SYCL Reference

Intel

May 01, 2020

CONTENTS

CHAPTER

ONE

LANGUAGE

SYCL programs are C++ programs. No extensions are added to the language.

Todo: C++ version mininum

1.1 Keywords

SYCL does not add any keywords to the C++ language.

1.2 Preprocessor Directives and Macros

Standard C++ preprocessing directives and macros are supported by the compiler. In addition, the SYCL Specification defines the SYCL specific preprocessor directives and macros.

The following preprocessor macros are supported by the compiler.

Macro	Value	Description
`SYCL_DUMP_IMAGES`	`true` or `false`	Instructs the runtime to dump the device image
`SYCL_USE_KERNEL_SPV`	<device binary>	Employ device binary to fulfill kernel launch request
`SYCL_PROGRAM_BUILD_OPTIONS`	<options>	Used to pass additional options for device program building.

1.3 Standard Library Classes Required for the Interface

The SYCL specification documents a facility to enable vendors to provide custom optimized implementations. Implementations require aliases for several STL interfaces. These are summarized as follows:

Todo: add STL interfaces

CHAPTER TWO

PROGRAMMING INTERFACE

For further details on SYCL, see the SYCL Specification.

2.1 Header File

A single header file must be included:

```
#include "sycl.hpp"
```

2.2 Namespaces

Unless otherwise noted, all symbols should be prefixed with the `cl::sycl` namespace. `buffer` is `cl::sycl::buffer`, and `info::device::name` is `cl::sycl::info::device::name`.

2.3 Standard Library Classes

2.4 Runtime classes

2.4.1 Device Selectors

Devices selectors allow the SYCL runtime to choose the device. Built-in device selectors allow the user to restrict the type of device. Users can also specify their own heuristics for choosing the device by implementing their own `device_selector`.

Device selection class

Device selector interface

```
class device_selector();
```

`device_selector` is an abstract class that cann

Member functions

(constructor)	constructs a device_selector
(destructor)	destroys the device_selector
select_device	

Nonmember functions

operator()	

(constructor)

```
device_selector(const device_selector &rhs);
```

```
device_selector &operator=(const device_selector &rhs);
```

Constructs a `device_selector` from another `device_selector`

Parameters

`rhs` - device

select_device

```
device select_device() const;
```

Returns

operator()

```
virtual int operator()(const device &device) const = 0;
```

Returns

Derived device selector

default_selector

```
class default_selector;
```

Selects a SYCL device based on a implementation-defined heuristic. Selects a *host device* if no other device can be found.

gpu_selector

```
class gpu_selector;
```

Selects a GPU.

Exceptions

Throws a `runtime_error` if a GPU device cannot be found

Example

```
#include <CL/sycl.hpp>

using namespace cl::sycl;

int main() {
  device d;

  try {
    d = device(gpu_selector());
  } catch (exception const& e) {
    std::cout << "Cannot select a GPU\n" << e.what() << "\n";
    std::cout << "Using a CPU device\n";
    d = device(cpu_selector());
  }

  std::cout << "Using " << d.get_info<sycl::info::device::name>();
}
```

Output on a system without a GPU

accelerator_selector

```
class accelerator_selector;
```

Selects an accelerator.

Exceptions

Throws a `runtime_error` if an accelerator device cannot be found.

Example

See *Example* for the use of a pre-defined selector.

cpu_selector

```
class cpu_selector;
```

Select a CPU device.

Exceptions

Throws a `runtime_error` if a CPU device cannot be found.

Example

See *Example* for the use of a pre-defined selector.

host_selector

```
class host_selector;
```

Example

See *Example* for the use of a pre-defined selector.

2.4.2 Platform class

```
class platform;
```

Abstraction for SYCL platform.

Member functions

(constructor)	constructs a platform
destructor	destroys a platform
get	returns OpenCL platform ID
get_devices	returns devices bound to the platform
get_info	queries properties of the platform
has_extension	checks if platform has an extension
is_host	checks if platform has a SYCL host device

Nonmember functions

get_platforms	returns available platforms

Example

Enumerate the platforms and the devices they contain.

```
#include <CL/sycl.hpp>

namespace sycl = cl::sycl;

int main() {
  auto platforms = sycl::platform::get_platforms();

  for (auto &platform : platforms) {
    std::cout << "Platform: "
              << platform.get_info<sycl::info::platform::name>()
              << std::endl;

    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
                << device.get_info<sycl::info::device::name>()
                << std::endl;
    }
  }

  return 0;
}
```

Output:

```
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
  Device: Intel(R) FPGA Emulation Device
Platform: Intel(R) OpenCL
  Device: Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
Platform: Intel(R) CPU Runtime for OpenCL(TM) Applications
  Device: Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
Platform: SYCL host platform
  Device: SYCL host device
```

(constructor)

```
platform();
```

`explicit platform(cl_platform_id platformID);` [1]

`explicit platform(const device_selector &deviceSelector);` [2]

Constructs a platform handle.

[1] Constructs a SYCL platform that retains an OpenCL id

[2] Selects a platform that contains the desired device

Parameters

`platformID` - OpenCL platform ID
`deviceSelector` - Platform must contain the selected device

get

```
cl_platform_id get() const;
```

Returns OpenCL platform id used in the constructor.

get_devices

```
vector_class<device> get_devices(
   info::device_type = info::device_type::all) const;
```

Returns vector of devices of the requested type

Parameters

`device_type` - limits type of device returned

Returns

`vector` containing devices of the specified type bound to the platform.

Example

See *platform-example*.

get_info

```
template< info::platform param >
typename info::param_traits<info::platform, param>::return_type get_info() const;
```

Returns information about the platform, as specified by `param`.

Returns

Requested information

Example

See *platform-example*.

has_extension

```
bool has_extension(const string_class &extension) const;
```

Checks if the platform has the requested extension.

Parameters

`extension` -

Returns

`true` if the platform has `extension`

is_host

```
bool is_host() const;
```

Checks if the platform contains a SYCL *host device*

Returns

`true` if the platform contains a host device

get_platforms

```
static vector_class<platform> get_platforms();
```

Returns vector of platforms

Returns

vector_class containing SYCL platforms bound to the system

Example

See *platform-example*.

2.4.3 Context class

```
class context;
```

Member functions

(constructor)	constructs a context
get	returns OpenCL conext ID
is_host	checks if contains a SYCL host device
get_platform	
get_devices	returns devices bound to the context
get_info	queries properties

(constructor)

```
explicit context(const property_list &propList = {});
```

```
context(async_handler asyncHandler,
        const property_list &propList = {});
```

```
context(const device &dev, const property_list &propList = {});
```

```
context(const device &dev, async_handler asyncHandler,
        const property_list &propList = {});
```

```
context(const platform &plt, const property_list &propList = {});
```

```
context(const platform &plt, async_handler asyncHandler,
        const property_list &propList = {});
```

```
context(const vector_class<device> &deviceList,
        const property_list &propList = {});
```

```
context(const vector_class<device> &deviceList,
        async_handler asyncHandler, const property_list &propList = {});
```

```
context(cl_context clContext, async_handler asyncHandler = {});
```

Parameters

propList -
asyncHandler -
dev -
plt -
deviceList -

get

```
cl_context get() const;
```

Returns

is_host

```
bool is_host() const;
```

Returns

get_platform

```
platform get_platform() const;
```

Returns

get_devices

```
vector_class<device> get_devices() const;
```

Returns

get_info

```
template <info::context param>
typename info::param_traits<info::context, param>::return_type get_info() const;
```

Returns

2.4.4 Device class

Device interface

```
class device;
```

An abstract class representing various models of SYCL devices

Member functions

(constructor)	
(destructor)	
get	
is_host	
is_cpu	
is_gpu	
is_accelerator	
get_platform	
get_info	
has_extension	
create_sub_devices	

Nonmember functions

get_devices	

(constructor)

`device();` [4]

`explicit device(cl_device_id deviceId);` [5]

`explicit device(const device_selector &deviceSelector);` [6]

Parameters

`deviceID` - OpenCL device id

`deviceSelector` - Device selector

[4] Default Constructor. Constructs a device object in host mode.

[5] Constructs a device object from another device object and retains the cl_device_id object if the device is not in host mode.

[6] Use deviceSelector to choose device

get

```
cl_device_id get() const;
```

Return the cl_device_id of the underlying OpenCL platform

Returns

cl_device_id of underlying OpenCL platform

is_host

```
bool is_host() const;
```

Checks if the device is a SYCL host device

Returns

True if the device is a *host device*, false otherwise.

is_cpu

```
bool is_cpu() const;
```

Checks if the device is a CPU

Returns

True if the device is a CPU, false otherwise

is_gpu

```
bool is_gpu() const;
```

Checks if the device is a GPU

Returns

True if the device is a GPU, false otherwise

is_accelerator

```
bool is_accelerator() const;
```

Checks if the device is a GPU

Returns

True if the device is a GPU, false otherwise

get_platform

```
platform get_platform() const;
```

Returns the platform that contains the device

Returns

Platform object

get_info

```
template <info::device param>
typename info::param_traits<info::device, param>::return_type
get_info() const;
```

Queries the device for information specific to `param`.

Template parameters

`param` - refer to info::device table

Returns

Device information

Example

See *Example*.

has_extension

```
bool has_extension(const string_class &extension) const;
```

Check

Parameters

`extension` - name of extension

Returns

create_sub_devices

```
template <info::partition_property prop>
vector_class<device> create_sub_devices(size_t nbSubDev) const; [7]
```

```
template <info::partition_property prop>
vector_class<device> create_sub_devices(const vector_class<size_t> &counts)␣
↪const; [8]
```

```
template <info::partition_property prop>
vector_class<device> create_sub_devices(info::affinity_domain affinityDomain)␣
↪const; [9]
```

Parameters

nbSubDev - counts - affinityDomain -

Returns

get_devices

```
static vector_class<device> get_devices(
    info::device_type deviceType = info::device_type::all);
```

Returns

2.4.5 queue

```
class queue;
```

[7] Available only when prop == info::partition_property::partition_equally

[8] Available only when prop == info::partition_property::partition_by_counts

[9] Available only when prop == info::partition_property::partition_by_affinity_domain

Member functions

(constructor)	
(destructor)	
get	
get_context	
get_device	
get_info	
is_host	
submit	
wait	
wait_and_throw	
throw_asynchronous	

(constructor)

```
explicit queue(const property_list &propList = {});
```

```
explicit queue(const async_handler &asyncHandler,
               const property_list &propList = {});
```

::

> **explicit queue(const device_selector &deviceSelector,** const property_list &propList = {});

::

> **explicit queue(const device_selector &deviceSelector,** const async_handler &asyncHandler, const property_list &propList = {});

```
explicit queue(const device &syclDevice, const property_list &propList = {});
```

```
explicit queue(const device &syclDevice, const async_handler &asyncHandler,
               const property_list &propList = {});
```

```
explicit queue(const context &syclContext,
               const device_selector &deviceSelector,
               const property_list &propList = {});
```

```
explicit queue(const context &syclContext,
               const device_selector &deviceSelector,
               const async_handler &asyncHandler,
               const property_list &propList = {});
```

```
explicit queue(const context &syclContext,
               const device &syclDevice,
               const property_list &propList = {});
```

```
explicit queue(const context &syclContext, const device &syclDevice,
               const async_handler &asyncHandler,
               const property_list &propList = {});
```

```
explicit queue(cl_command_queue clQueue, const context& syclContext,
               const async_handler &asyncHandler = {});
```

get

```
cl_command_queue get() const;
```

Returns

get_context

```
context get_context() const;
```

Returns

get_device

```
device get_device() const;
```

Returns

is_host

```
bool is_host() const;
```

Returns

get_info

```
template <info::queue param>
typename info::param_traits<info::queue, param>::return_type get_info() const;
```

Returns

submit

```
template <typename T>
event submit(T cgf);
```

```
template <typename T>
event submit(T cgf, const queue &secondaryQueue);
```

Parameters

cgf -
secondaryQueue -

Returns

wait

```
void wait();
```

wait_and_throw

```
void wait_and_throw();
```

throw_asynchronous

```
void throw_asynchronous();
```

2.4.6 event

```
class event;
```

Member functions

(constructor)	
(destructor)	
cl_event_get	
is_host	
get_wait_list	
wait	
wait_and_throw	
get_info	
get_profiling_info	

(constructor)

```
event();
```

```
event(cl_event clEvent, const context& syclContext);
```

cl_event_get

```
cl_event get();
```

Returns

is_host

```
bool is_host() const;
```

Returns

get_wait_list

```
vector_class<event> get_wait_list();
```

Returns

wait

```
void wait();
```

```
static void wait(const vector_class<event> &eventList);
```

wait_and_throw

```
void wait_and_throw();
```

```
static void wait_and_throw(const vector_class<event> &eventList);
```

get_info

```
template <info::event param>
typename info::param_traits<info::event, param>::return_type get_info() const;
```

Returns

get_profiling_info

```
template <info::event_profiling param>
typename info::param_traits<info::event_profiling, param>::return_type get_profiling_
↪info() const;
```

Returns

2.5 Data access

2.5.1 Buffers

Buffer interface

Buffer properties

2.5.2 Images

Image interface

Image properties

2.5.3 Accessors

accessor

```
template<
    typename dataT,
    int dimensions,
    access::mode accessmode,
    access::target accessTarget = access::target::global_buffer,
    access::placeholder isPlaceholder = access::placeholder::false_t
> class accessor;
```

A DPC++ `accessor` encapsulates reading and writing memory objects which can be buffers, images, or device local memory. Creating an accessor requires a method to reference the desired access target. Construction also requires the type of the memory object, the dimensionality of the memory object, the access mode, and a placeholder argument.

Template parameters

`dataT` - type of buffer element
`dimensions`- dimensionality of buffer
`accessmode` - type of access
`accessTarget` - type of memory
`isPlaceholder` - placeholder

Member types

`value_type`	dataT
`reference`	dataT&
`const_reference`	const dataT&

Member functions

(constructor)	constructs an accessor
(destructor)	destroys the accessor
is_placeholder	
get_size	
get_count	
get_range	
get_offset	

get_size

```
size_t get_size() const
```

Description

get_count

2.5.4 Address space classes

2.5.5 Samplers

2.6 Expressing parallelism

2.6.1 Ranges and index space identifiers

2.6.2 Command group handler class

2.6.3 Invoking kernels

2.6.4 Memory operations

2.6.5 Kernel class

2.6.6 Program class

2.7 Error handling

2.7.1 Exception Class Interface

2.8 Data types

2.8.1 Scalar types

2.8.2 Vector types

2.9 Synchronization and atomics

2.9.1 Synchronization types

2.10 Stream class

2.10.1 Stream class interface

2.11 Built-in functions

2.11.1 Math functions

2.11.2 Integer functions

2.11.3 Common functions

2.11.4 Geometric functions

2.11.5 Relational functions

CHAPTER

THREE

STYLE GUIDE

We try to follow the style of cppreference. See style manual.

3.1 ClassExample

Parameters

Parameters

```
template<
    class T1
    class T2
> class ClassExample;
```

This is the description of the class. It is followed by a set of tables for template parameters and class members. This is followed by the member functions, one section each.

Template parameters

`T1` - description of parameter

`T2` - description of parameter

Member functions

(constructor)	constructs a ClassExample
fun1	checks . . .

Non-member functions

operator+	Adds . . .
fun3	Queries . . .

Example

Describe the example. . .

```
#include <CL/sycl.hpp>

namespace sycl = cl::sycl;

int main() {
  auto platforms = sycl::platform::get_platforms();

  for (auto &platform : platforms) {
    std::cout << "Platform: "
              << platform.get_info<sycl::info::platform::name>()
              << std::endl;

    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
                << device.get_info<sycl::info::device::name>()
                << std::endl;
    }
  }

  return 0;
}
```

3.1.1 (constructor)

```
ClassExample();
```

`ClassExample(int a);` [1]

`ClassExample(int a, int b);` [2]

Description of the function. The parameters are in a table below. We have a single table for all the overloads.

[1] Describe constructor with one arg

[2] Describe constructor with two args

Parameters

a - An argument called a
b - An argument called b

Description of the functions. Overloads are grouped together and may have footnotes for overload-specific description.

Template parameters

T - A parameter called T

Parameters

a - A parameter called a

Example

A member function can have its own example

3.1.2 operator+

3.1.3 fun3

CHAPTER

FOUR

GLOSSARY

accelerator Specialized component containing compute resources that can quickly execute a subset of operations. Examples include CPU, FPGA, GPU. See also: *device*

accessor Communicates the desired location (host, device) and mode (read, write) of access.

application scope Code that executes on the host.

buffers Memory object that communicates the type and number of items of that type to be communicated to the device for computation.

command group scope Code that acts as the interface between the host and device.

command queue Issues command groups concurrently.

compute unit A grouping of processing elements into a 'core' that contains shared elements for use between the processing elements and with faster access than memory residing on other compute units on the device.

device An accelerator or specialized component containing compute resources that can quickly execute a subset of operations. A CPU can be employed as a device, but when it is, it is being employed as an accelerator. Examples include CPU, FPGA, GPU. See also: *accelerator*

device code Code that executes on the device rather than the host. Device code is specified via lambda expression, functor, or kernel class.

fat binary Application binary that contains device code for multiple devices. The binary includes both the generic code (SPIR-V representation) and target specific executable code.

fat library Archive or library of object code that contains object code for multiple devices. The fat library includes both the generic object (SPIR-V representation) and target specific object code.

fat object File that contains object code for multiple devices. The fat object includes both the generic object (SPIR-V representation) and target specific object code.

host A CPU-based system (computer) that executes the primary portion of a program, specifically the application scope and command group scope.

host device A SYCL device that is always present and usually executes on the host CPU.

host code Code that is compiled by the host compiler and executes on the host rather than the device.

images Formatted opaque memory object that is accessed via built-in function. Typically pertains to pictures comprised of pixels stored in format like RGB.

kernel scope Code that executes on the device.

nd-range Short for N-Dimensional Range, a group of kernel instances, or work item, across one, two, or three dimensions.

processing element Individual engine for computation that makes up a compute unit.

single source Code in the same file that can execute on a host and accelerator(s).

SPIR-V Binary intermediate language for representing graphical-shader stages and compute kernels.

sub-group Sub-groups are an Intel extension.

work-group Collection of work-items that execute on a compute unit.

work-item Basic unit of computation in the oneAPI programming model. It is associated with a kernel which executes on the processing element.

INDEX