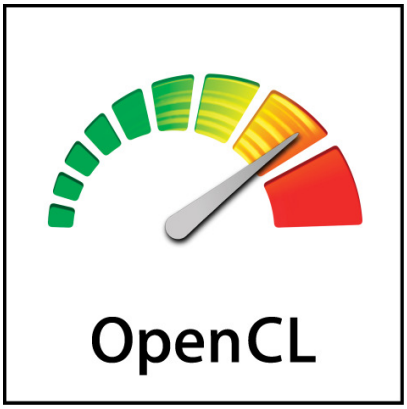

The OpenCL™ SPIR-V Environment Specification

Khronos OpenCL Working Group

Version 2.2-7, Sat, 12 May 2018 13:21:25 +0000

Table of Contents

1. Introduction . . . 2
2. SPIR-V Consumption . . . 3
   2.1. Validation Rules . . . 3
   2.2. Source Language Encoding . . . 4
   2.3. Numerical Type Formats . . . 4
   2.4. Supported Types . . . 4
   2.5. Image Channel Order Mapping . . . 5
   2.6. Image Channel Data Type Mapping . . . 6
   2.7. Kernels . . . 7
   2.8. Kernel Return Types . . . 7
   2.9. Kernel Arguments . . . 7
3. OpenCL 2.2 . . . 9
   3.1. Full Profile . . . 9
   3.2. Embedded Profile . . . 9
   3.3. Validation Rules . . . 10
4. OpenCL 2.1 . . . 12
   4.1. Full Profile . . . 12
   4.2. Embedded Profile . . . 12
   4.3. Validation Rules . . . 13
5. OpenCL 2.0 . . . 15
   5.1. Full Profile . . . 15
   5.2. Embedded Profile . . . 15
   5.3. Validation Rules . . . 16
6. OpenCL 1.2 . . . 18
   6.1. Full Profile . . . 18
   6.2. Embedded Profile . . . 18
   6.3. Validation Rules . . . 19
7. OpenCL Extensions . . . 21
   7.1. Declaring SPIR-V Extensions . . . 21
   7.2. Full and Embedded Profile Extensions . . . 21
   7.3. Embedded Profile Extensions . . . 24
8. OpenCL Numerical Compliance . . . 25
   8.1. Rounding Modes . . . 25
   8.2. Rounding Modes for Conversions . . . 25
   8.3. Out-of-Range Conversions . . . 26
   8.4. INF, NaN, and Denormalized Numbers . . . 26
   8.5. Floating-Point Exceptions . . . 26
   8.6. Relative Error as ULPs . . . 27

8.7. Edge Case Behavior . . . 37
9. Image Addressing and Filtering . . . 42
9.1. Image Coordinates . . . 42
9.2. Addressing and Filter Modes . . . 42
9.3. Precision of Addressing and Filter Modes . . . 48
9.4. Conversion Rules . . . 48
9.5. Selecting an Image from an Image Array . . . 55
9.6. Data Format for Reading and Writing Images . . . 56
9.7. Sampled and Sampler-less Reads . . . 57
10. Normative References . . . 58

Chapter 1. Introduction

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

Parallel programs in OpenCL may be written in the OpenCL C source language, or may compiled from OpenCL C, OpenCL C++, or other source languages into SPIR-V modules.

All SPIR-V intermediate binary modules are consumed by environments, such as an API, a specific version of an API, or an implementation of an API. The environment describes required support for some SPIR-V capabilities, additional semantics for some SPIR-V instructions, and additional validation rules a module must adhere to in order to be considered valid.

This document describes the environment for implementations of the OpenCL API. It is written for compiler developers who are generating SPIR-V modules to be consumed by the OpenCL API, for implementors of the OpenCL API who are consuming SPIR-V modules, and by software developers who are using SPIR-V modules with the OpenCL API.

Chapter 2. SPIR-V Consumption

This section describes common properties of all OpenCL environments. Subsequent sections describe environments for specific versions of OpenCL, and how an environment may additionally be modified via OpenCL or SPIR-V extensions.

A SPIR-V module passed to an OpenCL environment is interpreted as a series of 32-bit words in host endianness, with literal strings packed as described in the SPIR-V specification. The first few words of the SPIR-V module must be a magic number and a SPIR-V version number, as described in the SPIR-V specification.

2.1. Validation Rules

The following are a list of validation rules that apply to SPIR-V modules executing in all OpenCL environments:

- The *Execution Model* declared in **OpEntryPoint** must be **Kernel**.
- The *Addressing Model* declared in **OpMemoryModel** must be either **Physical32** or **Physical64**:
  - Modules indicating a **Physical32** *Addressing Model* are valid for OpenCL devices reporting `32` for `CL_DEVICE_ADDRESS_BITS`.
  - Modules indicating a **Physical64** *Addressing Model* are valid for OpenCL devices reporting `64` for `CL_DEVICE_ADDRESS_BITS`.
- The *Memory Model* declared in **OpMemoryModel** must be **OpenCL**.
- For all **OpTypeInt** integer type-declaration instructions:
  - *Signedness* must be 0, indicating no signedness semantics.
- For all **OpTypeImage** type-declaration instructions:
  - *Sampled Type* must be **OpTypeVoid**.
  - *Sampled* must be 0, indicating that the image usage will be known at run time, not at compile time.
  - *MS* must be 0, indicating single-sampled content.
  - *Arrayed* may only be set to 1, indicating arrayed content, when *Dim* is set to **1D** or **2D**.
  - *Image Format* must be **Unknown**, indicating that the image does not have a specified format.
  - The optional image *Access Qualifier* must be present.
- The image write instruction **OpImageWrite** must not include any optional *Image Operands*.
- The image read instructions **OpImageRead**, **OpImageFetch**, and **OpImageSampleExplicitLod** must not include the optional *Image Operand* **ConstOffset**.
- For all **Atomic Instructions**:
  - Only 32-bit integer types are supported for the *Result Type* and/or type of *Value*.
  - The *Pointer* operand must be a pointer to the **Function**, **Workgroup**, or **CrossWorkGroup**

Storage Classes. Note that an **Atomic Instruction** on a pointer to the **Function** *Storage Class* is valid, but does not have defined behavior.

- Recursion is not supported. The static function call graph for an entry point must not contain cycles.

2.2. Source Language Encoding

If a SPIR-V module represents a program written in OpenCL C, then the *Source Language* operand for the **OpSource** instruction should be **OpenCL_C**, and the 32-bit literal language *Version* should describe the version of OpenCL C, encoded MSB to LSB as:

```
0 | Major Number | Minor Number | Revision Number (optional)
```

Hence, OpenCL C 1.2 would be encoded as `0x00010200`, and OpenCL C 2.0 as `0x00020000`.

If a SPIR-V module represents a program written in OpenCL C++, then the *Source Language* operand for the **OpSource** instruction should be **OpenCL_CPP**, and the 32-bit literal language *Version* should describe the version of OpenCL C++, encoded similarly. Hence, OpenCL C++ 2.2 would be encoded as `0x00020200`.

The source language version is purely informational and has no semantic meaning.

2.3. Numerical Type Formats

For all OpenCL environments, floating-point types are represented and stored using IEEE-754 semantics. All integer formats are represented and stored using 2's-complement format.

2.4. Supported Types

The following types are supported by OpenCL environments. Note that some types may require additional capabilities, and may not be supported by all OpenCL environments.

OpenCL environments support arrays declared using **OpTypeArray**, structs declared using **OpTypeStruct**, functions declared using **OpTypeFunction**, and pointers declared using **OpTypePointer**.

2.4.1. Basic Scalar and Vector Types

OpTypeVoid is supported.

The following scalar types are supported by OpenCL environments:

- **OpTypeBool**
- **OpTypeInt**, with *Width* equal to 8, 16, 32, or 64, and with *Signedness* equal to zero, indicating no signedness semantics.
- **OpTypeFloat**, with *Width* equal to 16, 32, or 64.

OpenCL environments support vector types declared using **OpTypeVector**. The vector *Component Type* may be any of the scalar types described above. Supported vector *Component Counts* are 2, 3, 4, 8, or 16.

2.4.2. Image-Related Data Types

The following table describes the **OpTypeImage** image types supported by OpenCL environments:

Table 1. Image Types

Dim	*Depth*	*Arrayed*	**Description**
1D	`0`	`0`	A 1D image.
1D	`0`	`1`	A 1D image array.
2D	`0`	`0`	A 2D image.
2D	`1`	`0`	A 2D depth image.
2D	`0`	`1`	A 2D image array.
2D	`1`	`1`	A 2D depth image array.
3D	`0`	`0`	A 3D image.
Buffer	`0`	`0`	A 1D buffer image.

OpTypeSampler may be used to declare sampler types in OpenCL environments.

2.4.3. Other Data Types

The following table describes other data types that may be used in an OpenCL environment:

Table 2. Other Data Types

Type	**Description**
OpTypeEvent	OpenCL event type.
OpTypeDeviceEvent	OpenCL device-side event type.
OpTypePipe	OpenCL pipe type
OpTypeReserveId	OpenCL pipe reservation identifier.
OpTypeQueue	OpenCL device-side command queue.

2.5. Image Channel Order Mapping

The following table describes how the results of the SPIR-V **OpImageQueryOrder** instruction correspond to the OpenCL host API image channel orders.

Table 3. Image Channel Order mapping

SPIR-V Image Channel Order	**OpenCL Image Channel Order**
`R`	`CL_R`
`A`	`CL_A`
`RG`	`CL_RG`

SPIR-V Image Channel Order	OpenCL Image Channel Order
RA	CL_RA
RGB	CL_RGB
RGBA	CL_RGBA
BGRA	CL_BGRA
ARGB	CL_ARGB
Intensity	CL_INTENSITY
Luminance	CL_LUMINANCE
Rx	CL_Rx
RGx	CL_RGx
RGBx	CL_RGBx
Depth	CL_DEPTH
DepthStencil	CL_DEPTH_STENCIL
sRGB	CL_sRGB
sRGBA	CL_sRGBA
sBGRA	CL_sBGRA
sRGBx	CL_sRGBx

2.6. Image Channel Data Type Mapping

The following table describes how the results of the SPIR-V **OpImageQueryFormat** instruction correspond to the OpenCL host API image channel data types.

Table 4. Image Channel Data Type mapping

SPIR-V Image Channel Data Type	OpenCL Image Channel Data Type
SnormInt8	CL_SNORM_INT8
SnormInt16	CL_SNORM_INT16
UnormInt8	CL_UNORM_INT8
UnormInt16	CL_UNORM_INT16
UnormInt24	CL_UNORM_INT24
UnormShort565	CL_UNORM_SHORT_565
UnormShort555	CL_UNORM_SHORT_555
UnormInt101010	CL_UNORM_INT_101010
SignedInt8	CL_SIGNED_INT8
SignedInt16	CL_SIGNED_INT16
SignedInt32	CL_SIGNED_INT32
UnsignedInt8	CL_UNSIGNED_INT8
UnsignedInt16	CL_UNSIGNED_INT16
UnsignedInt32	CL_UNSIGNED_INT32
HalfFloat	CL_HALF_FLOAT
Float	CL_FLOAT

2.7. Kernels

An **OpFunction** in a SPIR-V module that is identified with **OpEntryPoint** defines an OpenCL kernel that may be invoked using the OpenCL host API enqueue kernel interfaces.

2.8. Kernel Return Types

The *Result Type* for an **OpFunction** identified with **OpEntryPoint** must be **OpTypeVoid**.

2.9. Kernel Arguments

An **OpFunctionParameter** for an **OpFunction** that is identified with **OpEntryPoint** defines an OpenCL kernel argument. Allowed types for OpenCL kernel arguments are:

- **OpTypeInt**
- **OpTypeFloat**
- **OpTypeStruct**
- **OpTypeVector**
- **OpTypePointer**
- **OpTypeSampler**
- **OpTypeImage**
- **OpTypePipe**
- **OpTypeQueue**

For **OpTypeInt** parameters, supported *Widths* are 8, 16, 32, and 64, and must have no signedness semantics.

For **OpTypeFloat** parameters, *Width* must be 32.

For **OpTypeStruct** parameters, supported structure *Member Types* are:

- **OpTypeInt**
- **OpTypeFloat**
- **OpTypeStruct**
- **OpTypeVector**
- **OpTypePointer**

For **OpTypePointer** parameters, supported *Storage Classes* are:

- **CrossWorkgroup**
- **Workgroup**
- **UniformConstant**

OpenCL kernel argument types must have a representation in the OpenCL host API.

Environments that support extensions or optional features may allow additional types in an entry point's parameter list.

Chapter 3. OpenCL 2.2

An OpenCL 2.2 environment must accept SPIR-V 1.0, 1.1, and 1.2 modules.

3.1. Full Profile

An OpenCL 2.2 Full Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**
- **DeviceEnqueue**
- **Float16Buffer**
- **GenericPointer**
- **Groups**
- **Int64**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**
- **SubgroupDispatch**
- **PipeStorage**

The following capabilities may be optionally supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **ImageReadWrite**
- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

3.2. Embedded Profile

An OpenCL 2.2 Embedded Profile environment is guaranteed to support the following SPIR-V

capabilities:

- **Address**
- **DeviceEnqueue**
- **Float16Buffer**
- **GenericPointer**
- **Groups**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**
- **SubgroupDispatch**
- **PipeStorage**

Furthermore, the following capabilities may optionally be supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **ImageReadWrite**
- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

3.3. Validation Rules

The following are a list of validation rules for SPIR-V modules executing in an OpenCL 2.2 environment:

Scope for *Execution* is generally limited to:

- **Workgroup**
- **Subgroup**

Scope for *Memory* is generally limited to:

- **CrossDevice**
- **Device**
- **Workgroup**
- **Invocation**

Scope for *Execution* for the **OpGroupAsyncCopy** and **OpGroupWaitEvents** instructions is specifically limited to:

- **Workgroup**

The *Pointer* operand to all **Atomic Instructions** may additionally be a pointer to the **Generic** *Storage Class*, however behavior is still undefined if the **Generic** pointer represents a pointer to the **Function** *Storage Class*.

Chapter 4. OpenCL 2.1

An OpenCL 2.1 environment must accept SPIR-V 1.0 modules.

4.1. Full Profile

An OpenCL 2.1 Full Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**
- **DeviceEnqueue**
- **Float16Buffer**
- **GenericPointer**
- **Groups**
- **Int64**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**

The following capabilities may be optionally supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **ImageReadWrite**
- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

4.2. Embedded Profile

An OpenCL 2.1 Embedded Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**

- **DeviceEnqueue**
- **Float16Buffer**
- **GenericPointer**
- **Groups**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**

Furthermore, the following capabilities may optionally be supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **ImageReadWrite**
- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

4.3. Validation Rules

The following are a list of validation rules for SPIR-V modules executing in an OpenCL 2.1 environment:

Scope for *Execution* is generally limited to:

- **Workgroup**
- **Subgroup**

Scope for *Memory* is generally limited to:

- **CrossDevice**
- **Device**
- **Workgroup**
- **Invocation**

Scope for *Execution* for the **OpGroupAsyncCopy** and **OpGroupWaitEvents** instructions is specifically limited to:

- **Workgroup**

The *Pointer* operand to all **Atomic Instructions** may additionally be a pointer to the **Generic** *Storage Class*, however behavior is still undefined if the **Generic** pointer represents a pointer to the **Function** *Storage Class*.

Chapter 5. OpenCL 2.0

An OpenCL 2.0 environment must accept SPIR-V 1.0 modules if it includes the optional extension `cl_khr_il_program` in the host API `CL_PLATFORM_EXTENSIONS` or `CL_DEVICE_EXTENSIONS` string.

5.1. Full Profile

An OpenCL 2.0 Full Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**
- **DeviceEnqueue**
- **Float16Buffer**
- **GenericPointer**
- **Groups**
- **Int64**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**

The following capabilities may be optionally supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **ImageReadWrite**
- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

5.2. Embedded Profile

An OpenCL 2.0 Embedded Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**
- **DeviceEnqueue**
- **Float16Buffer**
- **GenericPointer**
- **Groups**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**

Furthermore, the following capabilities may optionally be supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **ImageReadWrite**
- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

5.3. Validation Rules

The following are a list of validation rules for SPIR-V modules executing in an OpenCL 2.0 environment:

Scope for *Execution* is generally limited to:

- **Workgroup**

Scope for *Memory* is generally limited to:

- **CrossDevice**
- **Device**
- **Workgroup**
- **Invocation**

The *Pointer* operand to all **Atomic Instructions** may additionally be a pointer to the **Generic** *Storage Class*, however behavior is still undefined if the **Generic** pointer represents a pointer to the **Function** *Storage Class*.

Chapter 6. OpenCL 1.2

An OpenCL 1.2 environment must accept SPIR-V 1.0 modules if it includes the optional extension `cl_khr_il_program` in the host API `CL_PLATFORM_EXTENSIONS` or `CL_DEVICE_EXTENSIONS` string.

6.1. Full Profile

An OpenCL 1.2 Full Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**
- **Float16Buffer**
- **Groups**
- **Int64**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Vector16**

The following capabilities may be optionally supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

6.2. Embedded Profile

An OpenCL 1.2 Embedded Profile environment is guaranteed to support the following SPIR-V capabilities:

- **Address**
- **Float16Buffer**
- **Groups**
- **Int16**

- **Int8**
- **Kernel**
- **Linkage**
- **Pipes**
- **Vector16**

The following capabilities may be optionally supported:

- **ImageBasic**, if `CL_DEVICE_IMAGE_SUPPORT` is `CL_TRUE`
- **Float64**, if the device supports double precision floating-point

If **ImageBasic** is supported then the following capabilities must also be supported:

- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**

6.3. Validation Rules

The following are a list of validation rules for SPIR-V modules executing in an OpenCL 1.2 environment:

Scope for *Execution* is generally limited to:

- **Workgroup**

Scope for *Memory* is generally limited to:

- **CrossDevice**
- **Device**
- **Workgroup**
- **Invocation**

The following **Group Instructions** are not supported:

- **OpGroupAll**
- **OpGroupAny**
- **OpGroupBroadcast**
- **OpGroupIAdd**
- **OpGroupFAdd**
- **OpGroupFMin**

- **OpGroupUMin**
- **OpGroupSMin**
- **OpGroupFMax**
- **OpGroupUMax**
- **OpGroupSMax**

For the **Barrier Instructions OpControlBarrier** and **OpMemoryBarrier**, the *Scope* for execution must be **Workgroup**, the *Scope* for memory must be **Workgroup**, and the *Memory Semantics* must be **SequentiallyConsistent**.

For the **Atomic Instructions**, the *Scope* must be **Device**, and the *Memory Semantics* must be **Relaxed**.

Chapter 7. OpenCL Extensions

An OpenCL environment may be modified by OpenCL extensions. For example, some OpenCL extensions may require support for support for additional SPIR-V capabilities or instructions, or by relaxing SPIR-V restrictions. Some OpenCL extensions may modify the OpenCL environment by requiring consumption of a SPIR-V module that uses a SPIR-V extension. In this case, the implementation will include the OpenCL extension in the host API `CL_PLATFORM_EXTENSIONS` or `CL_DEVICE_EXTENSIONS` string, but not the corresponding SPIR-V extension.

This section describes how the OpenCL environment is modified by Khronos (`khr`) OpenCL extensions. Other OpenCL extensions, such as multi-vendor (`ext`) extensions or vendor-specific extensions, describe how they modify the OpenCL environment in their individual extension specifications. The Khronos OpenCL extensions require no corresponding SPIR-V extensions.

7.1. Declaring SPIR-V Extensions

A SPIR-V module declares use of a SPIR-V extension using **OpExtension** and the name of the SPIR-V extension. For example:

```
OpExtension "SPV_KHR_extension_name"
```

Only use of SPIR-V extensions may be declared in a SPIR-V module using **OpExtension**; there is never a need to declare use of an OpenCL extension in a SPIR-V module using **OpExtension**.

7.2. Full and Embedded Profile Extensions

7.2.1. `cl_khr_3d_image_writes`

If the OpenCL environment supports the extension `cl_khr_3d_image_writes`, then the environment must accept *Image* operands to **OpImageWrite** that are declared with with dimensionality *Dim* equal to **3D**.

7.2.2. `cl_khr_depth_images`

If the OpenCL environment supports the extension `cl_khr_depth_images`, then the environment must accept modules that declare 2D depth image types using **OpTypeImage** with dimensionality *Dim* equal to **2D** and *Depth* equal to 1, indicating a depth image. 2D depth images may optionally be *Arrayed*, if supported.

Additionally, the following Image Channel Orders may be returned by **OpImageQueryOrder**:

- **Depth**

7.2.3. `cl_khr_device_enqueue_local_arg_types`

If the OpenCL environment supports the extension `cl_khr_device_enqueue_local_arg_types`, then

then environment will allow *Invoke* functions to be passed to **OpEnqueueKernel** with **Workgroup** memory pointer parameters of any type.

7.2.4. `cl_khr_fp16`

If the OpenCL environment supports the extension `cl_khr_fp16`, then the environment must accept modules that declare the following SPIR-V capabilities:

- **Float16**

7.2.5. `cl_khr_fp64`

If the OpenCL environment supports the extension `cl_khr_fp64`, then the environment must accept modules that declare the following SPIR-V capabilities:

- **Float64**

7.2.6. `cl_khr_gl_depth_images`

If the OpenCL environment supports the extension `cl_khr_gl_depth_images`, then the following Image Channel Orders may additionally be returned by **OpImageQueryOrder**:

- **DepthStencil**

Also, the following Image Channel Data Types may additionally be returned by **OpImageQueryFormat**:

- **UnormInt24**

7.2.7. `cl_khr_gl_msaa_sharing`

If the OpenCL environment supports the extension `cl_khr_gl_msaa_sharing`, then the environment must accept modules that declare 2D multi-sampled image types using **OpTypeImage** with dimensionality *Dim* equal to **2D** and *MS* equal to 1, indicating multi-sampled content. 2D multi-sampled images may optionally be *Arrayed* or *Depth* images, if supported.

The 2D multi-sampled images may be used with the following instructions:

- **OpImageRead**
- **OpImageQuerySizeLod**
- **OpImageQueryFormat**
- **OpImageQueryOrder**
- **OpImageQuerySamples**

7.2.8. `cl_khr_int64_base_atomics` and `cl_khr_int64_extended_atomics`

If the OpenCL environment supports the extension `cl_khr_int64_base_atomics` or `cl_khr_int64_extended_atomics`, then the environment must support 64-bit integer operands for all

of the SPIR-V **Atomic Instructions**.

When the **WorkgroupMemory** *Memory Semantic* is used the *Scope* must be **Workgroup**.

Note: OpenCL environments that consume SPIR-V must support both `cl_khr_int64_base_atomics` and `cl_khr_int64_extended_atomics` or neither of these extensions.

7.2.9. `cl_khr_mipmap_image`

If the OpenCL environment supports the extension `cl_khr_mipmap_image`, then the environment must accept non-zero optional **Lod** *Image Operands* for the following instructions:

- **OpImageSampleExplicitLod**
- **OpImageFetch**
- **OpImageRead**
- **OpImageQuerySizeLod**

Note: Implementations that support `cl_khr_mipmap_image` are not guaranteed to support the **ImageMipmap** capability, since this extension does not require non-zero optional **Lod** *Image Operands* for **OpImageWrite**.

7.2.10. `cl_khr_mipmap_image_writes`

If the OpenCL environment supports the extension `cl_khr_mipmap_image_writes`, then the environment must accept non-zero optional **Lod** *Image Operands* for the following instructions:

- **OpImageWrite**

Note: An implementation that supports `cl_khr_mipmap_image_writes` must also support `cl_khr_mipmap_image`, and support for both extensions does guarantee support for the **ImageMipmap** capability.

7.2.11. `cl_khr_subgroups`

If the OpenCL environment supports the extension `cl_khr_subgroups`, then the environment will generally allows the scope for *Execution* to include:

- **Subgroup**

However, the *Scope* for *Execution* for the **OpGroupAsyncCopy** and **OpGroupWaitEvents** instructions still is limited to:

- **Workgroup**

7.2.12. `cl_khr_subgroup_named_barrier`

If the OpenCL environment supports the extension `cl_khr_subgroup_named_barrier`, then the environment must accept modules that declare the following SPIR-V capabilities:

- **NamedBarrier**

7.3. Embedded Profile Extensions

7.3.1. `cles_khr_int64`

If the OpenCL environment supports the extension `cles_khr_int64`, then the environment must accept modules that declare the following SPIR-V capabilities:

- **Int64**

Chapter 8. OpenCL Numerical Compliance

This section describes features of the C++14 and IEEE-754 standards that must be supported by all OpenCL compliant devices.

This section describes the functionality that must be supported by all OpenCL devices for single precision floating-point numbers. Currently, only single precision floating-point is a requirement. Half precision floating-point is an optional feature indicated by the **Float16** capability. Double precision floating-point is also an optional feature indicated by the **Float64** capability.

8.1. Rounding Modes

Floating-point calculations may be carried out internally with extra precision and then rounded to fit into the destination type. IEEE 754 defines four possible rounding modes:

- *Round to nearest even*
- *Round toward +infinity*
- *Round toward -infinity*
- *Round toward zero*

The complete set of rounding modes supported by the device are described by the `CL_DEVICE_SINGLE_FP_CONFIG`, `CL_DEVICE_HALF_FP_CONFIG`, and `CL_DEVICE_DOUBLE_FP_CONFIG` device queries.

For double precision operations, *Round to nearest even* is a required rounding mode, and is therefore the default rounding mode for double precision operations.

For single precision operations, devices supporting the full profile must support *Round to nearest even*, therefore for full profile devices this is the default rounding mode for single precision operations. Devices supporting the embedded profile may support either *Round to nearest even* or *Round toward zero* as the default rounding mode for single precision operations.

For half precision operations, devices may support either *Round to nearest even* or *Round toward zero* as the default rounding mode for half precision operations.

Only static selection of rounding mode is supported. Dynamically reconfiguring the rounding mode as specified by the IEEE 754 spec is not supported.

8.2. Rounding Modes for Conversions

Results of the following conversion instructions may include an optional **FPRoundingMode** decoration:

- **OpConvertFToU**
- **OpConvertFToS**
- **OpConvertSToF**

- **OpConvertUToF**
- **OpFConvert**

The **FPRoundingMode** decoration may not be added to results of any other instruction.

If no rounding mode is specified explicitly via an **FPRoundingMode** decoration, then the default rounding mode for conversion operations is:

- *Round to nearest even*, for conversions to floating-point types.
- *Round toward zero*, for conversions from floating-point to integer types.

8.3. Out-of-Range Conversions

When a conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out-of-range.

Converting an out-of-range integer to an integer type without a **SaturatedConversion** decoration follows C99/C++14 conversion rules.

Converting an out-of-range floating point number to an integer type without a **SaturatedConversion** decoration is implementation-defined.

8.4. INF, NaN, and Denormalized Numbers

INFs and NaNs must be supported. Support for signaling NaNs is not required.

Support for denormalized numbers with single precision and half precision floating-point is optional. Denormalized single precision or half precision floating-point numbers passed as the input or produced as the output of single precision or half precision floating-point operations may be flushed to zero. Support for denormalized numbers is required for double precision floating-point.

Support for INFs, NaNs, and denormalized numbers is described by the `CL_FP_DENORM` and `CL_FP_INF_NAN` bits in the `CL_DEVICE_SINGLE_FP_CONFIG`, `CL_DEVICE_HALF_FP_CONFIG`, and `CL_DEVICE_DOUBLE_FP_CONFIG` device queries.

8.5. Floating-Point Exceptions

Floating-point exceptions are disabled in OpenCL. The result of a floating-point exception must match the IEEE 754 spec for the exceptions-not-enabled case. Whether and when the implementation sets floating-point flags or raises floating-point exceptions is implementation-defined.

This standard provides no method for querying, clearing or setting floating-point flags or trapping raised exceptions. Due to non-performance, non-portability of trap mechanisms, and the impracticality of servicing precise exceptions in a vector context (especially on heterogeneous hardware), such features are discouraged.

Implementations that nevertheless support such operations through an extension to the standard shall initialize with all exception flags cleared and the exception masks set so that exceptions raised by arithmetic operations do not trigger a trap to be taken. If the underlying work is reused by the implementation, the implementation is however not responsible for re-clearing the flags or resetting exception masks to default values before entering the kernel. That is to say that kernels that do not inspect flags or enable traps are licensed to expect that their arithmetic will not trigger a trap. Those kernels that do examine flags or enable traps are responsible for clearing flag state and disabling all traps before returning control to the implementation. Whether or when the underlying work-item (and accompanying global floating-point state if any) is reused is implementation-defined.

8.6. Relative Error as ULPs

In this section we discuss the maximum relative error defined as ulp (units in the last place). Addition, subtraction, multiplication, fused multiply-add, and conversion between integer and a single precision floating-point format are IEEE 754 compliant and are therefore correctly rounded. Conversion between floating-point formats and explicit conversions must be correctly rounded.

The ULP is defined as follows:

> If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then ulp(x) = |b - a|, otherwise ulp(x) is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, ulp(NaN) is NaN.

Attribution: This definition was taken with consent from Jean-Michel Muller with slight clarification for behavior at zero. Refer to: On the definition of ulp(x).

0 ULP is used for math functions that do not require rounding. The reference value used to compute the ULP value is the infinitely precise result.

8.6.1. ULP Values for Math Instructions - Full Profile

The ULP Values for Math Instructions table below describes the minimum accuracy of floating-point math arithmetic instructions for full profile devices given as ULP values.

Table 5. ULP Values for Math Instructions - Full Profile

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpFAdd	Correctly rounded	Correctly rounded	Correctly rounded
OpFSub	Correctly rounded	Correctly rounded	Correctly rounded
OpFMul	Correctly rounded	Correctly rounded	Correctly rounded
OpFDiv	Correctly rounded	<= 2.5 ulp	Correctly rounded
OpExtInst acos	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst acosh	<= 4 ulp	<= 4 ulp	<= 2 ulp

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst acospi	<= 5 ulp	<= 5 ulp	<= 2 ulp
OpExtInst asin	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst asinh	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst asinpi	<= 5 ulp	<= 5 ulp	<= 2 ulp
OpExtInst atan	<= 5 ulp	<= 5 ulp	<= 2 ulp
OpExtInst atanh	<= 5 ulp	<= 5 ulp	<= 2 ulp
OpExtInst atanpi	<= 5 ulp	<= 5 ulp	<= 2 ulp
OpExtInst atan2	<= 6 ulp	<= 6 ulp	<= 2 ulp
OpExtInst atan2pi	<= 6 ulp	<= 6 ulp	<= 2 ulp
OpExtInst cbrt	<= 2 ulp	<= 2 ulp	<= 2 ulp
OpExtInst ceil	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst copysign	0 ulp	0 ulp	0 ulp
OpExtInst cos	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst cosh	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst cospi	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst erfc	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst erf	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst exp	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst exp2	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst exp10	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst expm1	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst fabs	0 ulp	0 ulp	0 ulp
OpExtInst fdim	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst floor	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst fma	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst fmax	0 ulp	0 ulp	0 ulp
OpExtInst fmin	0 ulp	0 ulp	0 ulp
OpExtInst fmod	0 ulp	0 ulp	0 ulp
OpExtInst fract	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst frexp	0 ulp	0 ulp	0 ulp
OpExtInst hypot	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst ilogb	0 ulp	0 ulp	0 ulp
OpExtInst ldexp	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst lgamma	Implementation-defined	Implementation-defined	Implementation-defined

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst lgamma_r	Implementation-defined	Implementation-defined	Implementation-defined
OpExtInst log	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst log2	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst log10	<= 3 ulp	<= 3 ulp	<= 2 ulp
OpExtInst log1p	<= 2 ulp	<= 2 ulp	<= 2 ulp
OpExtInst logb	0 ulp	0 ulp	0 ulp
OpExtInst mad	Implemented either as a correctly rounded fma, or as a multiply followed by an add, both of which are correctly rounded	Implemented either as a correctly rounded fma, or as a multiply followed by an add, both of which are correctly rounded	Implemented either as a correctly rounded fma, or as a multiply followed by an add, both of which are correctly rounded
OpExtInst maxmag	0 ulp	0 ulp	0 ulp
OpExtInst minmag	0 ulp	0 ulp	0 ulp
OpExtInst modf	0 ulp	0 ulp	0 ulp
OpExtInst nan	0 ulp	0 ulp	0 ulp
OpExtInst nextafter	0 ulp	0 ulp	0 ulp
OpExtInst pow	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst pown	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst powr	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst remainder	0 ulp	0 ulp	0 ulp
OpExtInst remquo	0 ulp for the remainder, at least the lower 7 bits of the integral quotient	0 ulp for the remainder, at least the lower 7 bits of the integral quotient	0 ulp for the remainder, at least the lower 7 bits of the integral quotient
OpExtInst rint	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst rootn	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst round	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst rsqrt	<= 2 ulp	<= 2 ulp	<= 1 ulp
OpExtInst sin	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst sincos	<= 4 ulp for sine and cosine values	<= 4 ulp for sine and cosine values	<= 2 ulp for sine and cosine values
OpExtInst sinh	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst sinpi	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst sqrt	Correctly rounded	<= 3 ulp	Correctly rounded
OpExtInst tan	<= 5 ulp	<= 5 ulp	<= 2 ulp
OpExtInst tanh	<= 5 ulp	<= 5 ulp	<= 2 ulp

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst tanpi	<= 6 ulp	<= 6 ulp	<= 2 ulp
OpExtInst tgamma	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst trunc	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst half_cos		<= 8192 ulp	
OpExtInst half_divide		<= 8192 ulp	
OpExtInst half_exp		<= 8192 ulp	
OpExtInst half_exp2		<= 8192 ulp	
OpExtInst half_exp10		<= 8192 ulp	
OpExtInst half_log		<= 8192 ulp	
OpExtInst half_log2		<= 8192 ulp	
OpExtInst half_log10		<= 8192 ulp	
OpExtInst half_powr		<= 8192 ulp	
OpExtInst half_recip		<= 8192 ulp	
OpExtInst half_rsqrt		<= 8192 ulp	
OpExtInst half_sin		<= 8192 ulp	
OpExtInst half_sqrt		<= 8192 ulp	
OpExtInst half_tan		<= 8192 ulp	
OpExtInst native_cos		Implementation-defined	
OpExtInst native_divide		Implementation-defined	
OpExtInst native_exp		Implementation-defined	
OpExtInst native_exp2		Implementation-defined	
OpExtInst native_exp10		Implementation-defined	
OpExtInst native_log		Implementation-defined	
OpExtInst native_log2		Implementation-defined	
OpExtInst native_log10		Implementation-defined	
OpExtInst native_powr		Implementation-defined	
OpExtInst native_recip		Implementation-defined	

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst native_rsqrt		Implementation-defined	
OpExtInst native_sin		Implementation-defined	
OpExtInst native_sqrt		Implementation-defined	
OpExtInst native_tan		Implementation-defined	

8.6.2. ULP Values for Math Instructions - Embedded Profile

The ULP Values for Math instructions for Embedded Profile table below describes the minimum accuracy of floating-point math arithmetic operations given as ULP values for the embedded profile.

Table 6. ULP Values for Math Instructions - Embedded Profile

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpFAdd	Correctly rounded	Correctly rounded	Correctly rounded
OpFSub	Correctly rounded	Correctly rounded	Correctly rounded
OpFMul	Correctly rounded	Correctly rounded	Correctly rounded
OpFDiv	<= 3 ulp	<= 3 ulp	<= 1 ulp
OpExtInst acos	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst acosh	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst acospi	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst asin	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst asinh	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst asinpi	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst atan	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst atanh	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst atanpi	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst atan2	<= 6 ulp	<= 6 ulp	<= 3 ulp
OpExtInst atan2pi	<= 6 ulp	<= 6 ulp	<= 3 ulp
OpExtInst cbrt	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst ceil	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst copysign	0 ulp	0 ulp	0 ulp
OpExtInst cos	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst cosh	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst cospi	<= 4 ulp	<= 4 ulp	<= 2 ulp

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst erfc	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst erf	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst exp	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst exp2	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst exp10	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst expm1	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst fabs	0 ulp	0 ulp	0 ulp
OpExtInst fdim	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst floor	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst fma	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst fmax	0 ulp	0 ulp	0 ulp
OpExtInst fmin	0 ulp	0 ulp	0 ulp
OpExtInst fmod	0 ulp	0 ulp	0 ulp
OpExtInst fract	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst frexp	0 ulp	0 ulp	0 ulp
OpExtInst hypot	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst ilogb	0 ulp	0 ulp	0 ulp
OpExtInst ldexp	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst lgamma	Implementation-defined	Implementation-defined	Implementation-defined
OpExtInst lgamma_r	Implementation-defined	Implementation-defined	Implementation-defined
OpExtInst log	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst log2	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst log10	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst log1p	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst logb	0 ulp	0 ulp	0 ulp
OpExtInst mad	Implemention-defined	Implemention-defined	Implemention-defined
OpExtInst maxmag	0 ulp	0 ulp	0 ulp
OpExtInst minmag	0 ulp	0 ulp	0 ulp
OpExtInst modf	0 ulp	0 ulp	0 ulp
OpExtInst nan	0 ulp	0 ulp	0 ulp
OpExtInst nextafter	0 ulp	0 ulp	0 ulp
OpExtInst pow	<= 16 ulp	<= 16 ulp	<= 5 ulp
OpExtInst pown	<= 16 ulp	<= 16 ulp	<= 5 ulp

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst powr	<= 16 ulp	<= 16 ulp	<= 5 ulp
OpExtInst remainder	0 ulp	0 ulp	0 ulp
OpExtInst remquo	0 ulp for the remainder, at least the lower 7 bits of the integral quotient	0 ulp for the remainder, at least the lower 7 bits of the integral quotient	0 ulp for the remainder, at least the lower 7 bits of the integral quotient
OpExtInst rint	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst rootn	<= 16 ulp	<= 16 ulp	<= 5 ulp
OpExtInst round	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst rsqrt	<= 4 ulp	<= 4 ulp	<= 1 ulp
OpExtInst sin	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst sincos	<= 4 ulp for sine and cosine values	<= 4 ulp for sine and cosine values	<= 2 ulp for sine and cosine values
OpExtInst sinh	<= 4 ulp	<= 4 ulp	<= 3 ulp
OpExtInst sinpi	<= 4 ulp	<= 4 ulp	<= 2 ulp
OpExtInst sqrt	<= 4 ulp	<= 4 ulp	<= 1 ulp
OpExtInst tan	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst tanh	<= 5 ulp	<= 5 ulp	<= 3 ulp
OpExtInst tanpi	<= 6 ulp	<= 6 ulp	<= 3 ulp
OpExtInst tgamma	<= 16 ulp	<= 16 ulp	<= 4 ulp
OpExtInst trunc	Correctly rounded	Correctly rounded	Correctly rounded
OpExtInst half_cos		<= 8192 ulp	
OpExtInst half_divide		<= 8192 ulp	
OpExtInst half_exp		<= 8192 ulp	
OpExtInst half_exp2		<= 8192 ulp	
OpExtInst half_exp10		<= 8192 ulp	
OpExtInst half_log		<= 8192 ulp	
OpExtInst half_log2		<= 8192 ulp	
OpExtInst half_log10		<= 8192 ulp	
OpExtInst half_powr		<= 8192 ulp	
OpExtInst half_recip		<= 8192 ulp	
OpExtInst half_rsqrt		<= 8192 ulp	
OpExtInst half_sin		<= 8192 ulp	
OpExtInst half_sqrt		<= 8192 ulp	
OpExtInst half_tan		<= 8192 ulp	
OpExtInst native_cos		Implementation-defined	

SPIR-V Instruction	Minimum Accuracy - Float64	Minimum Accuracy - Float32	Minimum Accuracy - Float16
OpExtInst native_divide		Implementation-defined	
OpExtInst native_exp		Implementation-defined	
OpExtInst native_exp2		Implementation-defined	
OpExtInst native_exp10		Implementation-defined	
OpExtInst native_log		Implementation-defined	
OpExtInst native_log2		Implementation-defined	
OpExtInst native_log10		Implementation-defined	
OpExtInst native_powr		Implementation-defined	
OpExtInst native_recip		Implementation-defined	
OpExtInst native_rsqrt		Implementation-defined	
OpExtInst native_sin		Implementation-defined	
OpExtInst native_sqrt		Implementation-defined	
OpExtInst native_tan		Implementation-defined	

8.6.3. ULP Values for Math Instructions - Unsafe Math Optimizations Enabled

The ULP Values for Math Instructions with Unsafe Math Optimizations table below describes the minimum accuracy of commonly used single precision floating-point math arithmetic instructions given as ULP values if the *-cl-unsafe-math-optimizations* compiler option is specified when compiling or building the OpenCL program.

For derived implementations, the operations used in the derivation may themselves be relaxed according to the ULP Values for Math Instructions with Unsafe Math Optimizations table.

The minimum accuracy of math functions not defined in the ULP Values for Math Instructions with Unsafe Math Optimizations table when the *-cl-unsafe-math-optimizations* compiler option is specified is as defined in the ULP Values for Math Instructions for Full Profile table when operating in the full profile, and as defined in the ULP Values for Math instructions for Embedded Profile table when operating in the embedded profile.

Table 7. ULP Values for Single Precision Math Instructions with -cl-unsafe-math-optimizations

SPIR-V Instruction	Minimum Accuracy
OpFDiv for 1.0 / *x*	<= 2.5 ulp for x in the domain of 2^{-126} to 2^{126} for the full profile, and <= 3 ulp for the embedded profile.
OpFDiv for *x* / *y*	<= 2.5 ulp for x in the domain of 2^{-62} to 2^{62} and *y* in the domain of 2^{-62} to 2^{62} for the full profile, and <= 3 ulp for the embedded profile.
OpExtInst acos	<= 4096 ulp
OpExtInst acosh	Implemented as log(x + sqrt(x*x - 1)).
OpExtInst acospi	Implemented as acos(x) * M_PI_F. For non-derived implementations, the error is <= 8192 ulp.
OpExtInst asin	<= 4096 ulp
OpExtInst asinh	Implemented as log(x + sqrt(x*x + 1)).
OpExtInst asinpi	Implemented as asin(x) * M_PI_F. For non-derived implementations, the error is <= 8192 ulp.
OpExtInst atan	<= 4096 ulp
OpExtInst atanh	Defined for x in the domain (-1, 1). For x in [-2^{-10}, 2^{-10}], implemented as x. For x outside of [-2^{-10}, 2^{-10}], implemented as 0.5f * log((1.0f + x) / (1.0f - x)). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst atanpi	Implemented as atan(x) * M_1_PI_F. For non-derived implementations, the error is <= 8192 ulp.
OpExtInst atan2	Implemented as atan(y/x) for x > 0, atan(y/x) + M_PI_F for x < 0 and y > 0, and atan(y/x) - M_PI_F for x < 0 and y < 0.
OpExtInst atan2pi	Implemented as atan2(y, x) * M_1_PI_F. For non-derived implementations, the error is <= 8192 ulp.
OpExtInst cbrt	Implemented as rootn(x, 3). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst cos	For x in the domain [-π, π], the maximum absolute error is <= 2^{-11} and larger otherwise.
OpExtInst cosh	Defined for x in the domain [-∞, ∞] and implemented as 0.5f * (exp(x) + exp(-x)). For non-derived implementations, the error is <= 8192 ULP.
OpExtInst cospi	For x in the domain [-1, 1], the maximum absolute error is <= 2^{-11} and larger otherwise.
OpExtInst exp	<= 3 + floor(fabs(2 * x)) ulp for the full profile, and <= 4 ulp for the embedded profile.
OpExtInst exp2	<= 3 + floor(fabs(2 * x)) ulp for the full profile, and <= 4 ulp for the embedded profile.
OpExtInst exp10	Derived implementations implement this as exp2(x * log2(10)). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst expm1	Derived implementations implement this as exp(x) - 1. For non-derived implementations, the error is <= 8192 ulp.

SPIR-V Instruction	Minimum Accuracy
OpExtInst log	For x in the domain [0.5, 2] the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <=3 ulp for the full profile and <= 4 ulp for the embedded profile
OpExtInst log2	For x in the domain [0.5, 2] the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <=3 ulp for the full profile and <= 4 ulp for the embedded profile
OpExtInst log10	For x in the domain [0.5, 2] the maximum absolute error is <= 2^{-21}; otherwise the maximum error is <=3 ulp for the full profile and <= 4 ulp for the embedded profile
OpExtInst log1p	Derived implementations implement this as log(x + 1). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst pow	Undefined for x = 0 and y = 0. Undefined for x < 0 and non-integer y. Undefined for x < 0 and y outside the domain [-2^{24}, 2^{24}]. For x > 0 or x < 0 and even y, derived implementations implement this as exp2(y * log2(x)). For x < 0 and odd y, derived implementations implement this as -exp2(y * log2(fabs(x)). For x == 0 and nonzero y, derived implementations return zero. For non-derived implementations, the error is <= 8192 ULP. On some implementations, powr() or pown() may perform faster than pow(). If x is known to be >= 0, consider using powr() in place of pow(), or if y is known to be an integer, consider using pown() in place of pow().
OpExtInst pown	Defined only for integer values of y. Undefined for x = 0 and y = 0. For x >= 0 or x < 0 and even y, derived implementations implement this as exp2(y * log2(x)). For x < 0 and odd y, derived implementations implement this as -exp2(y * log2(fabs(x))). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst powr	Defined only for x >= 0. Undefined for x = 0 and y = 0. Derived implementations implement this as exp2(y * log2(x)). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst rootn	Defined for x > 0 when y is non-zero, derived implementations implement this case as exp2(log2(x) / y). Defined for x < 0 when y is odd, derived implementations implement this case as -exp2(log2(-x) / y). Defined for x = +/-0 when y > 0, derived implementations will return +0 in this case. For non-derived implementations, the error is <= 8192 ULP.
OpExtInst sin	For x in the domain [-π, π], the maximum absolute error is <= 2^{-11} and larger otherwise.
OpExtInst sincos	ulp values as defined for sin(x) and cos(x).
OpExtInst sinh	Defined for x in the domain [-∞, ∞]. For x in [-2^{-10}, 2^{-10}], derived implementations implement as x. For x outside of [-2^{-10}, 2^{-10}], derived implement as 0.5f * (exp(x) - exp(-x)). For non-derived implementations, the error is <= 8192 ULP.
OpExtInst sinpi	For x in the domain [-1, 1], the maximum absolute error is <= 2^{-11} and larger otherwise.

SPIR-V Instruction	Minimum Accuracy
OpExtInst tan	Derived implementations implement this as sin(x) * (1.0f / cos(x)). For non-derived implementations, the error is <= 8192 ulp.
OpExtInst tanh	Defined for x in the domain [-∞, ∞]. For x in [-2^{-10}, 2^{-10}], derived implementations implement as x. For x outside of [-2^{-10}, 2^{-10}], derived implementations implement as (exp(x) - exp(-x)) / (exp(x) + exp(-x)). For non-derived implementations, the error is <= 8192 ULP.
OpExtInst tanpi	Derived implementations implement this as tan(x * M_PI_F). For non-derived implementations, the error is <= 8192 ulp for x in the domain [-1, 1].
OpFMul and **OpFAdd**, for *x* * *y* + *z*	Implemented either as a correctly rounded fma or as a multiply and an add both of which are correctly rounded.

8.7. Edge Case Behavior

The edge case behavior of the math functions shall conform to sections F.9 and G.6 of ISO/IEC 9899:TC 2, except where noted below in the *Additional Requirements Beyond ISO/IEC 9899:TC2 section*.

8.7.1. Additional Requirements Beyond ISO/IEC 9899:TC2

Functions that return a NaN with more than one NaN operand shall return one of the NaN operands. Functions that return a NaN operand may silence the NaN if it is a signaling NaN. A non-signaling NaN shall be converted to a non-signaling NaN. A signaling NaN shall be converted to a NaN, and should be converted to a non-signaling NaN. How the rest of the NaN payload bits or the sign of NaN is converted is undefined.

The usual allowances for rounding error (*Relative Error as ULPs section*) or flushing behavior (*Edge Case Behavior in Flush To Zero Mode section*) shall not apply for those values for which *section F.9* of ISO/IEC 9899:,TC2, or *Additional Requirements Beyond ISO/IEC 9899:TC2* and *Edge Case Behavior in Flush To Zero Mode sections* below (and similar sections for other floating-point precisions) prescribe a result (e.g. ceil(-1 < x < 0) returns -0). Those values shall produce exactly the prescribed answers, and no other. Where the ± symbol is used, the sign shall be preserved. For example, sin(±0) = ±0 shall be interpreted to mean sin(+0) is +0 and sin(-0) is -0.

- **OpExtInst acospi**:
  - acospi(1) = +0.
  - acospi(x) returns a NaN for | x | > 1.
- **OpExtInst asinpi**:
  - asinpi(±0) = ±0.
  - asinpi(x) returns a NaN for | x | > 1.
- **OpExtInst atanpi**:
  - atanpi(±0) = ±0.
  - atanpi (±∞) = ±0.5.

- **OpExtInst atan2pi**:
  - atan2pi (±0, -0) = ±1.
  - atan2pi (±0, +0) = ± 0.
  - atan2pi (±0, x) returns ± 1 for x < 0.
  - atan2pi (±0, x) returns ± 0 for x > 0.
  - atan2pi (y, ±0) returns -0.5 for y < 0.
  - atan2pi (y, ±0) returns 0.5 for y > 0.
  - atan2pi (±y, -∞) returns ± 1 for finite y > 0.
  - atan2pi (±y, +∞) returns ± 0 for finite y > 0.
  - atan2pi (±∞, x) returns ± 0.5 for finite x.
  - atan2pi (±∞, -∞) returns ±0.75.
  - atan2pi (±∞, +∞) returns ±0.25.
- **OpExtInst ceil**:
  - ceil(-1 < x < 0) returns -0.
- **OpExtInst cospi**:
  - cospi(±0) returns 1
  - cospi(n + 0.5) is +0 for any integer n where n + 0.5 is representable.
  - cospi(±∞) returns a NaN.
- **OpExtInst exp10**:
  - exp10(±0) returns 1.
  - exp10(-∞) returns +0.
  - exp10(+∞) returns +∞.
- **OpExtInst distance**:
  - distance(x, y) calculates the distance from x to y without overflow or extraordinary precision loss due to underflow.
- **OpExtInst fdim**:
  - fdim(any, NaN) returns NaN.
  - fdim(NaN, any) returns NaN.
- **OpExtInst fmod**:
  - fmod(±0, NaN) returns NaN.
- **OpExtInst fract**:
  - fract(x, iptr) shall not return a value greater than or equal to 1.0, and shall not return a value less than 0.
  - fract(+0, iptr) returns +0 and +0 in iptr.
  - fract(-0, iptr) returns -0 and -0 in iptr.

- fract(+inf, iptr) returns +0 and +inf in iptr.
- fract(-inf, iptr) returns -0 and -inf in iptr.
- fract(NaN, iptr) returns the NaN and NaN in iptr.

- **OpExtInst frexp**:
  - frexp(±∞, exp) returns ±∞ and stores 0 in exp.
  - frexp(NaN, exp) returns the NaN and stores 0 in exp.
- **OpExtInst length**:
  - length calculates the length of a vector without overflow or extraordinary precision loss due to underflow.
- **OpExtInst lgamma_r**:
  - lgamma_r(x, signp) returns 0 in signp if x is zero or a negative integer.
- **OpExtInst nextafter**:
  - nextafter(-0, y > 0) returns smallest positive denormal value.
  - nextafter(+0, y < 0) returns smallest negative denormal value.
- **OpExtInst normalize**:
  - normalize shall reduce the vector to unit length, pointing in the same direction without overflow or extraordinary precision loss due to underflow.
  - normalize(v) returns v if all elements of v are zero.
  - normalize(v) returns a vector full of NaNs if any element is a NaN.
  - normalize(v) for which any element in v is infinite shall proceed as if the elements in v were replaced as follows:

```
for( i = 0; i < sizeof(v) / sizeof(v[0] ); i++ )
    v[i] = isinf(v[i] )  ?  copysign(1.0, v[i]) : 0.0 * v [i];
```

- **OpExtInst pow**:
  - pow(±0, -∞) returns +∞
- **OpExtInst pown**:
  - pown(x, 0) is 1 for any x, even zero, NaN or infinity.
  - pown(±0, n) is ±∞ for odd n < 0.
  - pown(±0, n) is +∞ for even n < 0.
  - pown(±0, n) is +0 for even n > 0.
  - pown(±0, n) is ±0 for odd n > 0.
- **OpExtInst powr**:
  - powr(x, ±0) is 1 for finite x > 0.
  - powr(±0, y) is +∞ for finite y < 0.

- powr(±0, -∞) is +∞.
- powr(±0, y) is +0 for y > 0.
- powr(+1, y) is 1 for finite y.
- powr(x, y) returns NaN for x < 0.
- powr(±0, ±0) returns NaN.
- powr(+∞, ±0) returns NaN.
- powr(+1, ±∞) returns NaN.
- powr(x, NaN) returns the NaN for x >= 0.
- powr(NaN, y) returns the NaN.

- **OpExtInst rint**:
  - rint(-0.5 <= x < 0) returns -0.
- **OpExtInst remquo**:
  - remquo(x, y, &quo) returns a NaN and 0 in quo if x is ±∞, or if y is 0 and the other argument is non-NaN or if either argument is a NaN.
- **OpExtInst rootn**:
  - rootn(±0, n) is ±∞ for odd n < 0.
  - rootn(±0, n) is +∞ for even n < 0.
  - rootn(±0, n) is +0 for even n > 0.
  - rootn(±0, n) is ±0 for odd n > 0.
  - rootn(x, n) returns a NaN for x < 0 and n is even.
  - rootn(x, 0) returns a NaN.
- **OpExtInst round**:
  - round(-0.5 < x < 0) returns -0.
- **OpExtInst sinpi**:
  - sinpi(±0) returns ±0.
  - sinpi(+n) returns +0 for positive integers n.
  - sinpi(-n) returns -0 for negative integers n.
  - sinpi(±∞) returns a NaN.
- **OpExtInst tanpi**:
  - tanpi(±0) returns ±0.
  - tanpi(±∞) returns a NaN.
  - tanpi(n) is copysign(0.0, n) for even integers n.
  - tanpi(n) is copysign(0.0, - n) for odd integers n.
  - tanpi(n + 0.5) for even integer n is +∞ where n + 0.5 is representable.
  - tanpi(n + 0.5) for odd integer n is -∞ where n + 0.5 is representable.

- **OpExtInst trunc**:
  - trunc(-1 < x < 0) returns -0.

8.7.2. Changes to ISO/IEC 9899: TC2 Behavior

OpExtInst modf behaves as though implemented by:

```
gentype modf( gentype value, gentype *iptr )
{
    *iptr = trunc( value );
    return copysign( isinf( value ) ? 0.0 : value - *iptr, value );
}
```

OpExtInst rint always rounds according to round to nearest even rounding mode even if the caller is in some other rounding mode.

8.7.3. Edge Case Behavior in Flush To Zero Mode

If denormals are flushed to zero, then a function may return one of four results:

1. Any conforming result for non-flush-to-zero mode.
2. If the result given by 1 is a sub-normal before rounding, it may be flushed to zero.
3. Any non-flushed conforming result for the function if one or more of its sub-normal operands are flushed to zero.
4. If the result of 3 is a sub-normal before rounding, the result may be flushed to zero.

In each of the above cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

If subnormals are flushed to zero, a device may choose to conform to the following edge cases for **OpExtInst nextafter** instead of those listed in *Additional Requirements Beyond ISO/IEC 9899:TC2 section*:

- nextafter (+smallest normal, y < +smallest normal) = +0.
- nextafter (-smallest normal, y > -smallest normal) = -0.
- nextafter (-0, y > 0) returns smallest positive normal value.
- nextafter (+0, y < 0) returns smallest negative normal value.

For clarity, subnormals or denormals are defined to be the set of representable numbers in the range 0 < x < TYPE_MIN and -TYPE_MIN < x < -0. They do not include ±0. A non-zero number is said to be sub-normal before rounding if, after normalization, its radix-2 exponent is less than (TYPE_MIN_EXP - 1). [1: Here `TYPE_MIN` and `TYPE_MIN_EXP` should be substituted by constants appropriate to the floating-point type under consideration, such as `FLT_MIN` and `FLT_MIN_EXP` for float.]

Chapter 9. Image Addressing and Filtering

This section describes how image operations behave in an OpenCL environment.

9.1. Image Coordinates

Let w_t, h_t and d_t be the width, height (or image array size for a 1D image array) and depth (or image array size for a 2D image array) of the image in pixels. Let `coord.xy` (also referred to as `(s,t)`) or `coord.xyz` (also referred to as `(s,t,r)`) be the coordinates specified to an image read instruction (such as **OpImageRead**) or an image write instruction (such as **OpImageWrite**).

If image coordinates specified to an image read instruction are normalized (as specified in the sampler), the `s`, `t`, and `r` coordinate values are multiplied by w_t, h_t and d_t respectively to generate the unnormalized coordinate values. For image arrays, the image array coordinate (i.e. `t` if it is a 1D image array or `r` if it is a 2D image array) specified to the image read instruction must always be the unnormalized image coordinate value.

Image coordinates specified to an image write instruction are always unnormalized image coordinate values.

Let `(u,v,w)` represent the unnormalized image coordinate values.

If values in `(s,t,r)` or `(u,v,w)` are INF or NaN, the behavior of the image read instruction or image write instruction is undefined.

9.2. Addressing and Filter Modes

After generating the image coordinate `(u,v,w)` we apply the appropriate addressing and filter mode to generate the appropriate sample locations to read from the image.

9.2.1. Clamp and None Addressing Modes

We first describe how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is `CL_ADDRESS_CLAMP`, `CL_ADDRESS_CLAMP_TO_EDGE`, or `CL_ADDRESS_NONE`.

Nearest Filtering

When the filter mode is `CL_FILTER_NEAREST`, the result of the image read instruction is the image element that is nearest (in Manhattan distance) to the image element location `(i,j,k)`. The image element location `(i,j,k)` is computed as:

$$
\begin{aligned}
i &= address\_mode((int)floor(u)) \\
j &= address\_mode((int)floor(v)) \\
k &= address\_mode((int)floor(w))
\end{aligned}
$$

For a 3D image, the image element at location `(i,j,k)` becomes the color value. For a 2D image, the image element at location `(i,j)` becomes the color value.

The below table describes the `address_mode` function.

Table 8. Addressing Modes to Generate Texel Location

Addressing Mode	Result of *address_mode(coord)*
`CL_ADDRESS_CLAMP`	*clamp (coord, -1, size)*
`CL_ADDRESS_CLAMP_TO_EDGE`	*clamp (coord, 0, size - 1)*
`CL_ADDRESS_NONE`	*coord*

The size term in the table above is w_t for u, h_t for v and d_t for w.

The clamp function used in the table above is defined as:

$$clamp(a, b, c) = return(a<b)?b:((a>c)?c:a)$$

If the addressing mode is `CL_ADDRESS_CLAMP` or `CL_ADDRESS_CLAMP_TO_EDGE`, and the selected texel location `(i,j,k)` refers to a location outside the image, the border color is used as the color value for the texel.

Otherwise, if the addressing mode is `CL_ADDRESS_NONE` and the selected texel location `(i,j,k)` refers to a location outside the image, the color value for the texel is undefined.

Linear Filtering

When the filter mode is `CL_FILTER_LINEAR`, a 2 x 2 square of image elements (for a 2D image) or a 2 x 2 x 2 cube of image elements (for a 3D image is selected). This 2 x 2 square or 2 x 2 x 2 cube is obtained as follows.

Let:

$$\begin{aligned} i0 &= address\_mode((int)floor(u-0.5)) \\ j0 &= address\_mode((int)floor(v-0.5)) \\ k0 &= address\_mode((int)floor(w-0.5)) \\ i1 &= address\_mode((int)floor(u-0.5)+1) \\ j1 &= address\_mode((int)floor(v-0.5)+1) \\ k1 &= address\_mode((int)floor(w-0.5)+1) \\ a &= frac(u-0.5) \\ b &= frac(v-0.5) \\ c &= frac(w-0.5) \end{aligned}$$

The frac function determines the fractional part of x and is computed as:

$$frac(x) = x - floor(x)$$

For a 3D image, the color value is computed as:

$$
\begin{aligned}
T \quad = \quad & (1-a) \times (1-b) \times (1-c) \times T_{i0\,j0k0} \\
& + a \times (1-b) \times (1-c) \times T_{i1\,j0k0} \\
& + (1-a) \times b \times (1-c) \times T_{i0\,j1k0} \\
& + a \times b \times (1-c) \times T_{i1\,j1k0} \\
& + (1-a) \times (1-b) \times c \times T_{i0\,j0k1} \\
& + a \times (1-b) \times c \times T_{i1\,j0k1} \\
& + (1-a) \times b \times c \times T_{i0\,j1k1} \\
& + a \times b \times c \times T_{i1\,j1k1}
\end{aligned}
$$

where T_{ijk} is the image element at location `(i,j,k)` in the 3D image.

For a 2D image, the color value is computed as:

$$
\begin{aligned}
T \quad = \quad & (1-a) \times (1-b) \times T_{i0\,j0} \\
& + a \times (1-b) \times T_{i1\,j0} \\
& + (1-a) \times b \times T_{i0\,j1} \\
& + a \times b \times T_{i1\,j1}
\end{aligned}
$$

where T_{ij} is the image element at location `(i,j)` in the 2D image.

If the addressing mode is `CL_ADDRESS_CLAMP` or `CL_ADDRESS_CLAMP_TO_EDGE`, and any of the selected T_{ijk} or T_{ij} refers to a location outside the image, the border color is used as the image element.

Otherwise, if the addressing mode is `CL_ADDRESS_NONE`, and any of the selected T_{ijk} or T_{ij} refers to a location outside the image, the color value is undefined.

If the image channel type is `CL_FLOAT` or `CL_HALF_FLOAT`, and any of the image elements T_{ijk} or T_{ij} is INF or NaN, the color value is undefined.

9.2.2. Repeat Addressing Mode

We now discuss how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is `CL_ADDRESS_REPEAT`.

Nearest Filtering

When filter mode is `CL_FILTER_NEAREST`, the result of the image read instruction is the image element that is nearest (in Manhattan distance) to the image element location `(i,j,k)`. The image element location `(i,j,k)` is computed as:

$$
\begin{aligned}
& u = (s - floor(s)) \times w_t \\
& i = (int) floor(u) \\
& if\,(i > w_t - 1) \\
& \qquad i = i - w_t \\
& v = (t - floor(t)) \times h_t \\
& j = (int) floor(v) \\
& if\,(j > h_t - 1) \\
& \qquad j = j - h_t \\
& w = (r - floor(r)) \times d_t \\
& k = (int) floor(w) \\
& if\,(k > d_t - 1) \\
& \qquad k = k - d_t
\end{aligned}
$$

For a 3D image, the image element at location (i, j, k) becomes the color value. For a 2D image, the

image element at location (i, j) becomes the color value.

Linear Filtering

When filter mode is `CL_FILTER_LINEAR`, a 2 x 2 square of image elements for a 2D image or a 2 x 2 x 2 cube of image elements for a 3D image is selected. This 2 x 2 square or 2 x 2 x 2 cube is obtained as follows.

Let

$$
\begin{aligned}
&u = (s - floor(s)) \times w_t \\
&i0 = (int)floor(u - 0.5) \\
&i1 = i0 + 1 \\
&if(i0<0) \\
&i0 = w_t + i0 \\
&if\,(i1>w_t - 1) \\
&\quad i1 = i1 - w_t \\
&v = (t - floor(t)) \times h_t \\
&j0 = (int)floor(v - 0.5) \\
&j1 = j0 + 1 \\
&if(j0<0) \\
&j0 = h_t + j0 \\
&if\,(j1>h_t - 1) \\
&\quad j1 = j1 - h_t \\
&w = (r - floor(r)) \times d_t \\
&k0 = (int)floor(w - 0.5) \\
&k1 = k0 + 1 \\
&if(k0<0) \\
&\quad k0 = d_t + k0 \\
&if\,(k1>d_t - 1) \\
&\quad k1 = k1 - d_t \\
&a = frac(u - 0.5) \\
&b = frac(v - 0.5) \\
&c = frac(w - 0.5)
\end{aligned}
$$

For a 3D image, the color value is computed as:

$$
\begin{aligned}
T \quad = \quad & (1-a) \times (1-b) \times (1-c) \times T_{i0\,j0k0} \\
& + a \times (1-b) \times (1-c) \times T_{i1\,j0k0} \\
& + (1-a) \times b \times (1-c) \times T_{i0\,j1k0} \\
& + a \times b \times (1-c) \times T_{i1\,j1k0} \\
& + (1-a) \times (1-b) \times c \times T_{i0\,j0k1} \\
& + a \times (1-b) \times c \times T_{i1\,j0k1} \\
& + (1-a) \times b \times c \times T_{i0\,j1k1} \\
& + a \times b \times c \times T_{i1\,j1k1}
\end{aligned}
$$

where T_{ijk} is the image element at location `(i,j,k)` in the 3D image.

For a 2D image, the color value is computed as:

$$
\begin{aligned}
T \quad = \quad & (1-a) \times (1-b) \times T_{i0\,j0} \\
& + a \times (1-b) \times T_{i1\,j0} \\
& + (1-a) \times b \times T_{i0\,j1} \\
& + a \times b \times T_{i1\,j1}
\end{aligned}
$$

where T_{ij} is the image element at location `(i,j)` in the 2D image.

If the image channel type is CL_FLOAT or CL_HALF_FLOAT, and any of the image elements T_{ijk} or T_{ij} is INF or NaN, the color value is undefined.

9.2.3. Mirrored Repeat Addressing Mode

We now discuss how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is CL_ADDRESS_MIRRORED_REPEAT. The CL_ADDRESS_MIRRORED_REPEAT addressing mode causes the image to be read as if it is tiled at every integer seam, with the interpretation of the image data flipped at each integer crossing.

Nearest Filtering

When filter mode is CL_FILTER_NEAREST, the result of the image read instruction is the image element that is nearest (in Manhattan distance) to the image element location (i,j,k). The image element location (i,j,k) is computed as:

$$
\begin{aligned}
s' &= 2.0f \times rint(0.5f \times s) \\
s` &= fabs(s - s`) \\
u &= s' \times w_t \\
i &= (int)floor(u) \\
i &= min(i, w_t - 1) \\
t' &= 2.0f \times rint(0.5f \times t) \\
t` &= fabs(t - t`) \\
v &= t' \times h_t \\
j &= (int)floor(v) \\
j &= min(j, h_t - 1) \\
r' &= 2.0f \times rint(0.5f \times r) \\
r` &= fabs(r - r`) \\
w &= r' \times d_t \\
k &= (int)floor(w) \\
k &= min(k, d_t - 1)
\end{aligned}
$$

For a 3D image, the image element at location (i, j, k) becomes the color value. For a 2D image, the image element at location (i, j) becomes the color value.

Linear Filtering

When filter mode is CL_FILTER_LINEAR, a 2 x 2 square of image elements for a 2D image or a 2 x 2 x 2 cube of image elements for a 3D image is selected. This 2 x 2 square or 2 x 2 x 2 cube is obtained as follows.

Let

$$
\begin{aligned}
s' &= 2.0f \times rint(0.5f \times s) \\
s` &= fabs(s - s`) \\
u &= s' \times w_t \\
i0 &= (int)floor(u - 0.5f) \\
i1 &= i0 + 1 \\
i0 &= max(i0, 0) \\
i1 &= min(i1, w_t - 1) \\
t' &= 2.0f \times rint(0.5f \times t) \\
t` &= fabs(t - t`) \\
v &= t' \times h_t \\
j0 &= (int)floor(v - 0.5f) \\
j1 &= j0 + 1 \\
j0 &= max(j0, 0) \\
j1 &= min(j1, h_t - 1) \\
r' &= 2.0f \times rint(0.5f \times r) \\
r` &= fabs(r - r`) \\
w &= r' \times d_t \\
k0 &= (int)floor(w - 0.5f) \\
k1 &= k0 + 1 \\
k0 &= max(k0, 0) \\
k1 &= min(k1, d_t - 1) \\
a &= frac(u - 0.5) \\
b &= frac(v - 0.5) \\
c &= frac(w - 0.5)
\end{aligned}
$$

For a 3D image, the color value is computed as:

$$
\begin{aligned}
T = {} & (1-a) \times (1-b) \times (1-c) \times T_{i0\,j0\,k0} \\
& + a \times (1-b) \times (1-c) \times T_{i1\,j0\,k0} \\
& + (1-a) \times b \times (1-c) \times T_{i0\,j1\,k0} \\
& + a \times b \times (1-c) \times T_{i1\,j1\,k0} \\
& + (1-a) \times (1-b) \times c \times T_{i0\,j0\,k1} \\
& + a \times (1-b) \times c \times T_{i1\,j0\,k1} \\
& + (1-a) \times b \times c \times T_{i0\,j1\,k1} \\
& + a \times b \times c \times T_{i1\,j1\,k1}
\end{aligned}
$$

where T_{ijk} is the image element at location `(i,j,k)` in the 3D image.

For a 2D image, the color value is computed as:

$$
\begin{aligned}
T = {} & (1-a) \times (1-b) \times T_{i0\,j0} \\
& + a \times (1-b) \times T_{i1\,j0} \\
& + (1-a) \times b \times T_{i0\,j1} \\
& + a \times b \times T_{i1\,j1}
\end{aligned}
$$

where T_{ij} is the image element at location `(i,j)` in the 2D image.

For a 1D image, the color value is computed as:

$$
T = (1-a) \times T_i0 + a \times T_i1
$$

where T_i is the image element at location `(i)` in the 1D image.

If the image channel type is `CL_FLOAT` or `CL_HALF_FLOAT` and any of the image elements T_{ijk} or T_{ij} is INF or NaN, the color value is undefined.

9.3. Precision of Addressing and Filter Modes

If the sampler is specified as using unnormalized coordinates (floating-point or integer coordinates), filter mode set to `CL_FILTER_NEAREST` and addressing mode set to one of the following modes - `CL_ADDRESS_CLAMP`, `CL_ADDRESS_CLAMP_TO_EDGE` or `CL_ADDRESS_NONE` - the location of the image element in the image given by `(i,j,k)` will be computed without any loss of precision.

For all other sampler combinations of normalized or unnormalized coordinates, filter modes, and addressing modes, the relative error or precision of the addressing mode calculations and the image filter operation are not defined. To ensure precision of image addressing and filter calculations across any OpenCL device for these sampler combinations, developers may unnormalize the image coordinate in the kernel, and then implement the linear filter in the kernel with appropriate read image instructions with a sampler that uses unnormalized coordinates, filter mode set to `CL_FILTER_NEAREST`, addressing mode set to `CL_ADDRESS_CLAMP`, `CL_ADDRESS_CLAMP_TO_EDGE` or `CL_ADDRESS_NONE`, and finally performing the interpolation of color values read from the image to generate the filtered color value.

9.4. Conversion Rules

In this section we discuss conversion rules that are applied when reading and writing images in a kernel.

9.4.1. Conversion Rules for Normalized Integer Channel Data Types

In this section we discuss converting normalized integer channel data types to half-precision and single-precision floating-point values and vice-versa.

Converting Normalized Integer Channel Data Types to Half Precision Floating-point Values

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, image read instructions will convert the channel values from an 8-bit or 16-bit unsigned integer to normalized half precision floating-point values in the range [0.0h … 1.0h].

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, image read instructions will convert the channel values from an 8-bit or 16-bit signed integer to normalized half precision floating-point values in the range [-1.0h … 1.0h].

These conversions are performed as follows:

- `CL_UNORM_INT8` (8-bit unsigned integer) → `half`

$$normalized\_half\_value(x) = round\_to\_half(\frac{x}{255})$$

- `CL_UNORM_INT_101010` (10-bit unsigned integer) → `half`

$$normalized\_half\_value(x) = round\_to\_half(\frac{x}{1023})$$

- `CL_UNORM_INT16` (16-bit unsigned integer) → `half`

$$normalized\_half\_value(x) = round\_to\_half(\frac{x}{65535})$$

- `CL_SNORM_INT8` (8-bit signed integer) → `half`

$$normalized\_half\_value(x) = max(-1.0h, round\_to\_half(\frac{x}{127}))$$

- `CL_SNORM_INT16` (16-bit signed integer) → `half`

$$normalized\_half\_value(x) = max(-1.0h, round\_to\_half(\frac{x}{32767}))$$

The precision of the above conversions is <= 1.5 ulp except for the following cases:

For `CL_UNORM_INT8`:

- 0 must convert to 0.0h, and
- 255 must convert to 1.0h

For `CL_UNORM_INT_101010`:

- 0 must convert to 0.0h, and
- 1023 must convert to 1.0h

For `CL_UNORM_INT16`:

- 0 must convert to 0.0h, and
- 65535 must convert to 1.0h

For `CL_SNORM_INT8`:

- -128 and -127 must convert to -1.0h,
- 0 must convert to 0.0h, and
- 127 must convert to 1.0h

For `CL_SNORM_INT16`:

- -32768 and -32767 must convert to -1.0h,
- 0 must convert to 0.0h, and
- 32767 must convert to 1.0h

Converting Half Precision Floating-point Values to Normalized Integer Channel Data Types

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, image write instructions will convert the half precision floating-point color value to an 8-bit or 16-bit unsigned integer.

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, image write instructions will convert the half precision floating-point color value to an 8-bit or 16-bit signed integer.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described below. When approximate rounding is used instead of the preferred rounding, the result of the conversion must satisfy the bound given below.

The conversions from half precision floating-point values to normalized integer values are performed is as follows:

- `half` → `CL_UNORM_INT8` (8-bit unsigned integer)

$$f(x) = max(0, min(255, 255 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6, \; x \neq \infty \text{and} x \neq NaN$$

- `half` → `CL_UNORM_INT16` (16-bit unsigned integer)

$$f(x) = max(0, min(65535, 65535 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6, \; x \neq \infty \text{and} x \neq NaN$$

- `half` → `CL_SNORM_INT8` (8-bit signed integer)

$$f(x) = max(-128, min(127, 127 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_int8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_int8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6, \; x \neq \infty \text{and} x \neq NaN$$

- `half` → `CL_SNORM_INT16` (16-bit signed integer)

$$f(x) = max(-32768, min(32767, 32767 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_int16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_int16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6, \; x \neq \infty \text{and} x \neq NaN$$

Converting Normalized Integer Channel Data Types to Floating-point Values

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, image read instructions will convert the channel values from an 8-bit or 16-bit unsigned integer to normalized floating-point values in the range [0.0f … 1.0f].

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, image read

instructions will convert the channel values from an 8-bit or 16-bit signed integer to normalized floating-point values in the range [-1.0f … 1.0f].

These conversions are performed as follows:

- `CL_UNORM_INT8` (8-bit unsigned integer) → `float`

$$normalized\_float\_value(x) = round\_to\_float(\frac{x}{255})$$

- `CL_UNORM_INT_101010` (10-bit unsigned integer) → `float`

$$normalized\_float\_value(x) = round\_to\_float(\frac{x}{1023})$$

- `CL_UNORM_INT16` (16-bit unsigned integer) → `float`

$$normalized\_float\_value(x) = round\_to\_float(\frac{x}{65535})$$

- `CL_SNORM_INT8` (8-bit signed integer) → `float`

$$normalized\_float\_value(x) = max(-1.0f, round\_to\_float(\frac{x}{127}))$$

- `CL_SNORM_INT16` (16-bit signed integer) → `float`

$$normalized\_float\_value(x) = max(-1.0f, round\_to\_float(\frac{x}{32767}))$$

The precision of the above conversions is <= 1.5 ulp except for the following cases.

For `CL_UNORM_INT8`:

- 0 must convert to 0.0f, and
- 255 must convert to 1.0f

For `CL_UNORM_INT_101010`:

- 0 must convert to 0.0f, and
- 1023 must convert to 1.0f

For `CL_UNORM_INT16`:

- 0 must convert to 0.0f, and
- 65535 must convert to 1.0f

For `CL_SNORM_INT8`:

- -128 and -127 must convert to -1.0f,
- 0 must convert to 0.0f, and
- 127 must convert to 1.0f

For `CL_SNORM_INT16`:

- -32768 and -32767 must convert to -1.0f,
- 0 must convert to 0.0f, and
- 32767 must convert to 1.0f

Converting Floating-point Values to Normalized Integer Channel Data Types

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, image write instructions will convert the floating-point color value to an 8-bit or 16-bit unsigned integer.

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, image write instructions will convert the floating-point color value to an 8-bit or 16-bit signed integer.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described below. When approximate rounding is used instead of the preferred rounding, the result of the conversion must satisfy the bound given below.

The conversions from half precision floating-point values to normalized integer values are performed is as follows:

- `float` → `CL_UNORM_INT8` (8-bit unsigned integer)

$$f(x) = max(0, min(255, 255 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_UNORM_INT_101010` (10-bit unsigned integer)

$$f(x) = max(0, min(1023, 1023 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint10(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint10(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_UNORM_INT16` (16-bit unsigned integer)

$$f(x) = max(0, min(65535, 65535 \times x))$$
$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$
$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_SNORM_INT8` (8-bit signed integer)

$$f(x) = max(-128, min(127, 127 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint8(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

- `float` → `CL_SNORM_INT16` (16-bit signed integer)

$$f(x) = max(-32768, min(32767, 32767 \times x))$$

$$f_{preferred}(x) = \begin{cases} round\_to\_nearest\_even\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$f_{approx}(x) = \begin{cases} round\_to\_impl\_uint16(f(x)) & x \neq \infty \text{and} x \neq NaN \\ \text{implementation-defined} & x = \infty \text{or} x = NaN \end{cases}$$

$$|f(x) - f_{approx}(x)| \leq 0.6,\ x \neq \infty \text{and} x \neq NaN$$

9.4.2. Conversion Rules for Half Precision Floating-point Channel Data Type

For images created with a channel data type of `CL_HALF_FLOAT`, the conversions of half to float and half to half are lossless. Conversions from float to half round the mantissa using the round to nearest even or round to zero rounding mode. Denormalized numbers for the half data type which may be generated when converting a float to a half may be flushed to zero. A float NaN must be converted to an appropriate NaN in the half type. A float INF must be converted to an appropriate INF in the half type.

9.4.3. Conversion Rules for Floating-point Channel Data Type

The following rules apply for reading and writing images created with channel data type of `CL_FLOAT`.

- NaNs may be converted to a NaN value(s) supported by the device.
- Denorms can be flushed to zero.
- All other values must be preserved.

9.4.4. Conversion Rules for Signed and Unsigned 8-bit, 16-bit and 32-bit Integer Channel Data Types

For images created with image channel data type of `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`, image read instructions will return the unmodified integer values stored in the image at specified location.

Likewise, for images created with image channel data type of `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`, image read instructions will return the unmodified unsigned integer values stored in the image at specified location.

Image write instructions will perform one of the following conversions:

- 32 bit signed integer → `CL_SIGNED_INT8` (8-bit signed integer):

$$int8\_value(x) = clamp(x, -128, 127)$$

- 32 bit signed integer → `CL_SIGNED_INT16` (16-bit signed integer):

$$int16\_value(x) = clamp(x, -32768, 32767)$$

- 32 bit signed integer → `CL_SIGNED_INT32` (32-bit signed integer):

$$int32\_value(x) = x \quad \text{(no conversion)}$$

- 32 bit unsigned integer → `CL_UNSIGNED_INT8` (8-bit unsigned integer):

$$uint8\_value(x) = clamp(x, 0, 255)$$

- 32 bit unsigned integer → `CL_UNSIGNED_INT16` (16-bit unsigned integer):

$$uint16\_value(x) = clamp(x, 0, 65535)$$

- 32 bit unsigned integer → `CL_UNSIGNED_INT32` (32-bit unsigned integer):

$$uint32\_value(x) = x \quad \text{(no conversion)}$$

The conversions described in this section must be correctly saturated.

9.4.5. Conversion Rules for sRGBA and sBGRA Images

Standard RGB data, which roughly displays colors in a linear ramp of luminosity levels such that an average observer, under average viewing conditions, can view them as perceptually equal steps on an average display. All 0`s maps to 0.0f, and all 1`s maps to 1.0f. The sequence of unsigned integer encodings between all 0`s and all 1`s represent a nonlinear progression in the floating-point interpretation of the numbers between 0.0f to 1.0f. For more detail, see the SRGB color standard.

Conversion from sRGB space is automatically done the image read instruction if the image channel order is one of the sRGB values described above. When reading from an sRGB image, the conversion from sRGB to linear RGB is performed before filtering is applied. If the format has an alpha channel, the alpha data is stored in linear color space. Conversion to sRGB space is automatically done by the image write instruction if the image channel order is one of the sRGB values described above and the device supports writing to sRGB images.

If the format has an alpha channel, the alpha data is stored in linear color space.

1. The following process is used by image read instructions to convert a normalized 8-bit unsigned integer sRGB color value x to a floating-point linear RGB color value y:
   a. Convert a normalized 8-bit unsigned integer sRGB value x to a floating-point sRGB value r as per rules described in Converting Normalized Integer Channel Data Types to Floating-point Values section.

$$r = normalized\_float\_value(x)$$

   b. Convert a floating-point sRGB value r to a floating-point linear RGB color value y:

$$c_{linear}(x) = \begin{cases} \frac{r}{12.92} & r \geq 0 \text{and} r \leq 0.04045 \\ (\frac{r + 0.055}{1.055})^{2.4} & r > 0.04045 \text{and} \leq 1 \end{cases}$$

$$y = c_{linear}(r)$$

2. The following process is used by image write instructions to convert a linear RGB floating-point color value y to a normalized 8-bit unsigned integer sRGB value x:

   a. Convert a floating-point linear RGB value y to a normalized floating point sRGB value r:

$$c_{linear}(x) = \begin{cases} 0 & y \geq NaN \text{or} y < 0 \\ 12.92 \times y & y \geq 0 \text{and} y < 0.0031308 \\ 1.055 \times y^{(\frac{1}{2.4})} & y \geq 0.0031308 \text{and} y \leq 1 \\ 1 & y > 1 \end{cases}$$

$$r = c_{sRGB}(y)$$

   b. Convert a normalized floating-point sRGB value r to a normalized 8-bit unsigned integer sRGB value x as per rules described in Converting Floating-point Values to Normalized Integer Channel Data Types section.

$$g(r) = \begin{cases} f_{preferred}(r) & \text{if rounding mode is round to even} \\ f_{approx}(r) & \text{if implementation-defined rounding mode} \end{cases}$$

$$x = g(r)$$

The accuracy required when converting a normalized 8-bit unsigned integer sRGB color value x to a floating-point linear RGB color value y is given by:

$$|x - 255 \times c_{sRGB}(y)| \leq 0.5$$

The accuracy required when converting a linear RGB floating-point color value y to a normalized 8-bit unsigned integer sRGB value x is given by:

$$|x - 255 \times c_{sRGB}(y)| \leq 0.6$$

9.5. Selecting an Image from an Image Array

Let `(u,v,w)` represent the unnormalized image coordinate values for reading from and/or writing to a 2D image in a 2D image array.

When read using a sampler, the 2D image layer selected is computed as:

$$layer = clamp(rint(w), 0, d_t - 1)$$

otherwise the layer selected is computed as:

$$layer = w$$

(since w is already an integer) and the result is undefined if w is not one of the integers 0, 1, ... d_t - 1.

Let `(u,v)` represent the unnormalized image coordinate values for reading from and/or writing to a 1D image in a 1D image array.

When read using a sampler, the 1D image layer selected is computed as:

$$layer = clamp(rint(v),\ 0,\ h_t - 1)$$

otherwise the layer selected is computed as:

$$layer = v$$

(since v is already an integer) and the result is undefined if v is not one of the integers 0, 1, ... h_t - 1.

9.6. Data Format for Reading and Writing Images

This section describes how image element data is returned by an image read instruction or passed as the *Texel* data that is written by an image write instruction:

For the following image channel orders, the data is a four component vector type:

Table 9. Mapping Image Data to Vector Components

Image Channel Order	Components
R, Rx	(R, 0, 0, 1)
A	(0, 0, 0, A)
RG, RGx	(R, G, 0, 1)
RGB, RGBx, sRGB, sRGBx	(R, G, B, 1)
RGBA, BGRA, ARGB, ABGR, sRGBA, sBGRA	(R, G, B, A)
Intensity	(I, I, I, I)
Luminance	(L, L, L, 1)

For the following image channel orders, the data is a scalar type:

Table 10. Scalar Image Data

Image Channel Order	Scalar Value
Depth	D
DepthStencil	D

The following table describes the mapping from image channel data type to the data vector component type or scalar type:

Table 11. Image Data Types

Image Channel Order	Data Type
SnormInt8, SnormInt16, UnormInt8, UnormInt16, UnormShort565, UnormShort555, UnormInt101010, UnormInt101010_2, UnormInt24, HalfFloat, Float	**OpTypeFloat**, with *Width* equal to 16 or 32.

Image Channel Order	Data Type
`SignedInt8`, `SignedInt16`, `SignedInt32`, `UnsignedInt8`, `UnsignedInt16`, `UnsignedInt32`	**OpTypeInt**, with *Width* equal to 32.

9.7. Sampled and Sampler-less Reads

SPIR-V instructions that read from an image without a sampler (such as **OpImageRead**) behave exactly the same as the corresponding image read instruction with a sampler that has *Sampler Filter Mode* set to **Nearest**, **Non-Normalized** coordinates, and *Sampler Addressing Mode* set to **None**.

There is one exception for cases where the image being read has *Image Format* equal to a floating-point type (such as **R32f**). In this exceptional case, when channel data values are denormalized, the non-sampler image read instruction may return the denormalized data, while the sampler image read instruction may flush denormalized channel data values to zero. The coordinates must be between 0 and image size in that dimension, non inclusive.

Chapter 10. Normative References

1. *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2008, http://dx.doi.org/10.1109/IEEESTD.2008.4610935 , August, 2008.
2. “ISO/IEC 9899:1999 - Programming Languages - C”, with technical corrigenda TC1 and TC2, https://www.iso.org/standard/29237.html .
3. “ISO/IEC 14882:2014 - Information technology - Programming languages - C++”, https://www.iso.org/standard/64029.html .
4. “The OpenCL Specification, Version 2.2”, https://www.khronos.org/registry/OpenCL/ .
5. “The OpenCL C Specification, Version 2.0”, https://www.khronos.org/registry/OpenCL/ .
6. “The OpenCL C++ 1.0 Specification”, https://www.khronos.org/registry/OpenCL/ .
7. “The OpenCL Extension Specification, Version 2.2”, https://www.khronos.org/registry/OpenCL/ .
8. “SPIR-V Specification, Version 1.2, Unified”, https://www.khronos.org/registry/spir-v/ .
9. Jean-Michel Muller. *On the definition of ulp(x)*. RR-5504, INRIA. 2005, pp.16. <inria-00070503>
10. “IEC 61966-2-1:1999 Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB”, https://webstore.iec.ch/publication/6169 .