[1]])
  cu_program <- cuda_make_program(cuda_file = tmp_cu,
                                 context = cu_ctx,
                                 ptx_file = tmp_ptx)
  cu_knl <- cuda_make_kernelptr(program = cu_program,
                                context = cu_ctx,
                                kernel_names = "cuda_mm_naive")
  print("CUDA implementation:")

```

```

print(system.time(res03 <- simple_wrapper(framework = "cuda",
                                          context_ptr = cu_ctx,
                                          kernel_ptr = cu_knl$cuda_mm_naive,
                                          arg_types = cuda_arg_types,
                                          arg_list = arg_list,
                                          problem_dims = as.integer(c(dim01,
                                                                    dim04,
                                                                    1L))),
                                          group_dims = NULL,
                                          workers_per = NULL)))

plot(as.vector(res01),
     res03,
     pch = 46,
     xlab = "builtin",
     ylab = "CUDA")
}
if (metal_is_available() &
    metal_devices_exist()) {
  print("Metal is available on this system!")
  mtl_dvcs <- metal_device_information()
  mtl_ctx <- metal_make_context(device = mtl_dvcs[[1]])
  if (metal_compiler_is_available()) {
    mtl_program <- metal_make_program(metal_file = tmp_metal,
                                      metallib_file = tmp_metallib,
                                      context = mtl_ctx)
  } else {
    mtl_program <- metal_make_program(metal_file = tmp_metal,
                                      context = mtl_ctx)
  }

  mtl_knl <- metal_make_kernelptr(program = mtl_program,
                                  context = mtl_ctx,
                                  kernel_names = "metal_mm_naive")
  print("Metal implementation:")
  print(system.time(res03 <- simple_wrapper(framework = "metal",
                                          context_ptr = mtl_ctx,
                                          kernel_ptr = mtl_knl$metal_mm_naive,
                                          arg_types = metal_arg_types,
                                          arg_list = arg_list,
                                          problem_dims = as.integer(c(dim01,
                                                                    dim04,
                                                                    1L))),
                                          group_dims = NULL,
                                          workers_per = NULL)))

  plot(as.vector(res01),
       res03,
       pch = 46,
       xlab = "builtin",
       ylab = "metal")
}

```

supported_types_by_context

Check Supported Kernel Argument Types for a Context

Description

Given a compute context and its associated framework, empirically probe which scalar C types can be used as kernel arguments on the device(s) backing that context.

Usage

```
supported_types_by_context(framework = c("opencl",
                                         "metal",
                                         "cuda"),
                          context_ptr)
```

Arguments

framework	Character: vector of length 1, one of "opencl", "metal", or "cuda", indicating which framework context_ptr belongs to. Partial matching via match.arg is supported; "metal" and "cuda" are not yet implemented, see Details.
context_ptr	externalptr: An external pointer to a compute context, as returned by opencl_make_context for the "opencl" framework. Its class must match framework, e.g. an "opencl_context" when framework = "opencl".

Details

This function a framework-specific list of type keywords, attempting to build a small single-argument kernel using that type via the C-level `opencl_probe_type_support` function. This is an empirical test rather than a spec or extension lookup: a type reported as unsupported may still be listed in a device's extension string, and vice versa. FALSE reported types should be treated as a compile-time fact rather than a statement about the underlying hardware.

Before probing, context_ptr's class is checked against framework ("opencl_context", "metal_context", or "cuda_context" respectively). A mismatch, e.g. passing an OpenCL context while framework = "metal", raises an error rather than silently probing the wrong API.

framework = "metal" and framework = "cuda" are accepted by [match.arg](#) but currently raise "not yet supported" errors once past the class check; support for these frameworks is planned but not yet implemented.

Value

A named logical vector, one entry per type keyword checked for the given framework. For framework = "opencl" the types are c("float", "double", "char", "short", "int", "long", "uchar", "ushort", "uint", "ulong"). Recorded as TRUE if a kernel using that type as a scalar argument compiles and builds successfully on the context's device(s), FALSE otherwise.

See Also[opengl_make_context](#), [opengl_make_program](#)**Examples**

```
if (opengl_is_available() &
    opengl_devices_exist()) {
  devices <- opengl_device_information()
  ctx <- opengl_make_context(devices[[1]])

  supported_types_by_context(backend = "opengl",
                             context_ptr = ctx)
}
```

Index

cuda_device_information, [2](#), [3](#), [5–7](#)
cuda_devices_exist, [2](#)
cuda_is_available, [2–4](#), [4](#), [13](#)
cuda_make_context, [3](#), [4](#), [5](#), [6–9](#), [27](#)
cuda_make_kernelptr, [4](#), [6](#), [6](#), [8](#), [9](#), [27](#)
cuda_make_program, [6](#), [7](#), [8](#), [27](#)

match.arg, [26](#), [32](#)
metal_compiler_is_available, [9](#)
metal_device_information, [3](#), [4](#), [11](#), [11](#), [14](#)
metal_devices_exist, [10](#), [11](#)
metal_is_available, [4](#), [5](#), [10–13](#), [13](#)
metal_make_context, [5](#), [14](#), [15–18](#), [27](#)
metal_make_kernelptr, [7](#), [14](#), [15](#), [16–18](#), [27](#)
metal_make_program, [8–10](#), [14](#), [15](#), [16](#), [27](#)

opencl_device_information, [4](#), [19](#), [20](#), [22](#),
[23](#)
opencl_devices_exist, [19](#), [22](#)
opencl_is_available, [4](#), [5](#), [13](#), [19–21](#), [21](#)
opencl_make_context, [5](#), [22](#), [24](#), [26](#), [27](#), [32](#),
[33](#)
opencl_make_kernelptr, [15](#), [23](#), [25–27](#)
opencl_make_program, [23](#), [24](#), [24](#), [27](#), [33](#)

simple_wrapper, [26](#)
supported_types_by_context, [32](#)