SYCL Reference

Intel

Apr 26, 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

The Data Parallel C++ (DPC++) programming language and runtime consists of a set of C++ classes, templates, and libraries used to express a DPC++program. This chapter provides a summary of the key classes, templates, and runtime libraries used to program.

2.1 Header File

2.2 Namespaces

2.3 Class Descriptions

The following sections provide further details on these items. These sections do not provide the exhaustive details found in the SYCL Specification. Instead, these sections provide:

- A summary that includes a description and the purpose
- Comments on the different constructors, if applicable
- Member function information, if applicable
- Special cases to consider with the DPC++ implementation compared to the SYCL Specification

For further details on SYCL, see the SYCL Specification.

2.3.1 Runtime Classes

2.3.1.1 `device_selector`

```
class device_selector();
```

Member functions

(constructor)	
(destructor)	
select_device	

Non-member functions

operator()	

(constructor)

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

(destructor)

```
virtual ~device_selector();
```

select_device

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

Return value

operator()

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

Return value

2.3.1.2 platform

```
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

Non-member functions

get_platforms	returns available platforms

Example

Demonstrates several methods for `platform`

```
#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.

Arguments

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

Arguments

device_type	limits type of device returned

Return value

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

Example

See *Example*.

[1] Constructs a SYCL platform that retains an OpenCL id
[2] Selects a platform that contains the desired device

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`.

Return value

Requested information

Example

See *Example*.

has_extension

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

Checks if the platform has the requested extension.

Arguments

extension	

Return value

`true` if the platform has `extension`

is_host

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

Checks if the platform contains a SYCL *host device*

Return value

`true` if the platform contains a host device

get_platforms

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

Returns vector of platforms

Return value

vector_class containing SYCL platforms bound to the system

Example

See *Example*.

2.3.1.3 context

```
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 = {});
```

Arguments

propList	
asyncHandler	
dev	
plt	
deviceList	

get

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

Return value

is_host

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

Return value

get_platform

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

Return value

get_devices

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

Return value

get_info

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

Return value

2.3.1.4 device

```
class device;
```

Member functions

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

Non-member functions

get_devices	

(constructor)

```
device();
explicit device(cl_device_id deviceId);
explicit device(const device_selector &deviceSelector);
```

Arguments

deviceID	
deviceSelector	

get

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

Return value

is_host

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

Return value

is_cpu

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

Return value

is_gpu

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

Return value

is_accelerator

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

Return value

get_platform

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

Return value

get_info

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

Return value

Example

See *Example*.

has_extension

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

Arguments

extension	

Return value

create_sub_devices

```
template <info::partition_property prop>
vector_class<device> create_sub_devices(size_t nbSubDev) const; [4]

template <info::partition_property prop>
vector_class<device> create_sub_devices(const vector_class<size_t> &counts)␣
↪const; [5]

template <info::partition_property prop>
vector_class<device> create_sub_devices(info::affinity_domain affinityDomain)␣
↪const; [6]
```

Arguments

nbSubDev	
counts	
affinityDomain	

[4] Available only when prop == info::partition_property::partition_equally
[5] Available only when prop == info::partition_property::partition_by_counts
[6] Available only when prop == info::partition_property::partition_by_affinity_domain

Return value

get_devices

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

Return value

2.3.1.5 queue

```
class queue;
```

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,
```

(continues on next page)

(continued from previous page)

```
              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;
```

Return value

get_context

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

Return value

get_device

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

Return value

is_host

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

Return value

get_info

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

Return value

submit

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

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

Arguments

cgf	
secondaryQueue	

Return value

wait

```
void wait();
```

wait_and_throw

```
void wait_and_throw();
```

throw_asynchronous

```
void throw_asynchronous();
```

2.3.1.6 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();
```

Return value

is_host

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

Return value

get_wait_list

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

Return value

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;
```

Return value

get_profiling_info

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

Return value

2.3.2 Data Access

2.3.2.1 accessor

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

```
size_t get_size() const [1]
size_t get_size(int b) const [2]
size_t get_size(int c, int d) const [3]
```

Description

[1] No arguments
[2] single argument
[3] 2 arguments

CHAPTER

THREE

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