Package ‘OpenCL’

July 21, 2025

Version 0.2-10

Title Interface allowing R to use OpenCL

Author Simon Urbanek <Simon.Urbanek@r-project.org>, Aaron Puchert <aaronpuchert@alice-dsl.net>

Maintainer Simon Urbanek <Simon.Urbanek@r-project.org>

Depends R (>= 2.0.0)

Description This package provides an interface to OpenCL, allowing R to leverage computing power of GPUs and other HPC accelerator devices.

License BSD_3_clause + file LICENSE

SystemRequirements OpenCL library, GNU make

URL http://www.rforge.net/OpenCL/

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-12-22 10:33:27 UTC

Contents

`clBuffer` *Create and handle OpenCL buffers*

Description

OpenCL buffers are just like numeric or integer vectors that reside on the GPU and can directly be accessed by kernels. Both non-scalar arguments to `oclRun` and its return type are OpenCL buffers.

Just like vectors in R, OpenCL buffers have a mode, which is (as of now) one of "double" or "numeric" (corresponds to `double` in OpenCL C), "single" (`float`) or "integer" (`int`).

The constructor `clBuffer` takes a context as created by `oclContext`, a length and a mode argument.

The conversion function `as.clBuffer` creates an OpenCL buffer of the same length and mode as the argument and copies the data. Conversely, `as.double` (= `as.numeric`) and `as.integer` read a buffer and coerce the result as vector the appropriate mode.

With `is.clBuffer` one can check if an object is an OpenCL buffer.

The methods `length.clBuffer` and `print.clBuffer` retrieve the length and print the contents, respectively.

Basic access to the data is available via `[...]`. Note that only contiguous memory operations are supported on GPU buffers, so if the index does not reference a contiguous region then the subsetting/assignment will be performed by retrieving the entire buffer and perfroming the operation in R which is very expensive.

Note that unlike regular R objects GPU buffers are by design mutable, i.e. the object is only a reference to the GPU memory and thus any modification will affect all refernces. The contents can be emerged into R via `x[]` at which point the result is a regular R vector and no longer tied to the source buffer. Analogously, `x[] <- value` is the canonical way to replace the entire contents of the buffer where `value` must have the same length as the buffer (no recycling).

Usage

```
clBuffer(context, length, mode = c("numeric", "single", "double", "integer"))
as.clBuffer(vector, context, mode = class(vector))
is.clBuffer(any)
## S3 method for class 'clBuffer'
as.double(x, ...)
## S3 method for class 'clBuffer'
as.integer(x, ...)
## S3 method for class 'clBuffer'
print(x, ...)
## S3 method for class 'clBuffer'
length(x)
## S3 method for class 'clBuffer'
x[i]
## S3 replacement method for class 'clBuffer'
x[i] <- value
```

Arguments

`context`	OpenCL context as created by `oclContext`
`length`	Length of the required buffer
`mode`	Mode of the buffer, can be one of "numeric", "single", "double" or "integer"
`vector`	Numeric or integer vector or `clFloat` object
`any`	Arbitrary object
`x`	OpenCL buffer object (`clBuffer`)
`i`	index specifying elements to extract or replace
`value`	New values
`...`	Arguments passed to subsequent methods

Author(s)

Aaron Puchert and Simon Urbanek

See Also

`oclContext`, `oclRun`

Examples

```
library(OpenCL)

## Only proceed if this machine has at least one OpenCL platform
if (length(oclPlatforms())) {

ctx<-oclContext()

buf<-clBuffer(ctx, 16, "numeric")
# Do not write buf<-..., as this replaces buf with a vector.
buf[]<-sqrt(1:16)
buf

intbuf<-as.clBuffer(1:16, ctx)
print(intbuf)

length(buf)
as.numeric(buf)
buf[]

buf[3:5]
buf[1:2] = 0
buf

## clBuffer is the required argument and return type of oclRun.
## See oclRun() examples.
}
```

clLocal — *Create and handle local memory arguments for OpenCL kernels*

Description

OpenCL kernels allow the use of local memory which is shared by all work-items of a work-group. In most cases, such memory is allocated inside the kernel at compile time such as `__local numeric temp[GROUP_SIZE]`. However, in some rare circumstances it may be desirable to allocate the buffer dynamically as an argument to the kernel. In that case the corresponding argument of the kernel is defined with the `__local` keyword and the caller has to specify the size of the local memory buffer at run-time when calling the kernel.

The `clLocal()` function creates a specification of the local memory buffer. It is the only object that may be passed to a kernel argument declared with `__local`. The object is merely a specification that `oclRun` knows how to interpret, `clLocal` doesn't actually allocate any memory.

By default, `size` is interpreted as bytes, but for convenience it can also specify the number of elements of a particular type. In the special case of `"numeric"` the actual size of one element (and thus the total buffer size) will depend on the context in which this specification is used (single or double precision).

With `is.clLocal` one can check if an object is a local buffer specification.

The methods `length.clLocal` and `print.clLocal` retrieve the length (number of elements) and print the contents, respectively.

Usage

```
clLocal(length, mode = c("byte", "numeric", "single", "double", "integer"))
is.clLocal(x)
## S3 method for class 'clLocal'
print(x, ...)
## S3 method for class 'clLocal'
length(x)
```

Arguments

`length`	numeric, length (number of elements) of the required buffer. The actual size will depend on `mode`.
`mode`	string, mode of the buffer (only used to compute the total size in bytes). The default is to treat `length` as the size in bytes (i.e., `"byte"` is aways allowed irrespective of the type of the kernel argument).
`x`	object
`...`	Ignored

Value

`clLocal` returns an object of the class `"cLocal"`

`is.clLocal` return `TRUE` for `"clLocal"` objects and `FALSE` otherwise.

`print` method returns `x` invisibly.

`length` returns a numeric scalar with the length (number of elements) in the buffer specification.

Note

The internal structure of the `clLocal` object should be considered private, may change and no user code should access its components. Similarly, `clLocal` objects are only legal when returned from the `clLocal()` function, they may not be created by other means or mutated.

Author(s)

Simon Urbanek

See Also

`oclRun`

`oclContext` *Create an OpenCL context for a given device.*

Description

OpenCL contexts host kernels and buffers for the device they are hosted on. They also have an attached command queue, which allows out-of-order execution of all operations. Once you have a context, you can create a kernel in the context with `oclSimpleKernel`.

Usage

```
oclContext(device = "default", precision = c("best", "single", "double"))
```

Arguments

`device`	Device object as obtained from `oclDevices` or a type as in `oclDevices`. In this case, a suitable device of the given type will be selected automatically.
`precision`	Default precision of the context. This is the precision that will be chosen by default for `numeric` buffers and kernels with `numeric` output mode.

Value

An OpenCL context.

Author(s)

Aaron Puchert

See Also

oclDevices, oclSimpleKernel

Examples

```
library(OpenCL)
cat("== Platforms:\n")
(platforms <- oclPlatforms())
if (length(platforms)) {
    cat("== Devices:\n")
    ## pick the first platform
    print(devices <- oclDevices(platforms[[1]]))
    if (length(devices)) {
        cat("== Context:\n")
        ## pick the first device
        print(ctx <- oclContext(devices[[1]]))
    }
    cat("== Default context:\n")
    ## Note that context can find device on its own
    ## (may be different from above if you have multiple devices)
    print(c2 <- oclContext())
}
```

---

oclDevices — *Get a list of OpenCL devices.*

Description

oclDevices retrieves a list of OpenCL devices for the given platform.

Usage

```
oclDevices(platform = oclPlatforms()[[1]],
           type = c("all", "cpu", "gpu", "accelerator", "default"))
```

Arguments

platform	OpenCL platform (see oclPlatforms)
type	Desired device type, character vector of length one. Valid values are "cpu", "gpu", "accelerator", "all", "default". Partial matches are allowed.

Value

List of devices. May be empty.

Author(s)

Simon Urbanek

See Also

oclPlatforms

Examples

```
p <- oclPlatforms()
if (length(p))
    print(oclDevices(p[[1]], "all"))
```

---

oclInfo *Retrieve information about an OpenCL object.*

Description

Some OpenCL obejcts have information tokens associated with them. For example the device obejct has a name, vendor, list of extensions etc. `oclInfo` returns a list of such properties for the given object.

Usage

```
oclInfo(item)
## S3 method for class 'clDeviceID'
oclInfo(item)
## S3 method for class 'clPlatformID'
oclInfo(item)
## S3 method for class 'list'
oclInfo(item)
```

Arguments

`item`	object to retrieve information properties from

Value

List of properties. The properties vary by object type. Some common properties are `"name"`, `"vendor"`, `"version"`, `"profile"` and `"exts"`.

Author(s)

Simon Urbanek

Examples

```
p <- oclPlatforms()
if (length(p)) {
    cat("== Platform information:\n")
    print(oclInfo(p[[1]]))
    d <- oclDevices(p[[1]])
    if (length(d)) {
        cat("== Device information:\n")
        print(oclInfo(d))
    }
}
```

`oclMemLimits` *OpenCL Memory Management and Limits*

Description

`oclMemLimits` manages the memory limits used internally to aid with R garbage collection and reports used buffer memory.

Usage

```
oclMemLimits(trigger = NULL, high = NULL)
```

Arguments

`trigger`	size specification for trigger limit or `NULL` to not change
`high`	size specification for high mark limit or `NULL` to not change

Details

In principle the memory management is simple: as long as a reference to a GPU object exists in R, that object is retained. As soon as R removes the reference object, the corresponding GPU object is released. This is sounds easy execpt for one important detail: R only releases unused objects when a garbage collection is run (see `gc`), but R does not know about the GPU memory so it may not decide that it is necessary if little R memory is used.

As a user, you can explicitly call `gc()` to force all unused objects to be collected, but garbage collection is expensive so it may impact your computation. Therefore OpenCL tracks allocated memory sizes used by `clBuffer` buffers and will trigger R garbage collection automatically if certain limits are reached.

There are two limits: `trigger` limit and `high` limit. The `trigger` limit is the threshold at which OpenCL will attempt to run garbage collection. This limit is checked before any buffer allocation. Once this limit is exceeded, OpenCL will run `gc()` to attempt to free memory. Hoever, if the current operation actually does require a lot of memory, no GPU memory may be freed. In that case running garbage collection would be wasteful, therefore OpenCL will disable further GC until the `high` limit is reached. Beyond that limit GC is always run.

The limit size specifications can be one of the following: a positive integer numeric (in bytes) or a scalar string consisting of the integer numeric and an optional unit suffix. The following suffixes are supported: `"k"`, `"m"` and `"g"` - corresponding powers of 1024. Note that the reported sizes are always in bytes represented as numerics.

Value

List with followng components:

- `trigger` — active trigger limit (in bytes) or 0 if not active
- `high` — active trigger limit (in bytes) or 0 if not active
- `used` — number of bytes currently allocated in clBuffers on the GPU
- `in.zone` — logical, `TRUE` if garbage collection is disabled due to the inability to reduce usage under `trigger`, i.e., the usage is between `trigger` and `high`

Note

Currently the default is to not enable the automatic garbage collection, becasue it is experimental and best settings will cary by the hardware used,but that is likey to change. It can always be disabled with `oclMemLimits(0,0)`.

IMPORTANT: The current tracking is global to OpenCL, so it is based on all the memory used across all devices.

Author(s)

Simon Urbanek

See Also

`gc`, `clBuffer`

Examples

```
oclMemLimits()
```

`oclPlatforms` *Retrieve available OpenCL platforms.*

Description

`oclPlatforms` retrieves all available OpenCL platforms.

Usage

```
oclPlatforms()
```

Value

List of available OpenCL platforms. If using OpenCL with Installable Client Driver (ICD) support, the result can be an empty list if no vendor ICD can be found. A warning is also issued in that case.

Author(s)

Simon Urbanek

See Also

`oclDevices`

Examples

```
print(oclPlatforms())
```

`oclRun` *Run a kernel using OpenCL.*

Description

`oclRun` is used to execute code that has been compiled for OpenCL.

Usage

```
oclRun(kernel, size, ..., dim = size)
```

Arguments

`kernel`	Kernel object as obtained from `oclSimpleKernel`
`size`	Length of the output vector
`...`	Additional arguments passed to the kernel
`dim`	Numeric vector describing the global work dimensions, i.e., the index range that the kernel will be run on. The kernel can use `get_global_id(n)` to obtain the `(n + 1)`-th dimension index and `get_global_size(n)` to get the dimension. OpenCL standard supports only up to three dimensions, you can use use index vectors as arguments if more dimensions are required. Note that `dim` is not necessarily the dimension of the result although it can be.

Details

oclRun pushes kernel arguments, executes the kernel and retrieves the result. The kernel is expected to have either __global double * or __global float * type (write-only) as the first argument which will be used for the result and const unsigned int second argument denoting the result length. All other arguments are assumed to be read-only and will be filled according to the ... values. These can either be OpenCL buffers as generated by clBuffer for pointer arguments, or scalar values (vectors of length one) for scalar arguments. Only integer (int), and numeric (double or float) scalars and OpenCL buffers are supported as kernel arguments. The caller is responsible for matching the argument types according to the kernel in a way similar to .C and .Call.

Note that the kernel must match the input types as well, so typically as.clBuffer() should include the mode (e.g., "numeric") to match the kernel and/or explicit as.numeric() coercion should be used.

Value

The resulting buffer of length size.

Author(s)

Simon Urbanek, Aaron Puchert

See Also

oclSimpleKernel, clBuffer

Examples

```
library(OpenCL)
## Only proceed if this machine has at least one OpenCL platform
if (length(oclPlatforms())) {
ctx = oclContext(precision="single")

code = c("
__kernel void dnorm(
  __global numeric* output,
 const unsigned int count,
  __global numeric* input,
 const numeric mu, const numeric sigma)
{
  size_t i = get_global_id(0);
  if(i < count)
    output[i] = exp((numeric) (-0.5 * ((input[i] - mu) / sigma) * ((input[i] - mu) / sigma)))
      / (sigma * sqrt((numeric) (2 * 3.14159265358979323846264338327950288 )) );
}")
k.dnorm <- oclSimpleKernel(ctx, "dnorm", code)
f <- function(x, mu=0, sigma=1)
  as.numeric(oclRun(k.dnorm, length(x), as.clBuffer(x, ctx, "numeric"), mu, sigma))

## expect differences since the above uses single-precision but
## it should be close enough
```

```
f(1:10/2) - dnorm(1:10/2)

## does the device support double-precision?
if (any("cl_khr_fp64" == oclInfo(attributes(ctx)$device)$exts)) {
  k.dnorm <- oclSimpleKernel(ctx, "dnorm", code, "double")
  f <- function(x, mu=0, sigma=1)
    as.numeric(oclRun(k.dnorm, length(x), as.clBuffer(x, ctx, "double"), mu, sigma))

  ## probably not identical, but close...
  f(1:10/2) - dnorm(1:10/2)
} else cat("\nSorry, your device doesn't support double-precision\n")

## Note that in practice you can use precision="best" in the first
## example which will pick "double" on devices that support it and
## "single" elsewhere
}
```

`oclSimpleKernel` *Create and compile OpenCL kernel code.*

Description

Creates a kernel object by compiling the supplied code. The kernel can then be used in `oclRun`.

Usage

```
oclSimpleKernel(context, name, code,
    output.mode = c("numeric", "single", "double", "integer"))
```

Arguments

`context`	Context (as created by `oclContext`) to compile the kernel in.
`name`	Name of the kernel function - must match the name used in the supplied code.
`code`	Character vector containing the code. The code will be concatenated (as-is, no newlines are added!) by the engine.
`output.mode`	Mode of the output argument of the kernel, as in `clBuffer`. This can be one of "single", "double", "integer", or "numeric". The default value "numeric" maps to the default precision of the context. The kernel code may use a type `numeric` that is typedef'd to the given precision, i.e. either `float` or `double`. The OpenCL extension `cl_khr_fp64` will be enabled automatically in the second case, so you don't have to add the pragma yourself.

Details

oclSimpleKernel builds the program specified by code and creates a kernel from the program.

The kernel built by this function is simple in that it can have exactly one vector output and arbitrarily many inputs. The first argument of the kernel must be __global double* or __global float* for the output and the second argument must be const unsigned int for the length of the output vector. Additional numeric scalar arguments are assumed to have the same mode as the output, i.e. if the output shall have "double" precision, then numeric scalar arguments are assumed to be double values, similarly for single-precision. All additional arguments are optional. See oclRun for an example of a simple kernel.

Note that building a kernel can take substantial amount of time (depending on the OpenCL implementation) so it is generally a good idea to compile a kernel once and re-use it many times.

Value

Kernel object that can be used by oclRun.

Author(s)

Simon Urbanek, Aaron Puchert

See Also

oclContext, oclRun

Index