SYCL™ (“シクル” と読みます) は、汎用プログラミングにより、さまざまなデバイスでカーネルコードの最適化を加速し、高レベルのアプリケーション・ソフトウェアを簡素にコーディングすることを可能にします。

開発者はネイティブ・アクセラレーション API よりも高レベルでプログラミングを行いますが、ネイティブ・アクセラレーション API とシームレスに統合することで、常に低レベルのコードにアクセスできます。

このリファレンス・ガイドのすべての定義は sycl 名前空間にあります。
[n.n] は、khronos.org/registry/sycl にある SYCL 2020 (リビジョン 2) 仕様のセクション番号を示します。

共通インターフェイス

共通参照セマンティクス [4.5.2]

T は、accessor、buffer、context、device、device_image、event、host_accessor、host_[un]sampled_image_accessor、kernel、kernel_id、kernel_bundle、local_accessor、platform、queue、[un]sampled_image、sampled_image_accessor です。

```
T(const T &rhs);
T(T &&rhs);
T &operator=(const T &rhs);
T &operator=(T &&rhs);
~T();
friend bool operator==(const T &lhs, const T &rhs);
friend bool operator!=(const T &lhs, const T &rhs);
```

共通値セマンティクス [4.5.2]

T は、id、range、item、nd_item、h_item、group、sub_group または nd_range です。

```
friend bool operator==(const T &lhs, const T &rhs);
friend bool operator!=(const T &lhs, const T &rhs);
```

プロパティー [4.5.4]

次に示す SYCL ランタイムクラスの各コンストラクターには、プロパティーを含む property_list を提供するパラメーターがあります: accessor、buffer、host_accessor、host_[un]sampled_image_accessor、context、local_accessor、queue、[un]sampled_image、[un]sampled_image_accessor、stream、および usm_allocator。

```
template <typename propertyT>
struct is_property;

template <typename propertyT>
inline constexpr bool is_property_v = is_property<
  propertyT>::value;

template <typename propertyT, typename syclObjectT>
struct is_property_of;template <typename propertyT,
  typename syclObjectT>

inline constexpr bool is_property_of_v = is_property_of<
  propertyT, syclObjectT>::value;

class T {
  ...
  template <typename propertyT>
  bool has_property() const;
  template <typename propertyT>
  propertyT get_property() const;
  ...
};
class property_list {
 public:
   template <typename... propertyTN>
   property_list(propertyTN... props);
};
```

SYCL アプリケーションの構造 [3.2]

以下は、OpenCL アクセラレーターで並行して実行されるジョブをスケジュールする典型的な SYCL アプリケーションの例です。この例の USM バージョンは、このリファレンス・ガイドの 15 ページ目にあります。

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (オプション) SYCL 名の前に "sycl::" を
                      // 記述する必要はない

int main() {
  int data[1024]; // 処理するデータを割り当て

  queue myQueue; // ワークを送信するデフォルトキューを作成

  // すべての SYCL ワークを {} ブロックで囲むことで、
  // resultBuf のデストラクターが待機するため、ブロックを終了する前に
  // すべての SYCL タスクを完了する必要がある
  {
    // データ変数をバッファーで囲む
    buffer<int, 1> resultBuf { data, range<1> { 1024 } };

    // コマンドグループを作成して、キューにコマンドを発行
    myQueue.submit([&](handler & cgh) {

      // 初期化なしでバッファーへのアクセスを要求
      accessor writeResult { resultBuf, cgh, write_only, no_init };

      // 1024 個のワーク項目を持つ parallel_for タスクをキューに送信
      cgh.parallel_for(1024, [=](auto idx) {

        // 0 から始まるランク番号でそれぞれのバッファー要素を初期化
        writeResult[idx] = idx;

      }); // カーネル関数の終端

    });       // キューコマンドの終端

  }           // スコープの終端。キューに送信されたワークの完了を待機

  // 結果を出力
  for (int i = 0; i < 1024; i++) {
    std::cout <<"data["<< i << "] = " << data[i] << std::endl;
  return 0;
}
```

ヘッダーファイル
この例で示す SYCL の機能を利用するには、SYCL プログラムで <sycl/sycl.hpp> ヘッダーファイルをインクルードする必要があります。

名前空間
SYCL 名は sycl 名前空間で定義されています。

キュー
この行は、実行の基盤となるでデバイスを暗黙的に選択します。キュークラス関数 **[4.6.5]** については、2 ページ目をご覧ください。

バッファー
カーネルが使用するすべてのデータは、バッファーまたはイメージ内になければなりません。そうでなければ USM を使用します。バッファークラス関数 **[4.7.2]** については、3 ページ目をご覧ください。

アクセサー
アクセサークラス関数 **[4.7.6.x]** については、4 および 5 ページ目をご覧ください。

ハンドラー
ハンドラークラス関数 **[4.9.4]** については、9 ページ目をご覧ください。

スコープ
カーネルスコープは、デバイスコンパイラーによってコンパイルされ、デバイス上で実行される単一のカーネル関数を指定します。

コマンド・グループ・スコープは、カーネル関数とアクセサーで構成されるワーク単位を指定します。

アプリケーション・スコープは、コマンド・グループ・スコープ外のすべてのコードを指定します。

また、9 ページ目のリダクション・カーネルの作成方法の例と、16 ページ目のカーネルの呼び出し方法の例も参照してください。

デバイス選択 [4.6.1]

デバイスの選択は、すでにデバイスの特定のインスタンスを保持していなければ、デバイスセレクターを使用して選択します。デバイスセレクターの実際のインターフェイスは、呼び出し可能なデバイス定数参照を取得し、暗黙的に int に変換可能な値を返します。システムは各デバイスの関数を呼び出し、最も高い値のデバイスを選択します。

事前定義 SYCL デバイスセレクター

default_selector_v	ヒューリスティックによって選択されたデバイス。
gpu_selector_v	info::device::device_type::gpu デバイスタイプに従ってデバイスを選択。
cpu_selector_v	info::device::device_type::cpu デバイスタイプに従ってデバイスを選択。
accelerator_selector_v	アクセラレーター・デバイスを選択。

プラットフォーム・クラス [4.6.2]

プラットフォーム・クラスは、カーネル関数を実行する単一のプラットフォームをカプセル化します。プラットフォームは、単一のバックエンドに関連付けられます。

```
platform();
template <typename DeviceSelector>
explicit platform(const DeviceSelector &deviceSelector);
backend get_backend() const noexcept;
std::vector<device> get_devices(
  info::device_type = info::device_type::all) const;
template <typename param>
  typename param::return_type get_info() const;
template <typename param>
  typename param::return_type get_backend_info()
const;
bool has(aspect asp) const;
static std::vector<platform> get_platforms();
```

事前定義 SYCL デバイスセレクター

version	**戻り値: std::string**
info::platform::name	
info::platform::vendor	

デバイスクラス [4.6.2]

デバイスクラスは、カーネル関数を実行する単一のデバイスをカプセル化します。デバイスクラスのすべてのメンバー関数は同期されます。

```
device();
template <typename DeviceSelector>
  explicit device(const DeviceSelector &deviceSelector);
backend get_backend() const noexcept;
platform get_platform() const;
bool is_cpu() const;
bool is_gpu() const;
bool is_accelerator() const;
template <typename param>
  typename param::return_type
    get_info() const;
template <typename param>
  typename param::return_type
    get_backend_info() const;
bool has(aspect asp) const;
template <info::partition_property prop> std::vector<device>
  create_sub_devices(size_t count) const;
```

(次のページへ続く) ▶

©2021 Khronos Group - Rev. 0221

◀ デバイスクラス (続き)

```
template <info::partition_property prop> std::vector<device>
  create_sub_devices(const std::vector<size_t> &counts)
    const;

template <info::partition_property prop>std::vector<device>
  create_sub_devices(info::affinity_domain domain) const;

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

get_info() でデバイスを照会

次の記述子は info::device 名前空間にあります。

記述子	戻り値
device_type	info::device_type
vendor_id	uint32_t
max_compute_units	uint32_t
max_work_item_dimensions	uint32_t
max_work_item_sizes<1>	id<1>
max_work_item_sizes<2>	id<2>
max_work_item_sizes<3>	id<3>
max_work_group_size	size_t
max_num_sub_groups	uint32_t
sub_group_independent_-forward_progress	bool
sub_group_sizes	std::vector<size_t>
preferred_vector_width_char	uint32_t
preferred_vector_width_short	
preferred_vector_width_int	
preferred_vector_width_long	
preferred_vector_width_float	
preferred_vector_width_double	
preferred_vector_width_half	
native_vector_width_char	uint32_t
native_vector_width_short	
native_vector_width_int	
native_vector_width_long	
native_vector_width_float	
native_vector_width_double	
native_vector_width_half	
max_clock_frequency	uint32_t
address_bits	uint32_t
max_mem_alloc_size	uint64_t
max_read_image_args	uint32_t
max_write_image_args	uint32_t
image2d_max_width	size_t
image2d_max_height	size_t
image3d_max_width	size_t
image3d_max_height	size_t
image3d_max_depth	size_t
image_max_buffer_size	size_t

コンテキスト・クラス [4.6.3]

コンテキスト・クラスはコンテキストを表します。コンテキストはプラットフォームに関連付けられたデバイスグループと対話するため、バックエンド API が必要とするランタイムのデータ構造と状態を表します。

```
explicit context(const property_list &propList = {});
explicit context(async_handler asyncHandler,
  const property_list &propList = {});
explicit context(const device &dev,
  const property_list &propList = {});
explicit context(const device &dev, async_handler asyncHandler,
  const property_list &propList = {});
explicit context(const std::vector<device> &deviceList,
  const property_list &propList = {});
explicit context(const std::vector<device> &deviceList,
  async_handler asyncHandler,
  const property_list &propList = {});

backend get_backend() const noexcept;

template <typename param>
  typename param::return_type
  get_info() const;

platform get_platform() const;

std::vector<device> get_devices() const;

template <typename param>
  typename param::return_type
  get_backend_info() const;
```

get_info() でコンテキストを照会

次の記述子は info::contexr 名前空間にあります。

記述子	戻り値
platform	platform
devices	std::vector<device>
atomic_memory_order_capabilities	std::vector<memory_order>
atomic_fence_order_capabilities	std::vector<memory_order>
atomic_memory_scope_capabilities	std::vector<memory_scope>
atomic_fence_scope_capabilities	std::vector<memory_scope>

記述子	戻り値
max_samplers	uint32_t
max_parameter_size	size_t
mem_base_addr_align	uint32_t
half_fp_config	std::vector<info::fp_config>
single_fp_config	std::vector<info::fp_config>
double_fp_config	std::vector<info::fp_config>
global_mem_cache_type	info::global_mem_cache_type
global_mem_cache_line_size	uint32_t
global_mem_cache_size	uint64_t
global_mem_size	uint64_t
global_mem_cache_size	uint64_t
global_mem_size	uint64_t
local_mem_type	info::local_mem_type
local_mem_size	uint64_t
error_correction_support	bool
atomic_memory_order_capabilities	std::vector<memory_order>
atomic_fence_order_capabilities	std::vector<memory_order>
atomic_memory_scope_capabilities	std::vector<memory_scope>
atomic_fence_scope_capabilities	std::vector<memory_scope
profiling_timer_resolution	size_t
is_available	bool
execution_capabilities	std::vector <info::execution_capability>
built_in_kernel_ids	std::vector<kernel_id>
built_in_kernels	std::vector<std::string>
platform	platform
name	std::string
vendor	std::string

記述子	戻り値
driver_version	std::string
version	std::string
backend_version	std::string
aspects	std::vector<aspect>
printf_buffer_size	size_t
parent_device	device
partition_max_sub_devices	uint32_t
partition_properties	std::vector<info::partition_properties>
partition_affinity_domains	std::vector <info::partition_affinity_domain>
partition_type_property	info::partition_property
partition_type_affinity_domain	info::partition_affinity_domain

デバイスアスペクト [4.6.4.3]

デバイスアスペクトは列挙型クラスアスペクトで定義されます。主な列挙子を以下に示します。特定のバックエンドが追加のアスペクトを定義する場合があります。

cpu	online_compiler
gpu	online_linker
accelerator	queue_profiling
custom	usm_device_allocations
fp16、fp64	usm_host_allocations
emulated	usm_atomic_host_allocations
host_debuggable	usm_shared_allocations
atomic64	usm_atomic_shared_allocations
Image	usm_system_allocations

キュークラス [4.6.5]

キュークラスは、デバイス上のカーネルをスケジュールする単一のキューをカプセル化します。キューを使用してメンバー関数送信によりランタイムで実行されるコマンドグループを送信できます。デストラクターはブロックしないことに留意してください。

```
explicit queue(const property_list &propList = {});
explicit queue(const async_handler &asyncHandler,
  const property_list &propList = {});
template <typename DeviceSelector>
  explicit queue(const DeviceSelector &deviceSelector,
    const property_list &propList = {});
template <typename DeviceSelector>
  explicit queue(const DeviceSelector &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 = {});
template <typename DeviceSelector>
  explicit queue(const context &syclContext,
    const DeviceSelector &deviceSelector,
    const property_list &propList = {});
template <typename DeviceSelector>
  explicit queue(const context &syclContext,
    const DeviceSelector &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 = {});
backend get_backend() const noexcept;
context get_context() const;
device get_device() const;
bool is_in_order() const;
template <typename param>
  typename param::return_type
  get_info() const;
template <typename param>
  typename param::return_type
  get_backend_info() const;
template <typename T> event submit(T cgf);
template <typename T>
  event submit(T cgf, const queue &secondaryQueue);
void wait();
void wait_and_throw();
void throw_asynchronous();
```

get_info() でキューを照会

記述子	戻り値
info::queue::context	コンテキスト
info::queue::device	デバイス

ショートカット

```
template <typename KernelName, typename KernelType>
  event single_task(const KernelType &kernelFunc);

template <typename KernelName, typename KernelType>
  event single_task(event depEvent,
    const KernelType &kernelFunc);

template <typename KernelName, typename KernelType>
  event single_task(const std::vector<event> &depEvents,
    const KernelType &kernelFunc);

template <typename KernelName, int Dims, typename... Rest>
  event parallel_for(range<Dims> numWorkItems, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
  event parallel_for(range<Dims> numWorkItems,
    event depEvent, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
  event parallel_for(range<Dims> numWorkItems,
    const std::vector<event> &depEvents, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
  event parallel_for(nd_range<Dims> executionRange, Rest&&... rest);

template <typename KernelName, int Dims, typename... Rest>
  event parallel_for(nd_range<Dims> executionRange,
    event depEvent, Rest&&... rest);

template <typename KernelName, int Dims, typename... rest>
  event parallel_for(nd_range<Dims> executionRange,
    const std::vector<event> &depEvents, Rest&&...rest);
```

(続く) ▶

©2021 Khronos Group - Rev. 0221

◀ キュークラス (続き)

USM 関数

```
event memcpy(void* dest, const void* src, size_t numBytes);
event memcpy(void* dest, const void* src, size_t numBytes,
  event depEvent);
event memcpy(void* dest, const void* src, size_t numBytes,
  const std::vector<event> &depEvents);

template <typename T>
  event copy(T* dest, const T *src, size_t count);
template <typename T>
  event copy(T* dest, const T *src, size_t count,
    event depEvent);
template <typename T>
  event copy(T* dest, const T *src, size_t count,
    const std::vector<event> &depEvents);

event memset(void* ptr, int value, size_t numBytes);
event memset(void* ptr, int value, size_t numBytes,
  event depEvent);
event memset(void* ptr, int value, size_t numBytes,
  const std::vector<event> &depEvents);

template <typename T>
  event fill(void* ptr, const T& pattern, size_t count);

template <typename T>
  event fill(void* ptr, const T& pattern, size_t count,
    event depEvent);

template <typename T>
  event fill(void* ptr, const T& pattern, size_t count,
    const std::vector<event> &depEvents);

event prefetch(void* ptr, size_t numBytes);
event prefetch(void* ptr, size_t numBytes, event depEvent);
event prefetch(void* ptr, size_t numBytes,
  const std::vector<event> &depEvents);

event mem_advise(void *ptr, size_t numBytes, int advice);
event mem_advise(void *ptr, size_t numBytes, int advice,
  event depEvent);

event (void *ptr, size_t numBytes, int advice,
  const std::vector<event> &depEvents);
```

明示的なコピー関数

```
template <typename T_src, int dim_src,
      access_mode mode_src, target tgt_src,
      access::placeholder isPlaceholder, typename T_dest>
  event copy(accessor<T_src, dim_src, mode_src, tgt_src,
    isPlaceholder> src, std::shared_ptr<T_dest> dest);

template <typename T_src, typename T_dest,
      int dim_dest, access_mode mode_dest,
      target tgt_dest, access::placeholder isPlaceholder>
  event copy(std::shared_ptr<T_src> src, accessor<T_dest,
    dim_dest, mode_dest, tgt_dest isPlaceholder> dest);

template <typename T_src, int dim_src,
      access_mode mode_src, target tgt_src,
      access::placeholder isPlaceholder, typename T_dest>
  event copy(accessor<T_src, dim_src, mode_src, tgt_src,
    isPlaceholder> src, T_dest *dest);

template <typename T_src, typename T_dest,
      int dim_dest, access_mode mode_dest,
      target tgt_dest, access::placeholder isPlaceholder>
  event copy(const T_src *src, accessor<T_dest, dim_dest,
    mode_dest, tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src,
      access_mode mode_src, target tgt_src,
      access::placeholder isPlaceholder_src,
      typename T_dest, int dim_dest,
      access_mode mode_dest, target tgt_dest,
      access::placeholder isPlaceholder_dest>
  event copy(accessor<T_src, dim_src, mode_src, tgt_src,
    isPlaceholder_src> src, accessor<T_dest, dim_dest,
    mode_dest, tgt_dest, isPlaceholder_dest> dest);

template <typename T, int dim, access_mode mode,
      target tgt, access::placeholder isPlaceholder>
  event update_host(accessor<T, dim, mode, tgt,
    isPlaceholder> acc);

template <typename T, int dim, access_mode mode,
      target tgt, access::placeholder isPlaceholder>
  event fill(accessor<T, dim, mode, tgt, isPlaceholder> dest,
    const T &src);
```

キュー・プロパティーのクラス・コンストラクター

property::queue::enable_profiling::enable_profiling();
property::queue::in_order::in_order();

get_info() で照会

記述子	戻り値
info::queue::context	**context**
info::queue::device	**device**

イベントクラス [4.6.6]

イベントには、ランタイムで実行される操作の状態を示すオブジェクトが含まれます。

```
event()
backend get_backend() const noexcept;
std::vector<event> get_wait_list();
void wait();
static void wait(const std::vector<event> &eventList);
void wait_and_throw();
static void wait_and_throw(
  const std::vector<event> &eventList);

template <typename param>
    typename param::return_type
    get_info() const;

template <typename param>
  typename param::return_type
  get_backend_info() const;

template <typename param>
  typename param::return_type
  get_profiling_info() const;
```

get_info() でイベントを照会

記述子	戻り値
info::event::command_execution_status	**info::event::command_status**

get_profiling_info() で照会

記述子	戻り値
info::event_profiling::command_submit	**uint64_t**
info::event_profiling::command_start	**uint64_t**
info::event_profiling::command_end	**uint64_t**

ホスト割り当て [4.7.1]

メモリー・オブジェクトのデフォルト・アロケーターは実装で定義されますが、ユーザーが独自のアロケーター・クラスを用意することができます。以下に例を示します。

```
buffer<int, 1, UserDefinedAllocator<int> > b(d);
```

デフォルト・アロケーターは、バッファーでは buffer_allocator、イメージでは image_allocator です。

バッファークラス [4.7.2]

バッファークラスは、カーネルによって使用されるアクセサークラスを介してアクセスする必要がある 1 次元、2 次元、または 3 次元の共有配列を定義します。デストラクターはブロックしないことに留意してください。

クラス宣言

```
template <typename T, int dimensions =1,
  typename AllocatorT =
    buffer_allocator<std::remove_const_t<T>>>
  class buffer;
```

メンバー関数

```
buffer(const range<dimensions> &bufferRange,
  const property_list &propList = {});
buffer(const range<dimensions> &bufferRange,
  AllocatorT allocator, const property_list &propList = {});
buffer(T *hostData, const range<dimensions> &bufferRange,
  const property_list &propList = {});
buffer(T *hostData, const range<dimensions> &bufferRange,
  AllocatorT allocator, const property_list &propList = {});

buffer(const T *hostData,
  const range<dimensions> &bufferRange,
  const property_list &propList = {});

buffer(const T *hostData,
  const range<dimensions> &bufferRange,
  AllocatorT allocator, const property_list &propList = {});
```

dimensions == 1 で std::data(container) が T* に変換可能な場合に利用可能

```
template <typename Container>
  buffer(Container &container, AllocatorT allocator,
    const property_list &propList = {});

template <typename Container>
  buffer(Container &container,
    const property_list &propList = {});
```

```
buffer(const std::shared_ptr<T> &hostData,
  const range<dimensions> &bufferRange,
  AllocatorT allocator, const property_list &propList = {});

buffer(const std::shared_ptr<T> &hostData,
  const range<dimensions> &bufferRange,
  const property_list &propList = {});

buffer(const std::shared_ptr<T[ ]> &hostData,
  const range<dimensions> &bufferRange,
  AllocatorT allocator, const property_list &propList = {});

buffer(const std::shared_ptr<T[ ]> &hostData,
  const range<dimensions> &bufferRange,
  const property_list &propList = {});

template <class InputIterator>
  buffer<T, 1>(InputIterator first, InputIterator last,
    AllocatorT allocator, const property_list &propList = {});

template <class InputIterator>
  buffer<T, 1>(InputIterator first, InputIterator last,
    const property_list &propList = {});

buffer(buffer &b,const id<dimensions> &baseIndex,
  const range<dimensions> &subRange);

get_range()
byte_size()
size_t size() const noexcept;

AllocatorT get_allocator() const;

template <access_mode mode = access_mode::read_write,
      target targ = target::device> accessor<T,
      dimensions, mode, targ>
  get_access(handler &commandGroupHandler);
template <access_mode mode = access_mode::read_write,
      target targ = target::device> accessor<T,
      dimensions, mode, targ>
  get_access(
    handler &commandGroupHandler, range<dimensions>
    accessRange, id<dimensions> accessOffset = {});

template<typename...Ts> auto get_access(Ts...);
template<typename...Ts> auto get_host_access(Ts...);
template <typename Destination = std::nullptr_t>
  void set_final_data(Destination finalData = nullptr);
void set_write_back(bool flag = true);
bool is_sub_buffer() const;

template <typename ReinterpretT, int ReinterpretDim>
  buffer<ReinterpretT, ReinterpretDim,
    typename std::allocator_traits<AllocatorT>::template
    rebind_alloc<ReinterpretT>>
  reinterpret(range<ReinterpretDim> reinterpretRange)
    const;
```

ReinterpretDim == 1 または (ReinterpretDim == dimensions) && (sizeof(ReinterpretT) == sizeof(T)) の場合に利用可能

```
template <typename ReinterpretT,
      int ReinterpretDim = dimensions>
  buffer<ReinterpretT, ReinterpretDim,
    typename std::allocator_traits<
    AllocatorT>::template rebind_alloc<ReinterpretT>>
  reinterpret() const;
```

バッファー・プロパティーのクラス・コンストラクター

```
property::buffer::use_host_ptr::use_host_ptr()

property::buffer::use_mutex::use_mutex(std::mutex
&mutexRef)

property::buffer::context_bound::context_bound(context
boundContext)
```

©2021 Khronos Group - Rev. 0221

非サンプルイメージとサンプルイメージ [4.7.3]

バッファーとイメージは、ストレージと所有権を定義します。
イメージのタイプは unsampled_image または sampled_image です。コンストラクターは、列挙型クラス image_format から image_format パラメーターを取得します。

列挙型クラス image_format の値

r8g8b8a8_unorm	r16g16b16a16_uint
r16g16b16a16_unorm	r32b32g32a32_uint
r8g8b8a8_sint	r16b16g16a16_sfloat
r16g16b16a16_sint	r32g32b32a32_sfloat
r32b32g32a32_sint	b8g8r8a8_unorm
r8g8b8a8_uint	

非サンプルイメージ [4.7.3.1]

クラス宣言

```
template <int dimensions = 1,
      typename AllocatorT = sycl::image_allocator>
      class unsampled_image;
```

コンストラクターとメンバー

```
unsampled_image(image_format format,
   const range<dimensions> &rangeRef,
   const property_list &propList = {});
unsampled_image(image_format format,
   const range<dimensions> &rangeRef, AllocatorT allocator,
   const property_list &propList = {});
unsampled_image(void *hostPointer, image_format format,
   const range<dimensions> &rangeRef,
   const property_list &propList = {});
unsampled_image(void *hostPointer,
   image_format format, const range<dimensions> &rangeRef,
   AllocatorT allocator, const property_list &propList = {});
unsampled_image(std::shared_ptr<void> &hostPointer,
   image_format format, const range<dimensions> &rangeRef,
   const property_list &propList = {});
unsampled_image(std::shared_ptr<void> &hostPointer,
   image_format format, const range<dimensions> &rangeRef,
   AllocatorT allocator, const property_list &propList = {});
```

dimensions > 1 の場合に利用可能

```
unsampled_image(image_format format,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   const property_list &propList = {});
```

dimensions > 1 の場合に利用可能

```
unsampled_image(image_format format,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   AllocatorT allocator, const property_list &propList = {});

unsampled_image(void *hostPointer,
   image_format format,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   property_list &propList = {});

unsampled_image(void *hostPointer,
   image_format format,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   AllocatorT allocator, const property_list &propList = {});

unsampled_image(std::shared_ptr<void> &hostPointer,
   image_format format,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   AllocatorT allocator, const property_list &propList = {});

unsampled_image(std::shared_ptr<void> &hostPointer,
   image_format format,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   const property_list &propList = {});
```

```
range<dimensions> get_range() const;
```

dimensions > 1 の場合に利用可能

```
range<dimensions-1> get_pitch() const;
```

```
size_t size() const noexcept;
size_t byte_size() const noexcept;
AllocatorT get_allocator() const;

template<typename... Ts>
   auto get_access(Ts... args);
template<typename... Ts>
   auto get_host_access(Ts... args);
template <typename Destination = std::nullptr_t>
   void set_final_data(Destination finalData = std::nullptr);
void set_write_back(bool flag = true);
```

サンプルイメージ [4.7.3.2]

クラス宣言

```
template <int dimensions = 1,
   typename AllocatorT = sycl::image_allocator>
   class sampled_image;
```

コンストラクターとメンバー

```
sampled_image(const void *hostPointer,
   image_format format, image_sampler sampler,
   const range<dimensions> &rangeRef,
   const property_list &propList = {});

sampled_image(std::shared_ptr<const void> &hostPointer,
   image_format format, image_sampler sampler,
   const range<dimensions> &rangeRef,
   const property_list &propList = {});
```

dimensions > 1 の場合に利用可能

```
sampled_image(const void *hostPointer,
   image_format format, image_sampler sampler,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   const property_list &propList = {});
sampled_image(std::shared_ptr<const void> &hostPointer,
   image_format format, image_sampler sampler,
   const range<dimensions> &rangeRef,
   const range<dimensions - 1> &pitch,
   const property_list &propList = {});
```

```
range<dimensions> get_range() const;
range<dimensions-1> get_pitch() const;
size_t byte_size() const noexcept;
size_t size() const noexcept;
template<typename... Ts>
   auto get_access(Ts... args);
template<typename... Ts>
   auto get_host_access(Ts... args);
```

イメージ・プロパティーのコンストラクターとメンバー [4.7.3.3]

```
property::image::use_host_ptr::use_host_ptr();

property::image::use_mutex::
   use_mutex(std::mutex &mutexRef);

property::image::context_bound::
   context_bound(context boundContext);

std::mutex *property::image::use_mutex::
   get_mutex_ptr() const;

context property::image::context_bound::
   get_context() const;
```

データアクセスとストレージ [4.7]

バッファーとイメージはストレージと所有権を定義します。
アクセサーはデータへのアクセスを提供します。

アクセサー [4.7.6]

アクセサークラスとオブジェクトは次のオブジェクトにアクセスします。

- 次の 2 つの用途のバッファーアクセサー (4.7.6.9 クラスアクセサー):
  - デバイスのグローバルメモリーを介してカーネル関数からバッファーにアクセス
  - ホストタスクからバッファーにアクセス
- コマンド外のホストコード向けのバッファーアクセサー (4.7.6.10 クラス host_accessor)。
- カーネル関数内からのローカルアクセサー (4.7.6.11 クラス local_accessor)。
- 2 種類の非サンプル・イメージ・アクセサー:
  - カーネル関数内から、またはホストタスク内から (4.7.6.13 クラス unsampled_image_accessor)。
  - ホストタスク外のホストコードから (4.7.6.13 クラス host_unsampled_image_accessor)。
- 2 種類のサンプル・イメージ・アクセサー:
  - カーネル関数内から、またはホストタスク内から (4.7.6.14 クラス sampled_image_accessor)。
  - ホストタスク外のホストコードから (4.7.6.14 クラス host_sampled_image_accessor)。

列挙型クラス access_mode [4.7.6.2]

read	write	Read_write

アクセサー・プロパティーのクラス・コンストラクター [4.7.6.4]

これはすべてのアクセサークラスで使用されます。

```
property::no_init::no_init()
```

アクセスターゲット [4.7.6.9]

target::device	デバイスのグローバルメモリーを介してカーネル関数からバッファーにアクセス
target::host_task	ホストタスクからバッファーにアクセス

列挙型クラス access::address_space [4.7.7.1]

global_space	すべてのワークグループのすべてのワーク項目にアクセス可能
constant_space	定数のグローバル空間
local_space	単一ワークグループのすべてのワーク項目にアクセス可能
private_space	単一のワーク項目にアクセス可能
generic_space	グローバル、ローカル、プライベートが重複する仮想アドレス空間

コマンドのバッファーアクセサー (クラスアクセサー) [4.7.6.9]

このクラスは、accessTarget に応じて 2 種類のアクセサーを提供します。

- デバイスのグローバルメモリーを介してカーネル関数からバッファーにアクセスする target::device
- ホストタスクからバッファーにアクセスする target::device

クラス宣言

```
template <typename dataT, int dimensions,
      access_mode accessMode =
         (std::is_const_v<dataT> ? access_mode::read
                                 : access_mode::read_write),
      target accessTarget = target::device,
class accessor;
```

コンストラクターとメンバー

```
accessor();
```

dimensions == 0 の場合に利用可能

```
template <typename AllocatorT>
   accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
   const property_list &propList = {});

template <typename AllocatorT>
   accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
   handler &commandGroupHandlerRef,
   const property_list &propList = {});
```

dimensions > 0 の場合に利用可能

```
template <typename AllocatorT>
   accessor(buffer<dataT, dimensions,
      AllocatorT> &bufferRef,
      const property_list &propList = {});

template <typename AllocatorT, typename TagT>
   accessor(buffer<dataT, dimensions,
      AllocatorT> &bufferRef, TagT tag,
      const property_list &propList = {});
```

dimensions > 0 の場合に利用可能

```
template <typename AllocatorT>
   accessor(buffer<dataT, dimensions, AllocatorT>
   &bufferRef,
   handler &commandGroupHandlerRef,
   const property_list &propList = {});

template <typename AllocatorT, typename TagT>
   accessor(buffer<dataT, dimensions, AllocatorT>
      &bufferRef, handler &commandGroupHandlerRef,
      TagT tag, const property_list &propList = {});

template <typename AllocatorT>
   accessor(buffer<dataT, dimensions, AllocatorT>
      &bufferRef, range<dimensions> accessRange,
      const property_list &propList = {});

template <typename AllocatorT, typename TagT>
   accessor(buffer<dataT, dimensions, AllocatorT>
   &bufferRef, range<dimensions> accessRange,
   TagT tag, const property_list &propList = {});
```

(続く) ▶

◀コマンドのバッファーアクセサー(続き)

dimensions > 0 の場合に利用可能

```
template <typename AllocatorT, typename TagT>
  accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
    range<dimensions> accessRange,
    id<dimensions> accessOffset, TagT tag,
    const property_list &propList = {});

template <typename AllocatorT>
  accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
    handler &commandGroupHandlerRef,
    range<dimensions> accessRange,
    const property_list &propList = {});

template <typename AllocatorT, typename TagT>
  accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
    handler &commandGroupHandlerRef,
    range<dimensions> accessRange, TagT tag,
    const property_list &propList = {});

template <typename AllocatorT>
  accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
    handler &commandGroupHandlerRef,
    range<dimensions> accessRange,
    id<dimensions> accessOffset,
    const property_list &propList = {});
```

dimensions > 0 の場合に利用可能

```
template <typename AllocatorT, typename TagT>
  accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
    handler &commandGroupHandlerRef,
    range<dimensions> accessRange,
    id<dimensions> accessOffset, TagT tag,
    const property_list &propList = {});
id<dimensions> get_offset() const;
```

```
void swap(accessor &other);
bool is_placeholder() const;
template <access::decorated IsDecorated>
  accessor_ptr<IsDecorated> get_multi_ptr() const noexcept;
```

共通インターフェイス関数 [表 79]

このクラスは、begin()、end()、cbegin()、cend()、rbegin()、rend()、crbegin()、および crend() に加えて次の関数をサポートします。

```
size_type byte_size() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
bool empty() const noexcept;
range<dimensions> get_range() const;
```

dimensions == 0 の場合に利用可能

```
operator reference() const;
```

dimensions > 0 の場合に利用可能

```
reference operator[](id<dimensions> index) const;
```

dimensions > 1 の場合に利用可能

```
__unspecified__ &operator[](size_t index) const;
```

dimensions == 1 の場合に利用可能

```
reference operator[](size_t index) const;
```

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

プロパティーのクラス・コンストラクター [4.7.3.3]

```
property::no_init::no_init()
```

コマンド外のホストコード向けのバッファーアクセサー (クラス host_accessor) [4.7.6.10]

クラス宣言

```
template <typename dataT, int dimensions,
      access_mode accessMode =
        (std::is_const_v<dataT> ? access_mode::read
                                : access_mode::read_write)>
class host_accessor;
```

コンストラクターとメンバー

すべてのコンストラクターは、同じバッファーにアクセスするカーネルからのデータが利用可能になるまでブロックします。

```
host_accessor();
```

dimensions == 0 の場合に利用可能

```
template <typename AllocatorT>
  host_accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
    const property_list &propList = {});
```

dimensions > 0 の場合に利用可能

```
template <typename AllocatorT>
  host_accessor(
    buffer<dataT, dimensions, AllocatorT> &bufferRef,
    const property_list &propList = {});

template <typename AllocatorT, typename TagT>
  host_accessor(
    buffer<dataT, dimensions, AllocatorT> &bufferRef,
    TagT tag, const property_list &propList = {});
```

dimensions > 0 の場合に利用可能

```
template <typename AllocatorT>
  host_accessor(
    buffer<dataT, dimensions, AllocatorT> &bufferRef,
    range<dimensions> accessRange,
    const property_list &propList = {});

template <typename AllocatorT, typename TagT>
  host_accessor(
    buffer<dataT, dimensions, AllocatorT> &bufferRef,
    range<dimensions> accessRange, TagT tag,
    const property_list &propList = {});

template <typename AllocatorT>
  host_accessor(
    buffer<dataT, dimensions, AllocatorT> &bufferRef,
    range<dimensions> accessRange,
    id<dimensions> accessOffset,
    const property_list &propList = {});

template <typename AllocatorT, typename TagT>
  host_accessor(
    buffer<dataT, dimensions, AllocatorT> &bufferRef,
    range<dimensions> accessRange,
    id<dimensions> accessOffset, TagT tag,
    const property_list &propList = {});
id<dimensions> get_offset() const;
```

```
void swap(host_accessor &other);
```

共通インターフェイス関数 [表 79]

このクラスは、begin()、end()、cbegin()、cend()、rbegin()、rend()、crbegin()、および crend() に加えて次の関数をサポートします。

```
size_t byte_size() const noexcept;
size_t size() const noexcept;
size_t max_size() const noexcept;
bool empty() const noexcept;
range<dimensions> get_range() const;
```

dimensions == 0 の場合に利用可能

```
operator reference() const;
```

dimensions > 0 の場合に利用可能

```
reference operator[](id<dimensions> index) const;
```

dimensions > 1 の場合に利用可能

```
__unspecified__ &operator[](size_t index) const;
```

dimensions == 1 の場合に利用可能

```
reference operator[](size_t index) const;
```

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

プロパティーのクラス・コンストラクター [4.7.3.3]

```
property::no_init::no_init()
```

カーネル関数内のローカルアクセサー (クラス local_accessor) [4.7.6.11]

dataT は任意の C++ タイプにできます。

クラス宣言

```
template <typename dataT, int dimensions>
  class local_accessor;
```

コンストラクターとメンバー

```
local_accessor();
```

dimensions == 0 の場合に利用可能

```
local_accessor(handler &commandGroupHandlerRef,
  const property_list &propList = {});
```

dimensions > 0 の場合に利用可能

```
local_accessor(range<dimensions> allocationSize,
  handler &commandGroupHandlerRef,
  const property_list &propList = {});
```

```
void swap(accessor &other);
```

```
template <access::decorated IsDecorated>
  accessor_ptr<IsDecorated>
  get_multi_ptr() const noexcept;
```

共通インターフェイス関数 [表 79]

このクラスは、begin()、end()、cbegin()、cend()、rbegin()、rend()、crbegin()、および crend() に加えて次の関数をサポートします。

```
size_t byte_size() const noexcept;
size_t size() const noexcept;
size_t max_size() const noexcept;
bool empty() const noexcept;
range<dimensions> get_range() const;
```

dimensions == 0 の場合に利用可能

```
operator reference() const;
```

dimensions > 0 の場合に利用可能

```
reference operator[](id<dimensions> index) const;
```

dimensions > 1 の場合に利用可能

```
__unspecified__ &operator[](size_t index) const;
```

dimensions == 1 の場合に利用可能

```
reference operator[](size_t index) const;
```

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

プロパティーのクラス・コンストラクター [4.7.3.3]

```
property::no_init::no_init()
```

©2021 Khronos Group - Rev. 0221

非サンプル・イメージ・アクセサー [4.7.6.13]

非サンプルイメージのアクセサーには次の 2 種類があります。

- クラス unsampled_image_accessor: カーネル関数内からまたはホストタスク内から
- クラス host_unsampled_image_accessor: ホストタスク外のホストコードから

unsampled_image_accessor クラス宣言

```
template <typename dataT, int dimensions,
      access_mode accessMode,
      image_target accessTarget = image_target::device >
   class unsampled_image_accessor;
```

コンストラクター

```
template <typename AllocatorT>
   unsampled_image_accessor(
   unsampled_image<dimensions, AllocatorT> &imageRef,
   handler &commandGroupHandlerRef,
   const property_list &propList = {});
```

```
template <typename AllocatorT, typename TagT>
   unsampled_image_accessor(
   unsampled_image<dimensions, AllocatorT> &imageRef,
   handler &commandGroupHandlerRef,
   TagT tag, const property_list &propList = {});
```

host_unsampled_image_accessor クラス宣言

```
template <typename dataT, int dimensions,
   access_mode accessMode>
   class host_unsampled_image_accessor;
```

コンストラクター

```
template <typename AllocatorT>
   host_unsampled_image_accessor(
   unsampled_image<dimensions, AllocatorT> &imageRef,
   const property_list &propList = {});
```

```
template <typename AllocatorT, typename TagT>
   host_unsampled_image_accessor(
   unsampled_image<dimensions, AllocatorT> &imageRef,
   TagT tag, const property_list &propList = {});
```

両方の非サンプルイメージのアクセサータイプで利用可能

```
size_t size() const noexcept;
```

(accessMode == access_mode::read) の場合に利用可能

```
template <typename coordT>
   dataT read(const coordT &coords) const;
```

(accessMode == access_mode::write) の場合に利用可能

```
template <typename coordT>
   void write(const coordT &coords,
   const dataT &color) const;
```

サンプルイメージのアクセサー [4.7.6.14]

サンプルイメージのアクセサーには次の 2 種類があります。

- クラス sampled_image_accessor: カーネル関数内からまたはホストタスク内から
- クラス host_sampled_image_accessor: ホストタスク外のホストコードから

sampled_image_accessor クラス宣言

```
template <typename dataT, int dimensions,
      image_target accessTarget = image_target::device>
      class sampled_image_accessor;
```

コンストラクター

```
template <typename AllocatorT>
      sampled_image_accessor(
      sampled_image<dimensions, AllocatorT> &imageRef,
      handler &commandGroupHandlerRef,
      const property_list &propList = {});
```

```
template <typename AllocatorT, typename TagT>
      sampled_image_accessor(
      sampled_image<dimensions, AllocatorT> &imageRef,
      handler &commandGroupHandlerRef, TagT tag,
      const property_list &propList = {});
```

host_sampled_image_accessor クラス宣言

```
template <typename dataT, int dimensions>
      class host_sampled_image_accessor;
```

コンストラクター

```
template <typename AllocatorT>
      host_sampled_image_accessor(
      sampled_image<dimensions, AllocatorT> &imageRef,
      const property_list &propList = {});
```

両方のサンプルイメージのアクセサータイプで利用可能

```
size_t size() const noexcept;
```

dimensions == 1 の場合は coordT = float
dimensions == 2 の場合は coordT = float2
dimensions == 4 の場合は coordT = float4

```
template <typename coordT>
      dataT read(const coordT &coords) const;
```

クラス multi_ptr [4.7.7.1]

アドレス空間は、global_space、local_space、private_space、および generic_space です。

クラス宣言

```
template <typename ElementType,
      access::address_space Space,
      access::decorated DecorateAddress>
   class multi_ptr;
```

メンバー: コンストラクター

```
multi_ptr();
multi_ptr(const multi_ptr&);
multi_ptr(multi_ptr&&);
explicit multi_ptr(multi_ptr<ElementType, Space,  yes>::pointer);
multi_ptr(std::nullptr_t);
```

Space == global_space または generic_space の場合に利用可能

```
template <int dimensions, access::mode Mode,
      access::placeholder isPlaceholder>
   multi_ptr(accessor<ElementType, dimensions, Mode,
      target::device, isPlaceholder>);
template <int dimensions>
   multi_ptr(local_accessor<ElementType, dimensions>);
```

メンバー: 割り当ておよびアクセス演算子

```
multi_ptr &operator=(const multi_ptr&);
multi_ptr &operator=(multi_ptr&&);
multi_ptr &operator=(std::nullptr_t);
```

Space == address_space::generic_space && ASP != access::address_space::constant_space の場合に利用可能

```
ASP != access::address_space::constant_space
template<access::address_space ASP,
      access::decorated IsDecorated>
   multi_ptr &operator=(
      const multi_ptr<value_type, ASP, IsDecorated>&);
template<access::address_space ASP,
      access::decorated IsDecorated>
   multi_ptr &operator=(
      multi_ptr<value_type, ASP, IsDecorated>&&);
```

```
reference operator*() const;
pointer operator->() const;
pointer get() const;
std::add_pointer_t<value_type> get_raw() const;
__unspecified__ * get_decorated() const;
```

メンバー: 変換

Space == address_space::generic_space の場合に private_ptr へキャスト

```
explicit operator multi_ptr<value_type,
   access::address_space::private_space,
   DecorateAddress>();
explicit operator multi_ptr<const value_type,
   access::address_space::private_space,
   DecorateAddress>() const;
```

Space == address_space::generic_space の場合に global_ptr へキャスト

```
explicit operator multi_ptr<value_type,
   access::address_space::global_space, DecorateAddress>();
explicit operator multi_ptr<const value_type,
   access::address_space::global_space,
   DecorateAddress>() const;
explicit operator multi_ptr<value_type,
   access::address_space::local_space,
   DecorateAddress>();
```

Space == global_space の場合に利用可能

```
template <typename ElementType, int dimensions,
   access_mode Mode, access::placeholder isPlaceholder>
multi_ptr(accessor<ElementType, dimensions, Mode,
   target::device, isPlaceholder>);
```

Space == local_space の場合に利用可能

```
template <typename ElementType, int dimensions>
   multi_ptr(local_accessor<ElementType, dimensions>);
```

Space == address_space::generic_space の場合に利用可能

```
explicit operator multi_ptr<value_type,
   access::address_space::local_space,
   DecorateAddress>() const;
```

multi_ptr への暗黙の変換

```
multi_ptr<void> への暗黙の変換。
value_type が const 修飾されていない場合にのみ利用可能。
template<access::decorated DecorateAddress>
   operator multi_ptr<void, Space, DecorateAddress>() const;

multi_ptr<const void> への暗黙の変換。
value_type が const 修飾されている場合にのみ利用可能。
template<access::decorated DecorateAddress>
   operator multi_ptr<const void, Space,
   DecorateAddress>() const;

multi_ptr<const value_type, Space> への暗黙の変換。
template<access::decorated DecorateAddress>
   operator multi_ptr<const value_type, Space,
   DecorateAddress>() const;

修飾されていない multi_ptr バージョンへの暗黙の変換。
is_decorated が true の場合にのみ利用可能。
operator multi_ptr<value_type, Space,
   access::decorated::no>() const;

修飾された multi_ptr バージョンへの暗黙の変換。
is_decorated が false の場合にのみ利用可能。.
operator multi_ptr<value_type, Space,
   access::decorated::yes>() const;
```

```
void prefetch(size_t numElements) const;
```

メンバー: 算術演算子

multi_ptr クラスは、標準の算術演算子と関係演算子をサポートします。

代入操作

```
multi_ptr &operator=(const multi_ptr&);
multi_ptr &operator=(multi_ptr&&);
multi_ptr &operator=(std::nullptr_t);
```

メンバー

```
pointer get() const;
explicit operator pointer() const;
template <typename ElementType>
   explicit operator multi_ptr<ElementType, Space,
   DecorateAddress>() const;
```

is_decorated が true の場合にのみ利用可能

```
operator multi_ptr<value_type, Space,
   access::decorated::no>() const;
```

(続く) ▶

void および const void に特殊化したクラス multi_ptr [4.7.7.1]

クラス宣言

```
template <access::address_space Space,
      access::decorated DecorateAddress>
   class multi_ptr<VoidType, Space, DecorateAddress>
```

DecorateAddress: yes、no
VoidType: void または const void

メンバー: コンストラクター

```
multi_ptr();
multi_ptr(const multi_ptr&);
multi_ptr(multi_ptr&&);
explicit multi_ptr(multi_ptr<VoidType, Space, yes>::pointer);
multi_ptr(std::nullptr_t);
```

©2021 Khronos Group - Rev. 0221

◀ multi_ptr 特殊化 (続き)

is_decorated が false の場合にのみ利用可能

operator multi_ptr<value_type, Space, access::decorated::yes>() const;

operator multi_ptr<const void, Space, DecorateAddress>() const;

template <access::address_space Space, access::decorated DecorateAddress, typename ElementType>
multi_ptr<ElementType, Space, DecorateAddress> **address_space_cast**(ElementType *);

演算子

multi_ptr クラスは、標準の算術演算子と関係演算子をサポートします。

明示的なポインターエイリアス [4.7.7.2]

access::address_space の特殊化ごとのクラス multi_ptr へのエイリアス。

global_ptr
local_ptr
private_ptr

修飾されていないポインターのエイリアス。

raw_global_ptr
raw_local_ptr
raw_private_ptr

修飾されたポインターのエイリアス。

decorated_global_ptr
decorated_local_ptr
decorated_private_ptr

サンプラークラス列挙型 [4.7.8]

SYCL image_sampler 構造体には、sampled_image をサンプルする設定が含まれています。

```
struct image_sampler {
    addressing_mode addressing;
    coordinate_mode coordinate;
    filtering_mode filtering;
}
```

アドレス指定
mirrored_repeat
repeat
clamp_to_edge
clamp
none

フィルター処理
nearest
linear

座標
normalized
unnormalized

統合共有メモリー [4.8]

統合共有メモリーは、バッファーモデルに代わってオプションのアドレス指定モデルを提供します。15 ページ目の例をご覧ください。

3 種類の USM 割り当てが利用できます (列挙型クラス alloc)。

ホスト	デバイスからアクセス可能なホストメモリー
デバイス	ホストからアクセスできないデバイスメモリー
共有	ホストとデバイスからアクセス可能な共有メモリー

クラス usm_allocator [4.8.3]
クラス宣言

usm_allocator(const context &*ctxt*, const device &*dev*, const property_list &*propList* = {}) noexcept;
usm_allocator(const queue &*q*, const property_list &*propList* = {}) noexcept;
template <class U> **usm_allocator**(usm_allocator<U, AllocKind, Alignment> const &) noexcept;
T ***allocate**(size_t *count*);
void **deallocate**(T **Ptr*, size_t *count*);

アロケーターは、同じ種類の USM、アライメント、コンテキスト、およびデバイスである場合にのみ同等であると判断

template <class U, usm::alloc AllocKindU, size_t AlignmentU>
friend bool **operator==**(const usm_allocator<T, AllocKind, Alignment> &, const usm_allocator<U, AllocKindU, AlignmentU> &);

template <class U, usm::alloc AllocKindU, size_t AlignmentU>
friend bool **operator!=**(const usm_allocator<T, AllocKind, Alignment> &, const usm_allocator<U, AllocKindU, AlignmentU> &);

malloc 形式 API [4.8.3]
デバイス割り当て関数 [4.8.3.2]

void* sycl::**malloc_device**(size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc_device**(size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**malloc_device**(size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc_device**(size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

ホスト割り当て関数 [4.8.3.3]

void* sycl::**malloc_host**(size_t *numBytes*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc_host**(size_t *count*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**malloc_host**(size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc_host**(size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *numBytes*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *count*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

共有割り当て関数 [4.8.3.4]

void* sycl::**malloc_shared**(size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc_shared**(size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**malloc_shared**(size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc_shared**(size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

void* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

パラメーター化された割り当て関数 [4.8.3.5]

void* sycl::**malloc**(size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc**(size_t *count*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

void* sycl::**malloc**(size_t *numBytes*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T>
T* sycl::**malloc**(size_t *count*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

void* sycl::**aligned_alloc**(size_t *alignment*, size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T>
T* sycl::**aligned_alloc**(size_t *alignment*, size_t *count*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

void* sycl::**aligned_alloc**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

emplate <typename T>
T* sycl::**aligned_alloc**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

メモリー解放関数 [4.8.3.6]

void sycl::**free**(void* *ptr*, sycl::context& *syclContext*);
void sycl::**free**(void* *ptr*, sycl::queue& *syclQueue*);

USM ポインター照会 [4.8.4]

これらの照会はホストでのみ利用できます。

usm::alloc **get_pointer_type**(const void *ptr, const context &*ctxt*)
sycl::device **get_pointer_device**(const void *ptr, const context &*ctxt*)

©2021 Khronos Group - Rev. 0221

レンジとインデックス空間識別子 [4.9.1]

クラス range [4.9.1.1]

並列ディスパッチ内の単一ワークグループの反復ドメイン、またはディスパッチ全体の次元を定義する 1D、2D または 3D ベクトル。これは整数で構成できます。このクラスは、標準の算術演算子、論理演算子、および関係演算子をサポートします。

クラス宣言

```
template <int dimensions = 1> class range;
```

コンストラクターとメンバー

```
range(size_t dim0);
range(size_t dim0, size_t dim1);
range(size_t dim0, size_t dim1, size_t dim2);
size_t get(int dimension) const;
size_t &operator[](int dimension);
size_t operator[](int dimension) const;
size_t size() const;
```

クラス nd_range [4.9.1.2]

ワークグループとディスパッチ全体の両方の反復ドメインを定義します。そのため、nd_range はカーネルが実行される範囲全体と、それぞれのワークグループの範囲の 2 つで構成されます。

クラス宣言

```
template <int dimensions = 1> class nd_range;
```

コンストラクターとメンバー

```
nd_range(range<dimensions> globalSize,
range<dimensions> localSize);
range<dimensions> get_global_range() const;
range<dimensions> get_local_range() const;
range<dimensions> get_group_range() const;
```

クラス id [4.9.1.3]

グローバル範囲またはローカル範囲の ID を示す次元のベクトル。同じランクアクセサーのインデックスとして使用できます。このクラスは、標準の算術演算子、論理演算子、および関係演算子をサポートします。

クラス宣言

```
template <int dimensions = 1> class id;
```

コンストラクターとメンバー

```
id();
id(size_t dim0);
id(size_t dim0, size_t dim1);
id(size_t dim0, size_t dim1, size_t dim2);
id(const range<dimensions> &range);
id(const item<dimensions> &item);
size_t get(int dimension) const;
size_t &operator[](int dimension);
size_t operator[](int dimension) const;
```

クラス item [4.9.1.4]

範囲内のそれぞれのポイントで実行される関数オブジェクトのインスタンスを識別します。これは、**parallel_for** 呼び出しに渡されるか、h_item のメンバー関数によって返されます。

クラス宣言

```
template <int dimensions = 1, bool with_offset = true>
  class item;
```

メンバー

```
id<dimensions> get_id() const;
size_t get_id(int dimension) const;
size_t operator[](int dimension) const;
range<dimensions> get_range() const;
size_t get_range(int dimension) const;
```

with_offset が false の場合に利用可能

```
operator item<dimensions, true>() const;
```

dimensions == 1 の場合に利用可能

```
operator size_t() const;
```

```
size_t get_linear_id() const;
```

クラス nd_item [4.9.1.5]

parallel_for 呼び出しに渡される nd_range<int dimensions> のポインターで実行される関数オブジェクトのインスタンスを識別します。

クラス宣言

```
template <int dimensions = 1> class nd_item;
```

メンバー

```
id<dimensions> get_global_id() const;
size_t get_global_id(int dimension) const;
size_t get_global_linear_id() const;
id<dimensions> get_local_id() const;
size_t get_local_id(int dimension) const;
size_t get_local_linear_id() const;
group<dimensions> get_group() const;
size_t get_group(int dimension) const;
size_t get_group_linear_id() const;
range<dimensions> get_group_range() const;
size_t get_group_range(int dimension) const;
range<dimensions> get_global_range() const;
size_t get_global_range(int dimension) const;
range<dimensions> get_local_range() const;
size_t get_local_range(int dimension) const;
nd_range<dimensions> get_nd_range() const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src,
    size_t numElements) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src,
    size_t numElements) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src,
    size_t numElements, size_t srcStride) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src,
    size_t numElements, size_t destStride) const;

template <typename... eventTN>
  void wait_for(eventTN... events) const;
```

クラス h_item [4.9.1.6]

parallel_for_work_item 呼び出しに渡されるローカル range<int dimensions>の各ポイントで実行される、または **parallel_for_work_item** 呼び出しに範囲が渡されない場合は対応する **parallel_for_work_group** 呼び出しで実行される group::**parallel_for_work_item** 関数のインスタンスを識別します。

クラス宣言

```
template <int dimensions> class h_item;
```

メンバー

```
item<dimensions, false> get_global() const;
item<dimensions, false> get_local() const;
item<dimensions, false> get_logical_local() const;
item<dimensions, false> get_physical_local() const;
range<dimensions> get_global_range() const;
size_t get_global_range(int dimension) const;
id<dimensions> get_global_id() const;
size_t get_global_id(int dimension) const;
range<dimensions> get_local_range() const;
size_t get_local_range(int dimension) const;
id<dimensions> get_local_id() const;
size_t get_local_id(int dimension) const;
range<dimensions> get_logical_local_range() const;
size_t get_logical_local_range(int dimension) const;
id<dimensions> get_logical_local_id() const;
size_t get_logical_local_id(int dimension) const;
range<dimensions> get_physical_local_range() const;
size_t get_physical_local_range(int dimension) const;
id<dimensions> get_physical_local_id() const;
size_t get_physical_local_id(int dimension) const;
```

クラス group [4.9.1.7]

並列実行内で特定のワークグループを表現するため必要なすべての機能をカプセル化します。ユーザーが作成することはできません。

クラス宣言

```
template <int dimensions = 1> class group;
```

メンバー

```
id<Dimensions> get_group_id() const;
size_t get_group_id(int dimension) const;
id<Dimensions> get_local_id() const;
size_t get_local_id(int dimension) const;
range<Dimensions> get_local_range() const;
size_t get_local_range(int dimension) const;
range<Dimensions> get_group_range() const;
size_t get_group_range(int dimension) const;
range<Dimensions> get_max_local_range() const;
size_t operator[](int dimension) const;
size_t get_group_linear_id() const;
size_t get_local_linear_id() const;
size_t get_group_linear_range() const;
size_t get_local_linear_range() const;
bool leader() const;

template<typename workItemFunctionT>
  void parallel_for_work_item(
    const workItemFunctionT &func) const;

template<typename workItemFunctionT>
  void parallel_for_work_item(range<dimensions>
    logicalRange, const workItemFunctionT &func) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src,
    size_t numElements) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src,
    size_t numElements) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src,
    size_t numElements, size_t srcStride) const;

template <typename dataT>
  device_event async_work_group_copy(
    decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src, size_t numElements,
    size_t destStride) const;

template <typename... eventTN>
  void wait_for(eventTN... events) const;
```

クラス sub_group [4.9.1.8]

並列実行内で特定のサブグループを表現するため必要なすべての機能をカプセル化します。ユーザーが作成することはできません。

メンバー

```
id<1> get_group_id() const;
id<1> get_local_id() const;
range<1> get_local_range() const;
range<1> get_group_range() const;
range<1> get_max_local_range() const;
uint32_t get_group_linear_id() const;
uint32_t get_local_linear_id() const;
uint32_t get_group_linear_range() const;
uint32_t get_local_linear_range() const;
bool leader() const;
```

リダクション変数 [4.9.2]

リダクションはすべての SYCL コピー可能タイプでサポートされます。

```
template <typename BufferT, typename BinaryOperation>
  __unspecified__ reduction(BufferT vars, handler& cgh,
    BinaryOperation combiner,
    const property_list &propList = {});

template <typename T, typename BinaryOperation>
  __unspecified__ reduction(T* var,
    BinaryOperation combiner,
    const property_list &propList ={});

template <typename T, typename Extent,
      typename BinaryOperation>
  __unspecified__ reduction(span<T, Extent> vars,
    BinaryOperation combiner,
    const property_list &propList = {});
```

has_known_identity<BinaryOperation, BufferT::value_type>::value が false の場合に利用可能

```
template <typename BufferT, typename BinaryOperation>
  __unspecified__ reduction(BufferT vars, handler& cgh,
    const BufferT::value_type& identity,
    BinaryOperation combiner,
    const property_list &propList = {});
```

has_known_identity<BinaryOperation, T>::value が false の場合に利用可能

```
template <typename T, typename BinaryOperation>
  __unspecified__ reduction(T* var, const T& identity,
    BinaryOperation combiner,
    const property_list &propList = {});
```

has_known_identity<BinaryOperation, T>::value が false の場合に利用可能

```
template <typename T, typename Extent,
      typename BinaryOperation>
  __unspecified__ reduction(span<T, Extent> vars,
    const T& identity, BinaryOperation combiner);
```

コマンドグループのハンドラークラス [4.9.4]

クラスハンドラー

コマンドグループのハンドラー・オブジェクトは、SYCL ランタイムでのみ構築できます。コマンドグループのスコープで定義されたすべてのアクセサーは、パラメーターとしてコマンドグループのハンドラー・インスタンスを受け取り、すべてのカーネル呼び出し関数はこのクラスのメンバー関数となります。

```
template <typename dataT, int dimensions,
      access_mode accessMode, access_target
      accessTarget, access::placeholder isPlaceholder>
  void require(accessor<dataT, dimensions, accessMode,
      accessTarget, placeholder> acc);

void depends_on(event depEvent);
void depends_on(const std::vector<event> &depEvents);
```

バックエンドの相互インターフェイス

```
template <typename T>
  void set_arg(int argIndex, T && arg);

template <typename... Ts>
  void set_args(Ts &&... args);
```

カーネル・ディスパッチ API

```
template <typename KernelName, typename KernelType>
  void single_task(const KernelType &kernelFunc);
template <typename KernelName, int dimensions,
      typename... Rest>
  void parallel_for(range<dimensions> numWorkItems,
      Rest&&... rest);
template <typename KernelName, int dimensions,
      typename... Rest>
  void parallel_for(nd_range<dimensions>
      executionRange,Rest&&... rest);
template <typename KernelName, typename
      WorkgroupFunctionType, int dimensions>
  void parallel_for_work_group(
      range<dimensions> numWorkGroups,
      const WorkgroupFunctionType &kernelFunc);
template <typename KernelName, typename
      WorkgroupFunctionType, int dimensions>
  void parallel_for_work_group(
      range<dimensions> numWorkGroups,
      range<dimensions> workGroupSize,
      const WorkgroupFunctionType &kernelFunc);
void single_task(const kernel_name &kernelObject);
template <int dimensions>
  void parallel_for(range<dimensions> numWorkItems,
      const kernel &kernelObject);
template <int dimensions>
  void parallel_for(range<dimensions> ndRange,
      const kernel &kernelObject);
```

リダクション・プロパティーのコンストラクター [4.9.2.2]

property::reduction::initialize_to_identity::initialize_to_identity()

レデューサー・クラス関数 [4.9.2.3]

SYCL カーネルの実行中にワーク項目とリダクション変数間のインターフェイスを定義し、リダクション変数へのアクセスを制限します。

```
template <typename T>
  void operator+=(reducer<T,plus<T>,0>& accum,
    const T& partial);
template <typename T>
  void operator*=(reducer<T,multiplies<T>,0>& accum,
    const T& partial);
```

整数タイプでのみ利用可能

```
template <typename T>
  void operator&=(reducer<T,bit_and<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator|=(reducer<T,bit_or<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator^=(reducer<T,bit_xor<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator++(reducer<T,plus<T>,0>& accum);
```

メンバー関数

```
void id combine(const T& partial) const;
__unspecified__ &operator[](size_t index) const;
T identity() const;
```

演算子

```
template <typename T>
  void operator+=(reducer<T, plus<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator*=(reducer<T, multiplies<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator|=(reducer<T, bit_or<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator&=(reducer<T, bit_and<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator^=(reducer<T, bit_xor<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator++(reducer<T, plus<T>,0>& accum);
```

リダクション・カーネルの例 [4.9.2]

次の例は、同一の入力値に対して 2 つのリダクションを同時に行い、バッファー内のすべての値の合計と最大値を計算するリダクション・カーネルの作成方法を示しています。

```
buffer<int> valuesBuf { 1024 };
{
  // 0, 1, 2, 3, ..., 1023 でホスト上のバッファーを初期化
  host_accessor a { valuesBuf };
  std::iota(a.begin(), a.end(), 0);
}

// リダクションの結果を取得する単一要素のバッファー
int sumResult = 0;
buffer<int> sumBuf { &sumResult, 1 };
int maxResult = 0;
buffer<int> maxBuf { &maxResult, 1 };

myQueue.submit([&](handler& cgh) {

  // リダクションの入力値は標準アクセサー
  auto inputValues =
            valuesBuf.get_access<access_mode::read>(cgh);

  // リダクションのセマンティクスで変数を記述する
  // 一時的なオブジェクトを作成
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());

  // parallel_for は 2 つのリダクション操作を実行
  // リダクション変数ごとの実装は以下の通り:
  // - 対応するレデューサーを作成
  // - レデューサー参照をパラメーターとしてラムダ関数に渡す
  cgh.parallel_for(range<1>{1024},
    sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
      // plus<>() は += 演算子に対応するため、
      // sum は += または combine() を介して更新
      sum += inputValues[idx];

      // maximum<>() には省略形の演算子がないため、
      // max は combine() を介してのみ更新できる
      max.combine(inputValues[idx]);
  });
});

// sumBuf と maxBuf には、カーネルが終了すると
// リダクションの結果が含まれる
assert(maxBuf.get_host_access()[0] == 1023
  && sumBuf.get_host_access()[0] == 523776);
```

USM 関数

```
void memcpy(void *dest, const void *src, size_t numBytes);
template <typename T>
  void copy(T *dest, const T *src, size_t count);
void memset(void *ptr, int value, size_t numBytes);
template <typename T>
  void fill(void *ptr, const T &pattern, size_t count);
void prefetch(void *ptr, size_t numBytes);
void mem_advise(void *ptr, size_t numBytes, int advice);
```

明示的なメモリー操作 API

カーネルに加えて、コマンド・グループ・オブジェクトはコマンド・グループ・ハンドラーのコピー API を使用して、ホストおよびデバイスのメモリーで手動で操作を行うために利用できます。以下にクラスハンドラーのメンバーを示します。

```
template <typename T_src, int dim_src,
      access_mode mode_src, target tgt_src,
      access::placeholder isPlaceholder,
      typename T_dest>
  void copy(accessor<T_src, dim_src,
      mode_src, tgt_src, isPlaceholder> src,
      std::shared_ptr<T_dest> dest);

template <typename T_src, typename T_dest, int dim_dest,
      access_mode mode_dest, target tgt_dest,
      access::placeholder isPlaceholder>
  void copy(std::shared_ptr<T_src> src,
      accessor<T_dest, dim_dest, mode_dest,
      tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src,
      access_mode mode_src, target tgt_src,
      access::placeholder isPlaceholder, typename T_dest>
  void copy(accessor<T_src, dim_src, mode_src, tgt_src,
      isPlaceholder> src, T_dest *dest);

template <typename T_src, typename T_dest,
      int dim_dest, access_mode mode_dest,
      target tgt_dest, access::placeholder isPlaceholder>
  void copy(const T_src *src, accessor<T_dest, dim_dest,
      mode_dest, tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src,
      access_mode mode_src, target tgt_src,
      access::placeholder isPlaceholder_src, typename T_dest,
      int dim_dest, access::mode mode_dest,
      access::target tgt_dest,
      access::placeholder isPlaceholder_dest>
  void copy(
      accessor<T_src, dim_src, mode_src, tgt_src,
      isPlaceholder_src> src,
      accessor<T_dest, dim_dest, mode_dest, tgt_dest,
      isPlaceholder_dest> dest);

template <typename T, int dim, access_mode mode, target tgt,
      access::placeholder isPlaceholder>
  void update_host(
      accessor<T, dim, mode, tgt, isPlaceholder> acc);

template<typename T, int dim, access_mode mode, target tgt,
      access::placeholder isPlaceholder>
  void fill(accessor<T, dim, mode, tgt, isPlaceholder> dest,
      const T& src);

template<auto& S>
  typename std::remove_reference_t<decltype(S)>::type
  get_specialization_constant();
```

カーネルバンドルを使用するメンバー関数 [4.9.4.4]

```
void use_kernel_bundle(const kernel_bundle<
  bundle_state::executable> &execBundle);
```

特殊化定数 [4.9.5]

クラス specialization_id 宣言

```
template <typename T>
  class specialization_id;
```

クラス specialization_id コンストラクター

```
template<class... Args >
  explicit constexpr specialization_id(Args&&... args);
```

クラスハンドラーのメンバー

```
template<auto& SpecName>
  void set_specialization_constant(
    typename std::remove_reference_t<decltype(
    SpecName)>::type value);

template<auto& SpecName>
    typename std::remove_reference_t<decltype(
    SpecName)>::type get_specialization_constant();
```

クラス kernel_handler のメンバー

```
template<auto& SpecName>
    typename std::remove_reference_t<decltype(
    SpecName)>::type get_specialization_constant();
```

クラス private_memory [4.10.4.2.3]

ワーク項目ごとに使用するプライベート・メモリーを保証するため、private_memory クラスでデータをラップできます。

```
class private_memory {
  public:
    private_memory(const group<Dimensions> &);
    T &operator()(const h_item<Dimensions> &id);
};
```

exception & exception_list クラス [4.13.2]

クラス exception は std::exception から派生します。

クラス exception のメンバー

```
exception(std::error_code ec, const std::string& what_arg);
exception(std::error_code ec, const char * what_arg);
exception(std::error_code ec);
exception(int ev, const std::error_category& ecat,
  const std::string& what_arg);
exception(int ev, const std::error_category& ecat,
  const char* what_arg);
exception(int ev, const std::error_category& ecat);
exception(context ctx, std::error_code ec,
  const std::string& what_arg);
exception(context ctx, std::error_code ec,
  const char* what_arg);
exception(context ctx, std::error_code ec);
exception(context ctx, int ev, const std::error_category& ecat,
  const std::string& what_arg);
exception(context ctx, int ev, const std::error_category& ecat,
  const char* what_arg);
exception(context ctx, int ev,
  const std::error_category& ecat);
const std::error_code& code() const noexcept;
const std::error_category& category() const noexcept;
bool has_context() const noexcept;
context get_context() const;
```

クラス exception_list のメンバー

```
size_type size() const;
iterator begin() const;
iterator end() const;
```

ヘルパー関数

```
フリー関数:
const std::error_category& sycl_category() noexcept;
template<backend b>
  const std::error_category& error_category_for() noexcept;
std::error_condition make_error_condition(errc e) noexcept;
std::error_code make_error_code(errc e) noexcept;
```

標準エラーコード (列挙型 errc)

runtime	invalid
kernel	memory_allocation
accessor	platform
nd_range	profiling
event	feature_not_supported
kernel_argument	kernel_not_supported
build	backend_mismatch

ホストタスク [4.10]

クラス interop_handle [4.10.1-2]

ホストタスクと関連するデバイスおよびコンテキストを呼び出す際に使用されるキューを抽象化します。

メンバー関数

```
backend get_backend() const noexcept;
```

バッファーを取得するオプションの相互関数 **get_native** が利用可能であり、accTarget が target::device である場合にのみ使用可能

```
template <backend Backend, typename dataT,
      int dims, access_mode accMode, target accTarget,
      access::placeholder isPlaceholder>
  backend_return_t<Backend, buffer<dataT, dims>>
  get_native_mem(const accessor<dataT, dims, accMode,
    accTarget, isPlaceholder> &bufferAcc) const;
```

unsampled_image を取得するオプションの相互関数 **get_native** が利用可能な場合にのみ使用可能

```
template <backend Backend, typename dataT, int dims,
      access_mode accMode>
  backend_return_t<Backend, unsampled_image<dims>>
  get_native_mem(
    const unsampled_image_accessor<dataT, dims,
    accMode, image_target::device> &imageAcc) const;
```

キューを取得するオプションの相互関数 **get_native** が利用可能な場合にのみ使用可能

```
template <backend Backend, typename dataT, int dims>
  backend_return_t<Backend, sampled_image<dims>>
  get_native_mem(
    const sampled_image_accessor<dataT, dims,
  image_target::device> &imageAcc) const;
```

キューを取得するオプションの相互関数 **get_native** が利用可能な場合にのみ使用可能

```
template <backend Backend>
  backend_return_t<Backend, queue>
  get_native_queue() const;
```

キューを取得するオプションの相互関数 **get_native** が利用可能な場合にのみ使用可能

```
template <backend Backend>
  backend_return_t<Backend, queue>
  get_native_device() const;
```

キューを取得するオプションの相互関数 **get_native** が利用可能な場合にのみ使用可能

```
template <backend Backend>
  backend_return_t<Backend, queue>
  get_native_context() const;
```

クラスハンドラーへの追加 [4.10.3]

```
template <typename T>
  void host_task(T &&hostTaskCallable);
```

カーネルの定義 [4.12]

SYCL デバイスで実行される関数は、SYCL カーネル関数と呼ばれます。SYCL カーネル関数を含むカーネルは、デバイスで実行するためデバイスキューに送信されます。

SYCL カーネル関数の戻り型は void です。
カーネルを定義するには、名前付き関数オブジェクトとラムダ関数の 2 つの方法があります。

カーネルを名前付き関数オブジェクトとして定義 [4.12.1]

カーネルは、名前付きオブジェクト・タイプとして定義でき、通常の C++ 関数オブジェクトと同等の機能を提供します。
例:

```
class RandomFiller {
  public:
    RandomFiller(accessor<int> ptr) : ptr_ { ptr } {
      std::random_device hwRand;
      std::uniform_int_distribution<> r { 1, 100 };
       randomNum_ = r(hwRand);
    }
    void operator()(item<1> item) const {
      ptr_[item.get_id()] = get_random(); }
    int get_random() { return randomNum_; }

  private:
    accessor<int> ptr_;
    int randomNum_;
};

void workFunction(buffer<int, 1>& b, queue& q,
      const range<1> r) {
  myQueue.submit([&](handler& cgh) {
    accessor ptr { buf, cgh };
    RandomFiller filler { ptr };

    cgh.parallel_for(r, filler);
  });
}
```

カーネルをラムダ関数として定義 [4.12.2]

カーネルはラムダ関数として定義できます。SYCL のラムダ関数名は、呼び出し元のメンバー関数でテンプレート・パラメーターとして指定できます (オプション)。
例:

```
// 明示的なカーネル名はオプションで、名前空間スコープクラス
// MyKernel で宣言できます
class MyKernel;

myQueue.submit([&](handler& h) {

  // 以前に前方宣言されたタイプでカーネルに
  // 明示的に名前を付けます
  h.single_task<MyKernel>([=]{
    // [カーネルコード]
  });

  // 名前空間スコープでタイプを前方宣言せずにカーネルに
  // 明示的に名前を付けます。名前空間スコープで宣言されて
  // いない場合でも、名前空間スコープで前方宣言可能
  // でなければなりません
  h.single_task<class MyOtherKernel>([=]{
    // [カーネルコード]
  });
});
```

同期とアトミック [4.15]

列挙型

クラス memory_order				
relaxed	acquire	release	acq_rel	seq_cst

クラス memory_scope				
work_item	sub_group	work_group	device	system

atomic_fence [4.15.1]

```
フリー関数:
void atomic_fence(memory_order order,
  memory_scope scope);
```

クラス device_event [4.15.2]

device_event クラスは、SYCL カーネル関数内でのみ利用可能な単一の SYCL デバイスイベントをカプセル化し、SYCL カーネル関数内の非同期操作の完了を待機するのに使用できます。このクラスには、不特定のコンストラクターとほかのメンバーが 1 つあります。

```
void wait() noexcept;
```

クラス atomic_ref [4.15.3]
クラス宣言

```
template <typename T, memory_order DefaultOrder,
      memory_scope DefaultScope, access::address_
      space Space = access::address_space::generic_space>
  class atomic_ref;
```

コンストラクターとメンバー

```
explicit atomic_ref(T& ref);
atomic_ref(const atomic_ref&) noexcept;
bool is_lock_free() const noexcept;
void store(T operand,
  memory_order order = default_write_order,
  memory_scope scope = default_scope) const noexcept;
T operator=(T desired) const noexcept;
T load(memory_order order = default_read_order,
  memory_scope scope = default_scope) const noexcept;
```

(続く) ▶

©2021 Khronos Group - Rev. 0221

◀ 同期とアトミック (続き)

operator T() const noexcept;
T **exchange**(T *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_weak**(T &*expected*, T *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_weak**(T &*expected*, T *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_strong**(T &*expected*, T *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_strong**(T &*expected*, T *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

整数タイプに特殊化された atomic_ref クラス宣言

template <memory_order DefaultOrder, memory_scope
DefaultScope, access::address_space
Space = access::address_space::generic_space>
class atomic_ref <Integral, DefaultOrder, DefaultScope,
Space>;

メンバー

Integral **fetch_add**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Integral **fetch_sub**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Integral **fetch_and**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Integral **fetch_or**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Integral **fetch_min**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Integral **fetch_max**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

OP は ++、--

Integral **operatorOP**(int) const noexcept;
Integral **operatorOP**() const noexcept;

OP は +=、-=、&=、|=、^=

Integral **operatorOP**(Integral) const noexcept;

浮動小数点タイプに特殊化された atomic_ref クラス宣言

template <memory_order DefaultOrder, memory_scope
DefaultScope, access::address_space
Space = access::address_space::generic_space>
class atomic_ref<Floating, DefaultOrder, DefaultScope,
Space>;

メンバー

Floating **fetch_add**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Floating **fetch_sub**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Floating **fetch_min**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
Floating **fetch_max**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

OP は +=、-=

Floating **operatorOP**(Floating) const noexcept;

ポインタータイプに特殊化された atomic_ref クラス宣言

template <typename T, memory_order DefaultOrder,
memory_scope DefaultScope, access::address_space
Space = access::address_space::generic_space>
class atomic_ref<T*, DefaultOrder, DefaultScope, Space>;

コンストラクターとメンバー

explicit **atomic_ref**(T*&);
atomic_ref(const atomic_ref&) noexcept;
void **store**(T* *operand*,
memory_order *order* = default_write_order,
memory_scope *scope* = default_scope) const noexcept;
T* **operator=**(T* *desired*) const noexcept;
T* **load**(memory_order *order* = default_read_order,
memory_scope *scope* = default_scope) const noexcept;
operator T*() const noexcept;
T* **exchange**(T* *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_weak**(T* &*expected*, T* *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_weak**(T* &*expected*, T* *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
bool **compare_exchange_strong**(T* &*expected*, T* *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = *default_scope*) const noexcept;
bool **compare_exchange_strong**(T* &*expected*, T* *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
T* **fetch_add**(difference_type,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;
T* **fetch_sub**(difference_type,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

OP は ++、--

T* **operatorOP**(int) const noexcept;
T* **operatorOP**() const noexcept;

OP は +=、-=

T* **operatorOP**(difference_type) const noexcept;

スカラー・データ・タイプ [4.15]

SYCL は、C++ の基本データタイプ (sycl 名前空間ではない) と、バイトおよびハーフ・データ・タイプ (sycl 名前空間内) をサポートします。

クラス device_event [4.15.2]

このクラスは、SYCL カーネル関数内でのみ利用可能な単一の SYCL デバイスイベントをカプセル化し、SYCL カーネル関数内の非同期操作の完了を待機するのに使用できます。このクラスには、不特定のコンストラクターとほかのメンバーが 1 つあります。

void **wait**() noexcept;

クラス stream [4.16]

列挙型

stream_manipulator				
dec	noshowbase	noshowpos	endl	hexfloat
hex	showbase	showpos	fixed	defaultfloat
oct			scientific	flush

コンストラクターとメンバー

stream(size_t *totalBufferSize*, size_t *workItemBufferSize*,
handler& *cgh*, const property_list &*propList* = {});
size_t **size**() const noexcept;
size_t **get_work_item_buffer_size**() const;

非メンバー関数

template <typename T>
const stream&
operator<<(const stream& *os*, const T &*rhs*);

関数オブジェクト [4.17.2]

SYCL は、ホストとデバイスの sycl 名前空間で、C++ 変換とプロモーション規則に従う各種関数オブジェクトを提供します。

```
template <typename T=void>
  struct plus {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct multiplies {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct bit_and {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct bit_or {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct bit_xor {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct logical_and {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct logical_or {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct minimum {
  T operator()(const T & x, const T & y) const;
};
template <typename T=void>
  struct maximum {
  T operator()(const T & x, const T & y) const;
};
```

グループ関数とアルゴリズム

グループ関数 [4.17.3]

```
template <typename Group, typename T>
  bool group_broadcast(Group g, T x);
template <typename Group, typename T>
  T group_broadcast(Group g, T x,
    Group::linear_id_type local_linear_id);
template <typename Group, typename T>
  T group_broadcast(Group g, T x, Group::id_type local_id);
template <typename Group>
  void group_barrier(Group g,
    memory_scope fence_scope = Group::fence_scope);
```

グループのアルゴリズム [4.17.4]

```
template <typename Group, typename Ptr, typename Predicate>
  bool joint_any_of(Group g, Ptr first, Ptr last, Predicate pred);

template <typename Group, typename T, typename Predicate>
  bool any_of_group(Group g, T x, Predicate pred);

template <typename Group>
  bool any_of_group(Group g, bool pred);

template <typename Group, typename Ptr,
    typename Predicate>
  bool joint_all_of(Group g, Ptr first, Ptr last, Predicate pred);

template <typename Group, typename T,
    typename Predicate>
  bool all_of_group(Group g, T x, Predicate pred);

template <typename Group>
    bool all_of_group(Group g, bool pred);

template <typename Group, typename Ptr,
    typename Predicate>
  bool joint_none_of(Group g, Ptr first, Ptr last, Predicate pred);

template <typename Group, typename T,
    typename Predicate>
  bool none_of_group(Group g, T x, Predicate pred);

template <typename Group>
  bool none_of_group(Group g, bool pred);

template <typename Group, typename T>
  T shift_group_left(Group g, T x,
    Group::linear_id_type delta = 1);

template <typename Group, typename T>
  T shift_group_right(Group g, T x,
    Group::linear_id_type delta = 1);

template <typename Group, typename T>
  T permute_group_by_xor(Group g, T x,
    Group::linear_id_type mask);

template <typename Group, typename T>
  T select_from_group(Group g, T x,
    Group::id_type remote_local_id);

template <typename Group, typename Ptr,
    typename BinaryOperation>
  std::iterator_traits<Ptr>::value_type joint_reduce(Group g,
    Ptr first, Ptr last, BinaryOperation binary_op);

template <typename Group, typename Ptr, typename T,
    typename BinaryOperation>
  T joint_reduce(Group g, Ptr first, Ptr last, T init,
    BinaryOperation binary_op);
```

(続く) ▶

◀ グループ関数とアルゴリズム (続き)

```
template <typename Group, typename T, typename
      BinaryOperation>
   T reduce_over_group(Group g, T x,
      BinaryOperation binary_op);

template <typename Group, typename V, typename T,
      typename BinaryOperation>
   T reduce_over_group(Group g, V x, T init,
      BinaryOperation binary_op);

template <typename Group, typename InPtr, typename OutPtr,
      typename BinaryOperation>
   OutPtr joint_exclusive_scan(Group g, InPtr first, InPtr last,
      OutPtr result, BinaryOperation binary_op);

template <typename Group, typename InPtr, typename OutPtr,
      typename T, typename BinaryOperation>
   T joint_exclusive_scan(Group g, InPtr first, InPtr last,
      OutPtr result, T init, BinaryOperation binary_op);

template <typename Group, typename T,
      typename BinaryOperation>
   T exclusive_scan_over_group(Group g, T x, BinaryOperation
      binary_op);

template <typename Group, typename V, typename T,
      typename BinaryOperation>
   T exclusive_scan_over_group(Group g, V x, T init,
      BinaryOperation binary_op);

template <typename Group, typename InPtr, typename OutPtr,
      typename BinaryOperation>
   OutPtr joint_inclusive_scan(Group g, InPtr first, InPtr last,
      OutPtr result, BinaryOperation binary_op);

template <typename Group, typename InPtr, typename OutPtr,
      typename T, typename BinaryOperation>
   T joint_inclusive_scan(Group g, InPtr first, InPtr last,
      OutPtr result, BinaryOperation binary_op, T init);

template <typename Group, typename T,
      typename BinaryOperation>
   T inclusive_scan_over_group(Group g, T x, BinaryOperation
      binary_op);

template <typename Group, typename V, typename T,
      typename BinaryOperation>
   T inclusive_scan_over_group(Group g, V x,
      BinaryOperation binary_op, T init);
```

数学関数 [4.17.5]

数学関数は、ホストとデバイスの sycl 名前空間で使用できます。以下のすべてのケースで、n は 2、3、4、8、または 16 です。

Tf (仕様では genfloat) は、float[*n*]、double[*n*] 、または half[*n*] タイプです。
Tff (genfloatf) は float[*n*] タイプです。
Tfd (genfloatd) は double[*n*] タイプです。
Th (genfloath) は half[*n*] タイプです。
sTf (sgenfloat) は float、double、または half タイプです。
Ti (genint) は int[*n*] タイプです。
uTi (ugenint) は unsigned int または uint*n* タイプです。
uTli (ugenlonginteger) は unsigned long int、ulonglong*n*、ulong*n* 、unsigned long long int タイプです。

N は sycl::native で利用可能なネイティブバリアントを示します。
H は sycl::halfprecision で利用でき、最低 10 ビットの精度で実装されるハーフバリアントを示します。

Tf **acos** (*Tf x*)	逆余弦
Tf **acosh** (*Tf x*)	逆双曲線余弦
Tf **acospi** (*Tf x*)	acos (x) / π
Tf **asin** (*Tf x*)	逆正弦
Tf **asinh** (*Tf x*)	逆双曲線正弦
Tf **asinpi** (*Tf x*)	asin (x) / π
Tf **atan** (*Tf y_over_x*)	逆正接
Tf **atan2** (*Tf y*, *Tf x*)	y / x の逆正接
Tf **atanh** (*Tf x*)	双曲線逆正接
Tf **atanpi** (*Tf x*)	atan (x) / π
Tf **atan2pi** (*Tf y*, *Tf x*)	atan2 (y, x) / π
Tf **cbrt** (*Tf x*)	立方根
Tf **ceil** (*Tf x*)	+ 無限大への整数丸め
Tf **copysign** (*Tf x*, *Tf y*)	x の符号を y の符号に変更
Tf **cos** (*Tf x*) *Tff* **cos** (*Tff x*) N H	余弦
Tf **cosh** (*Tf x*)	双曲線余弦
Tf **cospi** (*Tf x*)	cos (π x)
Tff **divide** (*Tff x*, *Tff y*) N H	x / y (cl::sycl では利用できません)
Tf **erfc** (*Tf x*)	誤差関数の値を補完
Tf **erf** (*Tf x*)	誤差関数の値を計算

Tf **exp** (*Tf x*) *Tff* **exp** (*Tff x*)) N H	指数基数 e
Tf **exp2** (*Tf x*) *Tff* **exp2** (*Tff x*)) N H	指数基数 2
Tf **exp10** (*Tf x*) *Tff* **exp10** (*Tff x*)) N H	指数基数 10
Tf **expm1** (*Tf x*)	e^x -1.0
Tf **fabs** (*Tf x*)	絶対値
Tf **fdim** (*Tf x*, *Tf y*)	x と y の正の差
Tf **floor** (*Tf x*)	無限大への整数丸め
Tf **fma** (*Tf a*, *Tf b*, *Tf c*)	乗算と加算を行い丸める
Tf **fmax** (*Tf x*, *Tf y*) *Tf* **fmax** (*Tf x*, *sTf y*)	x < y の場合 y を返し、 それ以外は x を返す
Tf **fmin** (*Tf x*, *Tf y*) *Tf* **fmin** (*Tf x*, *sTf y*)	y < x の場合 y を返し、 それ以外は x を返す
Tf **fmod** (*Tf x*, *Tf y*)	係数。 x – y * trunc (x/y) を返す
Tf **fract** (*Tf x*, *Tf*iptr*)	x の小数値
Tf **frexp** (*Tf x*, *Ti*exp*)	仮数と指数を抽出
Tf **hypot** (*Tf x*, *Tf y*)	$x^2 + y^2$ の平方根
Ti **ilogb** (*Tf x*)	指数を整数値として返す
Tf **ldexp** (*Tf x*, *Ti k*) *doublen* **ldexp** (*doublen x*, int *k*)	x * 2n
Tf **lgamma** (*Tf x*)	対数ガンマ関数
Tf **lgamma_r** (*Tf x*, *Ti *signp*)	対数ガンマ関数
Tf **log** (*Tf x*) *Tff* **log** (*Tff x*)) N H	自然対数
Tf **log2** (*Tf x*) *Tff* **log2** (*Tff x*)) N H	2 を底とする対数
Tf **log10** (*Tf x*) *Tff* **log10** (*Tff x*)) N H	10 を底とする対数
Tf **log1p** (*Tf x*)	ln (1.0 + x)
Tf **logb** (*Tf x*)	指数を整数値として返す
Tf **mad** (*Tf a*, *Tf b*, *Tf c*)	a * b + c の近似値
Tf **maxmag** (*Tf x*, *Tf y*)	x と y の最大値

Tf **minmag** (*Tf x*, *Tf y*)	x と y の最小値
Tf **modf** (*Tf x*, *Tf*iptr*)	浮動小数点数を分解
Tff **nan** (*uTi nancode*) *Tfd* **nan** (*uTli nancode*)	Quiet NaN (*nancode* がスカラーの場合、 戻り値はスカラー)
Tf **nextafter** (*Tf x*, *Tf y*)	Y 方向の x の次の表現可能な 浮動小数点値
Tf **pow** (*Tf x*, *Tf y*)	x を y の累乗で計算
Tf **pown** (*Tf x*, *Ti y*)	xy を計算 (y は整数)
Tf **powr** (*Tf x*, *Tf y*) *Tff* **powr** (*Tff x*, *Tff y*) N *Tff* **powr** (*Tff x*, *Th y*) H	xy を計算 (x は >= 0)
Tff **recip** (*Tff x*) N H	1 / x (cl::sycl では利用できません)
Tf **remainder** (*Tf x*, *Tf y*)	浮動小数点余剰
Tf **remquo** (*Tf x*, *Tf y*, *Ti *quo*)	余剰と商
Tf **rint** (*Tf x*)	最も近い偶数の整数値へ丸め
Tf **rootn** (*Tf x*, *Ti y*)	x を 1/y の累乗で計算
Tf **round** (*Tf x*)	x の丸めに最も近い整数値
Tf **rsqrt** (*Tf x*) *Tff* **rsqrt** (*Tff x*) N H	逆数平方根
Tf **sin** (*Tf x*) *Tff* **sin** (*Tff x*)	正弦
Tf **sincos** (*Tf x*, *Tf *cosval*)	x の正弦と余弦
Tf **sinh** (*Tf x*)	双曲線正弦
Tf **sinpi** (*Tf x*)	sin (π x)
Tf **sqrt** (*Tf x*) *Tff* **sqrt** (*Tff x*)	平方根
Tf **tan** (*Tf x*) *Tff* **tan** (*Tff x*)) N H	正接
Tf **tanh** (*Tf x*)	双曲線正接
Tf **tanpi** (*Tf x*)	tan (π x)
Tf **tgamma** (*Tf x*)	ガンマ関数
Tf **trunc** (*Tf x*)	ゼロ方向への整数丸め

整数関数 [4.17.6]

整数関数は名前空間 sycl で利用できます。以下のすべてのケースで、n は 2、3、4、8、または 16 です。関数タイプ名に [*x*bit] が含まれる場合、そのタイプのサイズは *x* ビットであることを示しています。パラメーター・タイプは、vec および marray にも対応しています。

Tint (仕様では geninteger) は、int[*n*]、uint[*n*]、unsigned int、char、char[*n*], signed char、schar*n*、uchar*n*、unsigned short[*n*]、unsigned short、ushort[*n*]、long*n*、ulong*n*、long int、unsigned long int、long long int、longlong*n*、ulonglong*n*、unsigned long long int タイプです。

uTint (ugeninteger) は、unsigned char、uchar*n*、unsigned short、ushort*n*、unsigned int、uint*n*、unsigned long int、ulong*n*、ulonglong*n*、unsigned long long int タイプです。

iTint (igeninteger) は、signed char、schar*n*、short[*n*]、int[*n*]、long int、long*n*、long long int、longlong*n* タイプです。

sTint (sgeninteger) は、char、signed char、unsigned char、short、unsigned short、int、unsigned int、long int、unsigned long int、long long int、unsigned long long int タイプです。

uTint **abs** (*Tint x*)	| *x* |
uTint **abs_diff** (*Tint x*, *Tint y*)	モジュロ・オーバーフローなしの | x – y |
Tint **add_sat** (*Tint x*, *Tint y*)	x + y の結果を飽和
Tint **hadd** (*Tint x*, *Tint y*)	モジュロ・オーバーフローなしの (x + y) >> 1

Tint **rhadd** (*Tint x*, *Tint y*)	(x + y + 1) >> 1
Tint **clamp** (*Tint x*, *Tint min*, *Tint max*) *Tint* **clamp** (*Tint x*, *sTint min*, *sTint max*)	min(max(x, min), max)
Tint **clz** (*Tint x*)	最上位ビット位置から始まる x の先行 0 ビットの数。 x が 0 の場合、x がベクトルであれば x のタイプまたは x のコンポーネント・タイプのビット単位のサイズを返す

(続く) ▶

©2021 Khronos Group - Rev. 0221

◀ 整数関数 (続き)

Tint **ctz** (*Tint x*)	x のトレール 0 ビットの数。x が 0 の場合、x がベクトルであれば x のタイプまたは x のコンポーネント・タイプのビット単位のサイズを返す
Tint **mad_hi** (*Tint a*, *Tint b*, *Tint c*)	mul_hi(a, b) + c
Tint **mad_sat** (*Tint a*, *Tint b*, *Tint c*)	a * b + c の結果を飽和
Tint **max** (*Tint x*, *Tint y*) *Tint* **max** (*Tint x*, *sTint y*)	x < y の場合 y を返し、それ以外は x を返す
Tint **min** (*Tint x*, *Tint y*) *Tint* **min** (*Tint x*, *sTint y*)	y < x の場合 y を返し、それ以外は x を返す
Tint **mul_hi** (*Tint x*, *Tint y*)	x と y の積の上位半分

Tint **popcount** (*Tint x*)	x の非ゼロ・ビットの数
Tint **rotate** (*Tint v*, *Tint i*)	result[indx] = v[indx] << i[indx]
Tint **sub_sat** (*Tint x*, *Tint y*)	x - y の結果を飽和させます
uTint16bit **upsample** (*uTint8bit hi*, *uTint8bit lo*)	result[i]= ((ushort)hi[i]<< 8)|lo[i]
iTint16bit **upsample** (*iTint8bit hi*, *uTint8bit lo*)	result[i]=((short)hi[i]<< 8)|lo[i]
uTint32bit **upsample** (*uTint16bit hi*, *uTint16bit lo*)	result[i]=((uint)hi[i]<< 16)|lo[i]
iTint32bit **upsample** (*iTint16bit hi*, *uTint16bit lo*)	result[i]=((int)hi[i]<< 16)|lo[i]

uTint64bit **upsample** (*uTint32bit hi*, *uTint32bit lo*)	result[i]= ((ulonglong)hi[i]<< 32)|lo[i]
iTint64bit **upsample** (*iTint32bit hi*, *uTint32bit lo*)	result[i]= ((longlong)hi[i]<< 32)|lo[i]
Tint32bit **mad24** (*Tint32bit x*, *Tint32bit y*, *Tint32bit z*) *Tint32bit* **mad24** (*Tint32bit x*, *Tint32bit y*, *Tint32bit z*)	24 ビット整数値 x と y を乗算し、32 ビット整数結果を 32 ビット整数 z に加算
Tint32bit **mul24** (*Tint32bit x*, *Tint32bit y*)	24 ビット整数値 x と y を乗算

共通関数 [4.17.7]

共通関数は、ホストとデバイスの sycl 名前空間で使用できます。ホストではベクトルタイプは vec クラスを使用し、OpenCL デバイスでは対応する OpenCL ベクトルタイプを使用します。以下のすべてのケースで、n は 2、3、4、8、または 16 です。ビルトイン関数は入力として float (またはオプションで double)、およびそれらの vec または marray に対応しています。

Tf (仕様では genfloat) は、float[*n*]、double[*n*]、または half[*n*] タイプです。
Tff (genfloatf) は float[*n*] タイプです。
Tfd (genfloatd) は double[*n*] タイプです

Tf **clamp** (*Tf x*, *Tf minval*, *Tf maxval*); *Tff* **clamp** (*Tff x*, *float minval*,float *maxval*); *Tfd* **clamp** (*Tfd x*, double *minval*, double*n maxval*);	x を minval、maxval で指定された範囲にクランプ
Tf **degrees** (*Tf radians*);	ラジアンを度に変換
Tf **abs** (*Tf x*, *Tf y*); *Tff* **abs** (*Tff x*, float *y*); *Tfd* **abs** (*Tfd x*, double *y*);	x と y の最大値
Tf **max** (*Tf x*, *Tf y*); *Tff* **max** (*Tff x*, float *y*); *Tfd* **max** (*Tfd x*, double *y*);	x と y の最大値
Tf **min** (*Tf x*, *Tf y*); *Tff* **min** (*Tff x*, float *y*); *Tfd* **min** (*Tfd x*, double *y*);	x と y の最小値
Tf **mix** (*Tf x*, *Tf y*, *Tf a*); *Tff* **mix** (*Tff x*, *Tff y*, float *a*); *Tfd* **mix** (*Tfd x*, *Tfd y*, double *a*) ;	x と x の線形混合
Tf **radians** (*Tf degrees*);	度からラジアンへの変換
Tf **step** (*Tf edge*, *Tf x*); *Tff* **step** (float *edge*, *Tff x*); *Tfd* **step** (double *edge*, *Tfd x*);	x < edge の場合は 0.0、それ以外は 1.0
Tf **smoothstep** (*Tf edge0*, *Tf edge1*, *Tf x*); *Tff* **smoothstep** (float *edge0*, float *edge1*, *Tff x*); *Tfd* **smoothstep** (double *edge0*, double *edge1*, *Tfd x*);	ステップと補間
Tf **sign** (*Tf x*);	x の正弦

プリプロセッサー・ディレクティブとマクロ [5.6]

SYCL_LANGUAGE_VERSION	整数バージョン 例: 202002
__ SYCL_DEVICE_ONLY__	デバイスのコンパイル時は == 1
__ SYCL_SINGLE_SOURCE__	ホストとデバイスの両方のバイナリーを作成する場合は == 1
SYCL_EXTERNAL	定義されている場合、外部カーネルリンクのサポートを示す

関係ビルトイン関数 [4.17.9]

関係関数は、ホストとデバイスの名前空間 sycl で使用できます。以下のすべてのケースで、n は 2、3、4、8、または 16 です。関数タイプ名に [*x*bit] が含まれる場合、そのタイプのサイズは *x* ビットであることを示しています。

Tint (仕様では geninteger) は、int[*n*]、uint[*n*]、unsigned int、char、char[*n*]、signed char、schar*n*、uchar*n*、unsigned short[*n*]、unsigned short、ushort[*n*]、long*n*、ulong*n*、long int、unsigned long int、long long int、longlong*n*、ulonglong*n*、unsigned long long int タイプです。

iTint (igeninteger) は、signed char、schar*n*、short[*n*]、int[*n*]、long int、long*n*、long long int、longlong*n* タイプです。

uTint (ugeninteger) は、unsigned char、uchar*n*、unsigned short、ushort*n*、unsigned int、uint*n*、unsigned long int、ulong*n*、ulonglong*n*、unsigned long long int です。

Ti (genint) は int[*n*] タイプです。

uTi (ugenint) は unsigned int または uint*n* タイプです。

Tff (genfloatf) は float[*n*] タイプです。

Tfd (genfloatd) は double[*n*] タイプです。

T (gentype) は、float[*n*]、double[*n*]、half[*n*]、または上記の Tint のいずれかのタイプです。

int any (*iTint x*)	x のコンポーネントの MSB が設定されている場合は 1、それ以外は 0
int **all** (*iTint x*)	x のすべてのコンポーネントの MSB が設定されている場合は 1、それ以外は 0
T **bitselect** (*T a*, *T b*, *T c*)	c の対応するビットが 0 である場合、結果の各ビットは a の対応するビット
Tint **select** (*Tint a*, *Tint b*, *iTint c*) *Tint* **select** (*Tint a*, *Tint b*, *uTint c*) *Tff* **select** (*Tff a*, *Tff b*, *Ti c*) *Tff* **select** (*Tff a*, *Tff b*, *uTi c*) *Tfd* **select** (*Tfd a*, *Tfd b*, *iTint*64bit *c*) *Tfd* **select** (*Tfd a*, *Tfd b*, *uTint*64bit *c*)	ベクトルタイプの各コンポーネントについて、c[i] の MSB が設定されていれば、result = c[i] ? b[i] : a[i]。スカラータイプでは result = c ? b :a
*iTint*32bit ***function*** (*Tff* x, *Tff* y) *iTint*64bit ***function*** (*Tfd* x, *Tfd* y) ***function***: isequal, isnotequal, isgreater, isgreaterequal, isless, islessequal, islessgreater, isordered, isunordered.	この形式は、多くの関係関数で使用。***function*** を関数名に置き換える
*iTint*32bit ***function*** (*Tff* x) *iTint*64bit ***function*** (*Tfd* x) ***function***: isfinite, isinf, isnan, isnormal, signbit.	

汎用関数 [4.17.8]

汎用関数は、ホストとデバイスの sycl 名前空間で使用できます。ビルトイン関数は入力として float (またはオプションで double)、およびそれらの 2、3、4 次元の vec または marray に対応しています。ホストではベクトルタイプは vec クラスを使用し、SYCL デバイスでは対応する SYCL バックエンドのベクトルタイプを使用します。

Tgf (仕様では gengeofloat) は float、float2、float3、float4 タイプです。
Tgd (gengeodouble) は double、double2、double3、double4 タイプです。

float4 **cross** (float4 *p0*, float4 *p1*) float3 **cross** (float3 *p0*, float3 *p1*) double4 **cross** (double4 *p0*, double4 *p1*) double3 **cross** (double3 *p0*, double3 *p1*)	クロス積
float **distance** (*Tgf p0*, *Tgf p1*) double **distance** (*Tgd p0*, *Tgd p1*)	ベクトルの距離
float **dot** (*Tgf p0*, *Tgf p1*) double **dot** (*Tgd p0*, *Tgd p1*)	ドット積
float **length** (*Tgf p*) double **length** (*Tgd p*)	ベクトル長
Tgf **normalize** (*Tgf p*) *Tgd* **normalize** (*Tgd p*)	通常のベクトル長は 1
float **fast_distance** (*Tgf p0*, *Tgf p1*)	ベクトルの距離
float **fast_length** (*Tgf p*)	ベクトル長
Tgf **fast_normalize** (*Tgf p*)	通常のベクトル長は 1

カーネル属性 [5.8.1]

次のように属性を適用できます。

```
[=] (item<1> it) [[sycl::reqd_work_group_size(16)]] {
    // [カーネルコード]
}
```

```
void operator()(item<1> it) const [[sycl::reqd_work_
    group_size(16)]] {
    // [カーネルコード]
};
```

属性

reqd_work_group_size(dim0)
reqd_work_group_size(dim0, dim1)
reqd_work_group_size(dim0, dim1, dim2)
work_group_size_hint(dim0)
work_group_size_hint(dim0, dim1)
work_group_size_hint(dim0, dim1, dim2)
vec_type_hint(<type>)
reqd_sub_group_size(dim)

デバイス関数属性 [5.8.2]

次の属性は、非カーネルデバイス関数の宣言に適用されます。

sycl::requires(has(aspect,...))

©2021 Khronos Group - Rev. 0221

バックエンド [4.1]

Khronos で定義されたバックエンドは、SYCL_BACKEND_BACKEND_NAME 形式のマクロに関連付けられています。利用可能な SYCL バックエンドは、enum class backend で識別されます。

```
enum class backend {
    実装定義
};
```

バックエンドの相互運用性 [4.5.1]

SYCL バックエンド固有の動作に依存する SYCL アプリケーションは、sycl/sycl.hpp ヘッダーに加えて SYCL バックエンド固有のヘッダーをインクルードする必要があります。
SYCL バックエンドの相互運用性サポートはオプションです。
SYCL バックエンドの相互運用性を使用する SYCL アプリケーションは、汎用性がない SYCL と見なされます。

バックエンドのタイプ特性、テンプレート関数

```
template <backend Backend>
class backend_traits {
   public:
      template <class T>
      using input_type = backend-specific;

      template <class T>
      using return_type = backend-specific;

      using errc = backend-specific;
};

template <backend Backend, typename SyclType>
using backend_input_t =
      typename backend_traits<Backend>::template
         Input_type<SyclType>;

template <backend Backend, typename SyclType>
using backend_return_t =
      typename backend_traits<Backend>::template
         return_type<SyclType>;
```

get_native (4.5.1.2)

syclObject に関連する SYCL アプリケーションの相互運用ネイティブ・バックエンド・オブジェクトを返します。これは、SYCL アプリケーションの相互運用に利用できます。

```
template <backend Backend, class T>
   backend_return_t<Backend, T>
   get_native(const T &syclObject);
```

バックエンド関数 (4.5.1.3)

```
template <backend Backend>
   platform make_platform(const backend_input_t <Backend,
      platform> &backendObject);

template<backend Backend>
   device make_device(const backend_input_t<Backend,
      device> &backendObject);

template <backend Backend>
   context make_context(const backend_input_t<Backend,
      context> &backendObject,
      const async_handler asyncHandler = {});

template <backend Backend>
   queue make_queue(const backend_input_t<Backend,
      queue> &backendObject,
      const context &targetContext,
      const async_handler asyncHandler = {});

template <backend Backend>
   event make_event(const backend_input_t<Backend, event>
      &backendObject,
      const context &targetContext);

template <backend Backend, typename T, int dimensions = 1,
      typename AllocatorT = buffer_allocator<
      std::remove_const_t<T>>>
   buffer make_buffer(const backend_input_t<Backend,
      buffer<T, dimensions, AllocatorT>> &backendObject,
      const context &targetContext,
      event availableEvent = {});

template <backend Backend, typename T, int dimensions = 1,
      typename AllocatorT = buffer_allocator<
      std::remove_const_t<T>>>
   buffer<T, dimensions, AllocatorT> make_buffer(const
      backend_input_t<Backend, buffer<T, dimensions,
      AllocatorT>> &backendObject,
      const context &targetContext);

template <backend Backend, int dimensions = 1,
      typename AllocatorT = sycl::image_allocator>
   sampled_image<dimensions, AllocatorT>
      make_sampled_image(
      const backend_input_t<Backend, sampled_image
      <dimensions, AllocatorT>> &backendObject,
      const context &targetContext,
      image_sampler imageSampler,
      event availableEvent = {});

template <backend Backend, int dimensions = 1,
      typename AllocatorT = sycl::image_allocator>
   sampled_image<dimensions, AllocatorT>
      make_sampled_image(
      const backend_input_t<Backend, sampled_image
      <dimensions, AllocatorT>> &backendObject,
      const context &targetContext,
      image_sampler imageSampler);

template <backend Backend, int dimensions = 1,
      typename AllocatorT = sycl::image_allocator>
   unsampled_image<dimensions, AllocatorT>
      make_unsampled_image(
      const backend_input_t<Backend, unsampled_image
      <dimensions, AllocatorT>> &backendObject,
      const context &targetContext,
      event availableEvent);

template <backend Backend, int dimensions = 1,
      typename AllocatorT = sycl::image_allocator>
   unsampled_image<dimensions, AllocatorT>
      make_unsampled_image(
      const backend_input_t<Backend, unsampled_image
      <dimensions, AllocatorT>> &backendObject,
      const context &targetContext);

template <backend Backend, bundle_state State>
   kernel_bundle<State> make_kernel_bundle(
      const backend_input_t<Backend,
      kernel_bundle<State>> &backendObject,
      const context &targetContext);

template <backend Backend>
   kernel make_kernel(const backend_input_t<Backend,
      kernel> &backendObject,
      const context &targetContext);
```

カーネルバンドル [4.11]

カーネルバンドルはコンテキストに関連付けられ、同じコンテキストに関連する複数のデバイスで実行できるカーネルのセットを表す高レベルの抽象化です。

バンドルの状態

バンドルの状態	カーネルバンドルのデバイスイメージの形式
bundle_state::input	カーネルを呼び出す前に、コンパイルしてリンクする必要があります。
bundle_state::object	カーネルを呼び出す前に、リンクする必要があります。
bundle_state::executable	デバイスで呼び出すことができます。

カーネル識別子 [4.11.6]

カーネルバンドルに関連する一部の関数は、kernel_id タイプの入力パラメーターを受け付けます。これは、メンバーを持つクラスです。

```
const char *get_name() const noexcept;
```

カーネルバンドルの取得 [4.11.7]

フリー関数:

```
std::vector<kernel_id> get_kernel_ids();

template <typename KernelName>
   kernel_id get_kernel_id();
```

カーネルバンドルの取得 [4.11.7]

フリー関数:

```
template<bundle_state State>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs);
template<bundle_state State>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs,
      const std::vector<kernel_id> &kernelIds);
template<bundle_state State, typename Selector>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs, Selector selector);
template<bundle_state State>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt);
```

```
template<bundle_state State>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt,
      const std::vector<kernel_id> &kernelIds);

template<bundle_state State, typename Selector>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt, Selector selector);

template<typename KernelName, bundle_state State>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt);

template<typename KernelName, bundle_state State>
   kernel_bundle<State>
   get_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs);
```

バンドルが存在するか照会 [4.11.8]

フリー関数:

```
template<bundle_state State>
   bool has_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs);

template<bundle_state State>
   bool has_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs,
      const std::vector<kernel_id> &kernelIds);

template<bundle_state State>
   bool has_kernel_bundle(const context &ctxt);

template<bundle_state State>
   bool has_kernel_bundle(const context &ctxt,
      const std::vector<kernel_id> &kernelIds);

template<typename KernelName, bundle_state State>
   bool has_kernel_bundle(const context &ctxt);

template<typename KernelName, bundle_state State>
   bool has_kernel_bundle(const context &ctxt,
      const std::vector<device> &devs);
```

カーネルがデバイスと互換性があるか照会 [4.11.9]

フリー関数:

```
bool is_compatible(const std::vector<kernel_id> &kernelIds,
   const device &dev);

template<typename KernelName>
   bool is_compatible(const device &dev);
```

カーネルバンドルのジョイン [4.11.10]

```
template<bundle_state State> kernel_bundle<State>
   join(const std::vector<kernel_bundle<State>> &bundles);
```

オンラインコンパイルとリンク [4.11.11]

フリー関数:

```
kernel_bundle<bundle_state::object>
   compile(const kernel_bundle <
      bundle_state::input> &inputBundle,
      const std::vector<device> &devs,
      const property_list &propList = {});

kernel_bundle<bundle_state::executable>
   link(const std::vector<kernel_bundle <
      bundle_state::object>> &objectBundles,
      const std::vector<device> &devs,
      const property_list &propList = {});

kernel_bundle<bundle_state::executable>
   build(const kernel_bundle <
      bundle_state::input> &inputBundle,
      const std::vector<device> &devs,
      const property_list &propList = {});

kernel_bundle<bundle_state::object>
   compile(const kernel_bundle <
      bundle_state::input> &inputBundle,
      const property_list &propList = {});

kernel_bundle<bundle_state::executable>
   link(const kernel_bundle <
      bundle_state::object> &objectBundle,
      const std::vector<device> &devs,
      const property_list &propList = {});

kernel_bundle<bundle_state::executable>
   link(const std::vector<kernel_bundle <
      bundle_state::object>> &objectBundles,
      const property_list &propList = {});

kernel_bundle<bundle_state::executable>
   link(const kernel_bundle <
      bundle_state::object> &objectBundle,
      const property_list &propList = {});

kernel_bundle<bundle_state::executable>
   build(const kernel_bundle <
      bundle_state::input> &inputBundle,
      const property_list &propList = {});
```

カーネル・バンドル・クラス [4.11.12]

クラス宣言

```
template<bundle_state State>class kernel_bundle;
```

メンバー

```
bool empty() const noexcept;
backend get_backend() const noexcept;
context get_context() const noexcept;
```

(続く) ▶

◀ カーネルバンドル (続き)

```
std::vector<device>
    get_devices() const noexcept;

bool has_kernel(const kernel_id &kernelId) const noexcept;

bool has_kernel(const kernel_id &kernelId, const device &dev)
    const noexcept;

std::vector<kernel_id>
    get_kernel_ids() const;
```

State == bundle_state::executable の場合に利用可能

```
kernel get_kernel(const kernel_id &kernelId) const;

bool contains_specialization_constants() const noexcept;

bool native_specialization_constant() const noexcept;

template<auto& S>
    bool has_specialization_constant() const noexcept;
```

State == bundle_state::input の場合に利用可能

```
template<auto& S>
    void set_specialization_constant(typename
        std::remove_reference_t<decltype(S)>::type value);

template<auto& S>
    typename std::remove_reference_t<decltype(S)>::type
    get_specialization_constant() const;

device_image_iterator begin() const;

device_image_iterator end() const;
```

カーネルクラス [4.11.13]

```
backend get_backend() const noexcept;

context get_context() const;

kernel_bundle<bundle_state::executable>
    get_kernel_bundle() const;

template <typename param>
    typename param::return_type
    get_info() const;

template <typename param>
    typename param::return_type
    get_info(const device &dev) const;

template <typename param>
    typename param::return_type
    get_backend_info() const;
```

get_info() で照会:

記述子	戻り値
info::kernel_device_specific::global_work_size	range<3>
info::kernel_device_specific::work_group_size	size_t
info::kernel_device_specific::compile_work_group_size	range<3>
info::kernel_device_specific::preferred_work_group_size_multiple	size_t
info::kernel_device_specific::private_mem_size	size_t
info::kernel_device_specific::max_num_sub_groups	uint_32
info::kernel_device_specific::compile_num_sub_groups	uint_32
info::kernel_device_specific::max_sub_group_size	uint_32
info::kernel_device_specific::compile_sub_group_size	uint_32

デバイス・イメージ・クラス [4.11.14]

クラス宣言

```
template<bundle_state State>class device_image;
```

メンバー

```
bool has_kernel(const kernel_id &kernelId) const noexcept;
bool has_kernel(const kernel_id &kernelId, const device &dev)
    const noexcept;
```

USM の例

USM 共有割り当ての例

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (オプション) SYCL 名の前に “sycl::” を記述する必要なし
int main() {

    // ワークを送信するデフォルトキューを作成
    queue myQueue;

    // デバイスにバインドされた共有メモリーとキューに関連するコンテキストを割り当て
    // malloc_shared を malloc_host に置き換えると、デバイスで使用可能なメモリーを
    // ホストに割り当てたプログラムを作成できる
    int *data = sycl::malloc_shared<int>(1024, myQueue);

    myQueue.parallel_for(1024, [=](id<1> idx) {
        // 0 から始まるランク番号でそれぞれのバッファー要素を初期化
        data[idx] = idx;
    }); // カーネル関数の終端

    myQueue.wait();

    // 結果を出力
    for (int i = 0; i < 1024; i++)
        std::cout <<"data["<< i << "] = " << data[i] << std::endl;

    return 0;
}
```

USM デバイス割り当ての例

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (オプション) SYCL 名の前に “sycl::” を記述する必要なし
int main() {

    // ワークを送信するデフォルトキューを作成
    queue myQueue;

    //デバイスにバインドされた共有メモリーとキューに関連するコンテキストを割り当て
    int *data = sycl::malloc_device<int>(1024, myQueue);

    myQueue.parallel_for(1024, [=](id<1> idx) {
        // 0 から始まるランク番号でそれぞれのバッファー要素を初期化
        data[idx] = idx;
    }); // カーネル関数の終端

    myQueue.wait();

    int hostData[1024];
    myQueue.memcpy(hostData, data, 1024*sizeof(int));

    myQueue.wait();

    // 結果を出力
    for (int i = 0; i < 1024; i++)
        std::cout <<"data["<< i << "] = " << data[i] << std::endl;

    return 0;
}
```

カーネルを呼び出す方法の例

例: single_task 起動 [4.9.4.2.1]

SYCL は、OpenCL デバイスでシリアル実行されるカーネルをキューに送信する簡単なインターフェイスを提供します。

```
myQueue.submit([&](handler & cgh) {
    cgh.single_task(
        [=] () {
        // [カーネルコード]
    }));
});
```

例: parallel_for 起動 [4.9.4.2.2]
例 1

カーネルの起動にラムダ関数を使用します。この **parallel_for** のバリアントは、実行されるインデックス空間のグローバル範囲を照会する必要がない用途向けに設計されています。

```
myQueue.submit([&](handler & cgh) {
    accessor acc { myBuffer, cgh, write_only };

    cgh.parallel_for(range<1>(numWorkItems),
            [=] (id<1> index) {
        acc[index] = 42.0f;
    });
});
```

例 2

ラムダ関数を使用して item パラメーターを渡すため、**parallel_for** を使用して SYCL カーネル関数を呼び出します。この **parallel_for** のバリアントは、実行されるインデックス空間のグローバル範囲を照会する用途向けに設計されています。

```
myQueue.submit([&](handler & cgh) {
    accessor acc { myBuffer, cgh, write_only };

    cgh.parallel_for(range<1>(numWorkItems),
            [=] (item<1> item) {
        // カーネル引数タイプは item
        size_t index = item.get_linear_id();
        acc[index] = index;
    });
});
```

例 3

次の 2 つの例は、各次元に 3 つの要素がある 3D グリッドでカーネル関数オブジェクトを起動する方法を示しています。最初の例ではワーク項目 ID の範囲は 0 から 2 であり、次の例のワーク項目 ID の範囲は 1 から 3 です。

```
myQueue.submit([&](handler & cgh) {
    cgh.parallel_for(
        range<3>(3,3,3), //グローバルレンジ
        [=] (item<3> it) {
            //[カーネルコード]
    });
});
```

例 4

各次元に 4 つずつ、8 つのワークグループに分割された 3 次元グリッド上の 64 個のワーク項目を起動します。

```
myQueue.submit([&](handler & cgh) {
    cgh.parallel_for(
        nd_range<3>(range<3>(4, 4, 4), range<3>(2, 2, 2)), [=] (nd_item<3> item) {
        // [カーネルコード]
        // 内部同期
        group_barrier(item.get_group());
        // [カーネルコード]
    });
});
```

階層的な呼び出しの並列処理 [4.9.4.2.3]

次の例は、8 つのワークグループを発行し、**parallel_for_work_group** 呼び出しにワークグループのサイズを渡さず、ランタイムにサイズを決定します。**parallel_for_work_item** ループのサイズも可変であり、その実行範囲はワークグループの次元とは関係なく、コンパイラーがギャップを埋めるため適切な反復空間を生成します。この場合、h_item はカーネルと **parallel_for_work_item** の呼び出し範囲の両方を反映するローカル id と範囲へのアクセスを提供します。

```
myQueue.submit([&](handler & cgh) {
  // それぞれ 8 つの work-item からなる 8 つのワークグループを発行

  cgh.parallel_for_work_group(range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {


    // [work-group のコード]
    int myLocal; // この変数は work-item 間で共有される
    // この変数は work-item ごとに個別にインスタンス化される
    private_memory<int> myPrivate(myGroup);

    // 並列 work-item を発行。ワークグループごとに発行される数は、parallel_for_work_group の
    // ワークグループのサイズによって決定される。この場合、8 つの work-item が
    // 8 つのワークグループのそれぞれに対し parallel_for_work_item のボディーを実行し、
    // 結果としてグローバル/合計で 64 回実行される
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      // [work-item コード]
      myPrivate(myItem) = 0;
    });

    // 暗黙の work-group のバリアー

    // ループ間でプライベート変数を伝搬
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      // [work-item コード]
      output[myItem.get_global_id()] = myPrivate(myItem);
    });
    // [work-group コード]
  });
});
```

このSYCL 2020 API リファレンス・ガイドは、Khronos Group のウェブサイト: https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf で公開されている『SYCL 2020 Reference Guide』の日本語参考訳です。

Khronos Group の許可を得て iSUS (IA Software User Society) が翻訳版を作成した iSUS の著作物です。原文は Khronos Group の Copyright であり、日本語参考訳版にも適用されます。

日本語訳に不明または誤りがある場合、原文を優先します。

SYCLcon International Workshop on SYCL

SYCL 開発者、研究者、サプライヤー、および Khronos SYCL ワーキンググループのメンバーによる国際コミュニティーが毎年集まり、ベストプラクティスを共有し、異種プラットフォームの C++ プログラムの SYCL 標準の利用と進化を促進しています。

syclcon.org

© 2021 Khronos Group. All rights reserved. SYCL は Khronos Group の商標です。Khronos Group は、各種プラットフォームやデバイスにおける並列コンピューティング、グラフィックス、ダイナミック・メディアなどのオーサリングと高速化のためオープン標準を作成する業界コンソーシアムです。Khronos Group の詳細については、www.khronos.org を参照してください。SYCL の詳細については、www.khronos.org/sycl を参照してください。http://www.khronos.org/