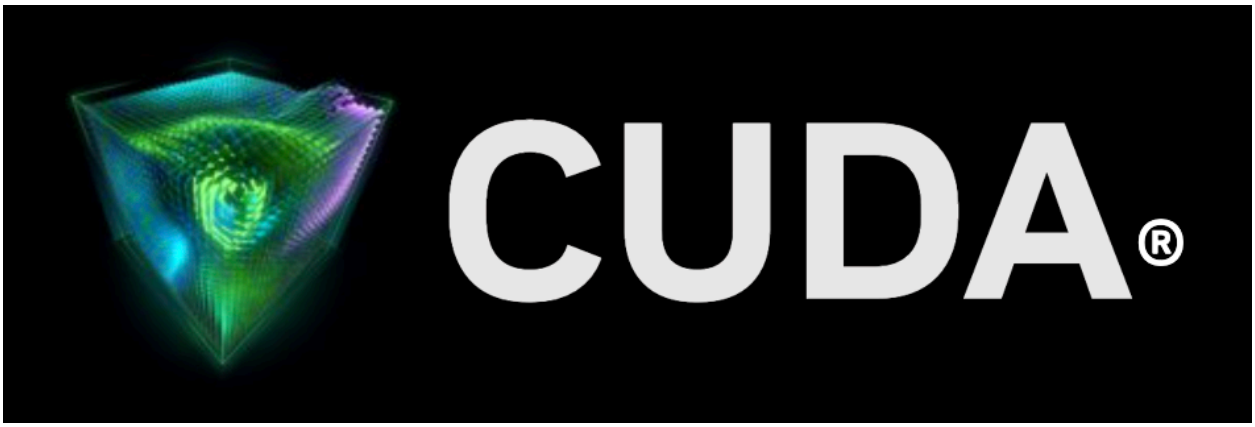

NVVM IR Specification

Release 13.1

NVIDIA Corporation

Jan 08, 2026

Contents

1 Identifiers **3**

2 High Level Structure **5**
2.1 Linkage Types . . . 5
2.2 Calling Conventions . . . 5
2.2.1 Rules and Restrictions . . . 5
2.3 Visibility Styles . . . 6
2.4 DLL Storage Classes . . . 7
2.5 Thread Local Storage Models . . . 7
2.6 Runtime Preemption Specifiers . . . 7
2.7 Structure Types . . . 7
2.8 Non-Integral Pointer Type . . . 7
2.9 Comdats . . . 7
2.10 source_filename . . . 7
2.11 Global Variables . . . 8
2.12 Functions . . . 8
2.13 Aliases . . . 8
2.14 Ifuncs . . . 8
2.15 Named Metadata . . . 9
2.16 Parameter Attributes . . . 9
2.17 Garbage Collector Strategy Names . . . 10
2.18 Prefix Data . . . 10
2.19 Prologue Data . . . 10
2.20 Attribute Groups . . . 10
2.21 Function Attributes . . . 10
2.22 Global Attributes . . . 11
2.23 Operand Bundles . . . 11
2.24 Module-Level Inline Assembly . . . 11
2.25 Data Layout . . . 11
2.26 Target Triple . . . 12
2.27 Pointer Aliasing Rules . . . 12
2.28 Volatile Memory Access . . . 12
2.29 Memory Model for Concurrent Operations . . . 12
2.30 Atomic Memory Ordering Constraints . . . 12
2.31 Fast-Math Flags . . . 13
2.32 Use-list Order Directives . . . 13

3 Type System **15**

4 Constants **17**

5 Other Values **19**
5.1 Inline Assembler Expressions . . . 19

6 Metadata **21**
6.1 Metadata Nodes and Metadata Strings . . . 21

7 ThinLTO Summary **23**

8 Intrinsic Global Variables **25**

9 Instructions **27**
9.1 Terminator Instructions . . . 27
9.2 Binary Operations . . . 27
9.3 Bitwise Binary Operations . . . 28
9.4 Vector Operations . . . 28
9.5 Aggregate Operations . . . 28
9.6 Memory Access and Addressing Operations . . . 28
9.6.1 alloca Instruction . . . 28
9.6.2 load Instruction . . . 29
9.6.3 store Instruction . . . 29
9.6.4 fence Instruction . . . 29
9.6.5 cmpxchg Instruction . . . 29
9.6.6 atomicrmw Instruction . . . 29
9.6.7 getelementptr Instruction . . . 30
9.7 Conversion Operations . . . 30
9.8 Other Operations . . . 30

10 Supported Intrinsic Functions **31**
10.1 Supported Variable Argument Handling Intrinsics . . . 31
10.2 Supported Standard C/C++ Library Intrinsics . . . 31
10.3 Supported Bit Manipulations Intrinsics . . . 32
10.4 Supported Specialised Arithmetic Intrinsics . . . 32
10.5 Supported Arithmetic with Overflow Intrinsics . . . 32
10.6 Supported Half Precision Floating Point Intrinsics . . . 32
10.7 Supported Debugger Intrinsics . . . 33
10.8 Supported Memory Use Markers . . . 33
10.9 Supported General Intrinsics . . . 33

11 Address Space **35**
11.1 Address Spaces . . . 35
11.2 Generic Pointers and Non-Generic Pointers . . . 36
11.2.1 Generic Pointers vs. Non-generic Pointers . . . 36
11.2.2 Conversion . . . 37
11.2.3 No Aliasing between Two Different Specific Address Spaces . . . 37
11.3 The alloca Instruction . . . 38

12 Global Property Annotation **39**
12.1 Overview . . . 39
12.2 Representation of Properties . . . 39
12.3 Supported Properties . . . 40

13 Texture and Surface **41**
13.1 Texture Variable and Surface Variable . . . 41
13.2 Accessing Texture Memory or Surface Memory . . . 41

14 NVVM Specific Intrinsic Functions **43**
14.1 Atomic . . . 43
14.2 Barrier and Memory Fence . . . 44

14.3 Address space conversion . . . 46
14.4 Special Registers . . . 46
14.5 Texture/Surface Access . . . 46
14.5.1 Texture Reads . . . 47
14.5.2 Surface Loads . . . 51
14.5.3 Surface Stores . . . 52
14.6 Warp-level Operations . . . 56
14.6.1 Barrier Synchronization . . . 56
14.6.2 Data Movement . . . 56
14.6.3 Vote . . . 57
14.6.4 Match . . . 58
14.6.5 Matrix Operation . . . 59
14.6.5.1 Load Fragments . . . 59
14.6.5.2 Store Fragments . . . 60
14.6.5.3 Matrix Multiply-and-Accumulate . . . 61

15 Source Level Debugging Support **63**

16 NVVM ABI for PTX **65**
16.1 Linkage Types . . . 65
16.2 Parameter Passing and Return . . . 66

17 Revision History **67**

18 Notices **69**
18.1 Notice . . . 69
18.2 OpenCL . . . 70
18.3 Trademarks . . . 70

NVVM IR Specification

Reference guide to the NVVM compiler (intermediate representation) based on the LLVM IR.

NVVM IR is a compiler IR (intermediate representation) based on the LLVM IR. The NVVM IR is designed to represent GPU compute kernels (for example, CUDA kernels). High-level language front-ends, like the CUDA C compiler front-end, can generate NVVM IR. The NVVM compiler (which is based on LLVM) generates PTX code from NVVM IR.

NVVM IR and NVVM compilers are mostly agnostic about the source language being used. The PTX codegen part of a NVVM compiler needs to know the source language because of the difference in DCI (driver/compiler interface).

NVVM IR is a binary format and is based on a subset of LLVM IR bitcode format. This document uses only human-readable form to describe NVVM IR.

Technically speaking, NVVM IR is LLVM IR with a set of rules, restrictions, and conventions, plus a set of supported intrinsic functions. A program specified in NVVM IR is always a legal LLVM program. A legal LLVM program may not be a legal NVVM program.

There are three levels of support for NVVM IR.

- Supported: The feature is fully supported. Most IR features should fall into this category.
- Accepted and ignored: The NVVM compiler will accept this IR feature, but will ignore the required semantics. This applies to some IR features that do not have meaningful semantics on GPUs and that can be ignored. Calling convention markings are an example.
- Illegal, not supported: The specified semantics is not supported, such as a `fence` instruction. Future versions of NVVM may either support or accept and ignore IRs that are illegal in the current version.

This document describes version 2.0 of the NVVM IR and version 3.1 of the NVVM debug metadata (see *Source Level Debugging Support*). The 2.0 version of NVVM IR is incompatible with the previous version 1.11. Linking of NVVM IR Version 1.11 with 2.0 will result in compiler error.

NVVM IR can be in one of two dialects. The LLVM 7 dialect is based on LLVM 7.0.1. The modern dialect is based on a more recent public release version of LLVM (LLVM 20.1.0). The modern dialect only supports Blackwell and later architectures (compute capability `compute_100` or greater). For the complete semantics of the IR, readers of this document should refer to either the official LLVM Language Reference Manual version 7 or version 20. This document is annotated with notes when differences between the two NVVM IR dialects are important.

Chapter 1. Identifiers

The name of a named global identifier must have the form:

`@[a-zA-Z$_][a-zA-Z$_0-9]*`

Note that it cannot contain the . character.

`[@%]llvm.nvvm.*` and `[@%]nvvm.*` are reserved words.

Chapter 2. High Level Structure

2.1. Linkage Types

Supported:

- `private`
- `internal`
- `available_externally`
- `linkonce`
- `weak`
- `common`
- `linkonce_odr`
- `weak_odr`
- `external`

All other linkage types are not supported.

See NVVM ABI for PTX for details on how linkage types are translated to PTX.

2.2. Calling Conventions

All LLVM calling convention markings are accepted and ignored. Functions and calls are generated according to the PTX calling convention.

2.2.1. Rules and Restrictions

1. When an argument with width less than 32-bit is passed, the `zeroext/signext` parameter attribute should be set. `zeroext` will be assumed if not set.
2. When a value with width less than 32-bit is returned, the `zeroext/signext` parameter attribute should be set. `zeroext` will be assumed if not set.

3. Arguments of aggregate or vector types that are passed by value can be passed by pointer with the `byval` attribute set (referred to as the `by-pointer-byval` case below). The align attribute must be set if the type requires a non-natural alignment (natural alignment is the alignment inferred for the aggregate type according to the Data Layout section).
4. If a function has an argument of aggregate or vector type that is passed by value directly and the type has a non-natural alignment requirement, the alignment must be annotated by the global property annotation `<align`, alignment>, where alignment is a 32-bit integer whose upper 16 bits represent the argument position (starting from 1) and the lower 16 bits represent the alignment.
5. If the return type of a function is an aggregate or a vector that has a non-natural alignment, then the alignment requirement must be annotated by the global property annotation `<align`, alignment>, where the upper 16 bits is 0, and the lower 16 bits represent the alignment.
6. It is not required to annotate a function with `<align`, alignment> otherwise. If annotated, the alignment must match the natural alignment or the align attribute in the `by-pointer-byval` case.
7. For an indirect call instruction of a function that has a non-natural alignment for its return value or one of its arguments that is not expressed in alignment in the `by-pointer-byval` case, the call instruction must have an attached metadata of kind `callalign`. The metadata contains a sequence of `i32` fields each of which represents a non-natural alignment requirement. The upper 16 bits of an `i32` field represent the argument position (0 for return value, 1 for the first argument, and so on) and the lower 16 bits represent the alignment. The `i32` fields must be sorted in the increasing order.

   For example,

```
%call = call %struct.S %fp1(%struct.S* byval align 8 %arg1p, %struct.S %arg2),!
↪callalign !10
!10 = !{i32 8, i32 520};
```

8. It is not required to have an `i32` metadata field for the other arguments or the return value otherwise. If presented, the alignment must match the natural alignment or the align attribute in the `by-pointer-byval case`.
9. It is not required to have a `callalign` metadata attached to a direct call instruction. If attached, the alignment must match the natural alignment or the alignment in the `by-pointer-byval` case.
10. The absence of the metadata in an indirect call instruction means using natural alignment or the align attribute in the `by-pointer-byval` case.

2.3. Visibility Styles

All styles—default, hidden, and protected—are accepted and ignored.

2.4. DLL Storage Classes

Not supported.

2.5. Thread Local Storage Models

Not supported.

2.6. Runtime Preemption Specifiers

Not supported.

2.7. Structure Types

Supported.

2.8. Non-Integral Pointer Type

Not supported.

2.9. Comdats

Not supported.

2.10. source_filename

Accepted and ignored.

2.11. Global Variables

A global variable, that is not an intrinsic global variable, may be optionally declared to reside in one of the following address spaces:

- `global`
- `shared`
- `constant`

If no address space is explicitly specified, the global variable is assumed to reside in the `global` address space with a generic address value. See Address Space for details.

`thread_local` variables are not supported.

No explicit section (except for the metadata section) is allowed.

Initializations of `shared` variables are not supported. Use undef initialization.

2.12. Functions

The following are not supported on functions:

- Alignment
- Explicit section
- Garbage collector name
- Prefix data
- Prologue
- Personality

2.13. Aliases

Supported only as aliases of non-kernel functions.

2.14. Ifuncs

Not supported.

2.15. Named Metadata

Accepted and ignored, except for the following:

- `!nvvm.annotations`: see Global Property Annotation
- `!nvvmir.version`
- `!llvm.dbg.cu`
- `!llvm.module.flags`

The NVVM IR version is specified using a named metadata called `!nvvmir.version`. The `!nvvmir.version` named metadata may have one metadata node that contains the NVVM IR version for that module. If multiple such modules are linked together, the named metadata in the linked module may have more than one metadata node with each node containing a version. A metadata node with NVVM IR version takes either of the following forms:

- It may consist of two i32 values—the first denotes the NVVM IR major version number and the second denotes the minor version number. If absent, the version number is assumed to be 1.0, which can be specified as:

```
!nvvmir.version = !{!0}
!0 = !{i32 1, i32 0}
```

- It may consist of four i32 values—the first two denote the NVVM IR major and minor versions respectively. The third value denotes the NVVM IR debug metadata major version number, and the fourth value denotes the corresponding minor version number. If absent, the version number is assumed to be 1.0, which can be specified as:

```
!nvvmir.version = !{!0}
!0 = !{i32 1, i32 0, i32 1, i32 0}
```

The version of NVVM IR described in this document is 2.0. The version of NVVM IR debug metadata described in this document is 3.1.

2.16. Parameter Attributes

Supported, except the following:

Accepted and ignored:

- `inreg`
- `nest`

All other parameter attributes are not supported.

See Calling Conventions for the use of the attributes.

2.17. Garbage Collector Strategy Names

Not supported.

2.18. Prefix Data

Not supported.

2.19. Prologue Data

Not supported.

2.20. Attribute Groups

Supported. The set of supported attributes is equal to the set of attributes accepted where the attribute group is used.

2.21. Function Attributes

Supported:

- `allocsize`
- `alwaysinline`
- `cold`
- `convergent`
- `inaccessiblememonly`
- `inaccessiblemem_or_argmemonly`
- `inlinehint`
- `minsize`
- `no-jump-tables`
- `noduplicate`
- `noinline`
- `noreturn`
- `norecurse`
- `nounwind`

- `"null-pointer-is-valid"`
- `optforfuzzing`
- `optnone`
- `optsize`
- `readnone`
- `readonly`
- `writeonly`
- `argmemonly`
- `speculatable`
- `strictfp`

All other function attributes are not supported.

2.22. Global Attributes

Not supported.

2.23. Operand Bundles

Not supported.

2.24. Module-Level Inline Assembly

Supported.

2.25. Data Layout

Only the following data layout is supported:

- 64-bit

  e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-i128:128:128-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64

The following data layouts are deprecated and will be removed in a future release.

- 32-bit

  e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-i128:128:128-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64

e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64

- 64-bit

  e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64

2.26. Target Triple

Only the following target triple is supported, where * can be any name:

- 64-bit: `nvptx64-*-cuda`

The following target triple is deprecated, and will be removed in future release:

- 32-bit: `nvptx-*-cuda`

2.27. Pointer Aliasing Rules

Supported.

2.28. Volatile Memory Access

Supported. Note that for code generation: `ld.volatile` and `st.volatile` will be generated.

2.29. Memory Model for Concurrent Operations

Not applicable. Threads in an NVVM IR program must use atomic operations or barrier synchronization to communicate.

2.30. Atomic Memory Ordering Constraints

Atomic loads and stores are not supported. Other atomic operations on other than 32-bit or 64-bit operands are not supported.

2.31. Fast-Math Flags

Supported.

2.32. Use-list Order Directives

Not supported.

Chapter 3. Type System

Supported, except for the following:

- Floating point types `fp128`, `x86_fp80` and `ppc_fp128` are not supported.
- The `x86_mmx` type is not supported.
- The `token` type is not supported.
- The `non-integral pointer` type is not supported.

Chapter 4. Constants

Supported, except for the following:

- `Token constants` are not supported.
- `blockaddress(@function, %block)` is not supported.
- For a constant expression that is used as the initializer of a global variable `@g1`, if the constant expression contains a global identifier `@g2`, then the constant expression is supported if it can be reduced to the form of `bitcast+offset`, where offset is an integer number (including `0`)

Chapter 5. Other Values

5.1. Inline Assembler Expressions

Inline assembler of PTX instructions is supported, with the following supported constraints:

Constraint	Type
c	i8
h	i16
r	i32
l	i64
f	f32
d	f64

The inline asm metadata `!srcloc` is accepted and ignored.

The inline asm dialect `inteldialect` is not supported.

Chapter 6. Metadata

6.1. Metadata Nodes and Metadata Strings

Supported.

The following metadata are understood by the NVVM compiler:

- Specialized Metadata Nodes
- `llvm.loop.unroll.count`
- `llvm.loop.unroll.disable`
- `llvm.loop.unroll.full`
- `callalign` (see Rules and Restrictions for Calling Conventions)

Module flags metadata (`llvm.module.flags`) is supported and verified, but the metadata values will be ignored.

All other metadata is accepted and ignored.

Chapter 7. ThinLTO Summary

Not supported.

Chapter 8. Intrinsic Global Variables

- The `llvm.used` global variable is supported.
- The `llvm.compiler.used` global variable is supported
- The `llvm.global_ctors` global variable is not supported
- The `llvm.global_dtors` global variable is not supported

Chapter 9. Instructions

9.1. Terminator Instructions

Supported:

- `ret`
- `br`
- `switch`
- `unreachable`

All other terminator instructions are not supported.

9.2. Binary Operations

Supported:

- `add`
- `fadd`
- `sub`
- `fsub`
- `mul`
- `fmul`
- `udiv`
- `sdiv`
- `fdiv`
- `urem`
- `srem`
- `frem`

9.3. Bitwise Binary Operations

Supported:

- `shl`
- `lshr`
- `ashr`
- `and`
- `or`
- `xor`

9.4. Vector Operations

Supported:

- `extractelement`
- `insertelement`
- `shufflevector`

9.5. Aggregate Operations

Supported:

- `extractvalue`
- `insertvalue`

9.6. Memory Access and Addressing Operations

9.6.1. alloca Instruction

The `alloca` instruction returns a generic pointer to the local address space. The `inalloca` attribute is not supported. Maximum alignment supported is 2^23. The `addrspace(<num>)` specifier is supported only if `num` is 0.

9.6.2. load Instruction

`load atomic` is not supported.

9.6.3. store Instruction

`store atomic` is not supported.

9.6.4. fence Instruction

Not supported. Use NVVM intrinsic functions instead.

9.6.5. cmpxchg Instruction

Supported for `i32`, `i64`, and `i128` types, with the following restrictions:

- The pointer must be either a global pointer, a shared pointer, or a generic pointer that points to either the global address space or the shared address space.
- The `weak` marker and the `failure ordering` are accepted and ignored.
- The `i128` type is only supported on `compute_90` and above.

9.6.6. atomicrmw Instruction

Only the following operations are supported:

- `xchg`
- `add`
- `sub`
- `and`
- `or`
- `xor`
- `max`
- `min`
- `umax`
- `umin`

All other operations are not supported.

The operations support the `i32` and `i64` types. The `xchg` operation additionally supports `i128` on `compute_90` and above.

The pointer operand must be either a global pointer, a shared pointer, or a generic pointer that points to either the `global` address space or the `shared` address space.

9.6.7. getelementptr Instruction

Supported.

9.7. Conversion Operations

Supported:

- `trunc .. to`
- `zext .. to`
- `sext .. to`
- `fptrunc .. to`
- `fpext .. to`
- `fptoui .. to`
- `fptosi .. to`
- `uitofp .. to`
- `sitofp .. to`
- `ptrtoint .. to`
- `inttoptr .. to`
- `addrspacecast .. to`
- `bitcast .. to`

  See Conversion for a special use case of `bitcast`.

9.8. Other Operations

Supported:

- `icmp`
- `fcmp`
- `phi`
- `select`
- `va_arg`
- `call` (See Calling Conventions for other rules and restrictions.)

All other operations are not supported.

Chapter 10. Supported Intrinsic Functions

10.1. Supported Variable Argument Handling Intrinsics

- `llvm.va_start`
- `llvm.va_end`
- `llvm.va_copy`

10.2. Supported Standard C/C++ Library Intrinsics

- `llvm.copysign`

  This is only supported in the modern NVVM IR dialect.
- `llvm.memcpy`

  Note that the constant address space cannot be used as the destination since it is read-only.
- `llvm.memmove`

  Note that the constant address space cannot be used since it is read-only.
- `llvm.memset`

  Note that the constant address space cannot be used since it is read-only.
- `llvm.sqrt`

  Supported for float/double and vector of float/double. Mapped to PTX `sqrt.rn.f32` and `sqrt.rn.f64`.
- `llvm.fma`

  Supported for float/double and vector of float/double. Mapped to PTX `fma.rn.f32` and `fma.rn.f64`

10.3. Supported Bit Manipulations Intrinsics

- `llvm.bitreverse`

  Supported for `i8`, `i16`, `i32`, and `i64`.
- `llvm.bswap`

  Supported for `i16`, `i32`, and `i64`.
- `llvm.ctpop`

  Supported for `i8`, `i16`, `i32`, `i64`, and vectors of these types.
- `llvm.ctlz`

  Supported for `i8`, `i16`, `i32`, `i64`, and vectors of these types.
- `llvm.cttz`

  Supported for `i8`, `i16`, `i32`, `i64`, and vectors of these types.
- `llvm.fshl`

  Supported for `i8`, `i16`, `i32`, and `i64`.
- `llvm.fshr`

  Supported for `i8`, `i16`, `i32`, and `i64`.

10.4. Supported Specialised Arithmetic Intrinsics

- `llvm.fmuladd`

10.5. Supported Arithmetic with Overflow Intrinsics

Supported for `i16`, `i32`, and `i64`.

10.6. Supported Half Precision Floating Point Intrinsics

- `llvm.convert.to.fp16`
- `llvm.convert.from.fp16`

10.7. Supported Debugger Intrinsics

- `llvm.dbg.addr`
- `llvm.dbg.declare`
- `llvm.dbg.value`

10.8. Supported Memory Use Markers

- `llvm.lifetime.start`
- `llvm.lifetime.end`
- `llvm.invariant.start`
- `llvm.invariant.end`

10.9. Supported General Intrinsics

- `llvm.var.annotation`

  Accepted and ignored.
- `llvm.ptr.annotation`

  Accepted and ignored.
- `llvm.annotation`

  Accepted and ignored.
- `llvm.trap`
- `llvm.expect`
- `llvm.assume`
- `llvm.donothing`
- `llvm.sideeffect`

Chapter 11. Address Space

11.1. Address Spaces

NVVM IR has a set of predefined memory address spaces, whose semantics are similar to those defined in CUDA C/C++, OpenCL C and PTX. Any address space not listed below is not supported .

Name	Address Space Number	Semantics/Example
code	0	functions, code ▶ CUDA C/C++ function ▶ OpenCL C function
generic	0	Can only be used to qualify the pointee of a pointer ▶ Pointers in CUDA C/C++
global	1	▶ CUDA C/C++ __device__ ▶ OpenCL C global
shared	3	▶ CUDA C/C++ __shared__ ▶ OpenCL C local
constant	4	▶ CUDA C/C++ __constant__ ▶ OpenCL C constant
local	5	▶ CUDA C/C++ local ▶ OpenCL C private
<reserved>	2, 101 and above	

Each global variable, that is not an intrinsic global variable, can be declared to reside in a specific non-zero address space, which can only be one of the following: `global`, `shared` or `constant`.

If a non-intrinsic global variable is declared without any address space number or with the address

space number 0, then this global variable resides in address space `global` and the pointer of this global variable holds a generic pointer value.

The predefined NVVM memory spaces are needed for the language front-ends to model the memory spaces in the source languages. For example,

```
// CUDA C/C++
__constant__ int c;
__device__ int g;

; NVVM IR
@c = addrspace(4) global i32 0, align 4
@g = addrspace(1) global [2 x i32] zeroinitializer, align 4
```

Address space numbers 2 and 101 or higher are reserved for NVVM compiler internal use only. No language front-end should generate code that uses these address spaces directly.

11.2. Generic Pointers and Non-Generic Pointers

11.2.1. Generic Pointers vs. Non-generic Pointers

There are generic pointers and non-generic pointers in NVVM IR. A generic pointer is a pointer that may point to memory in any address space. A non-generic pointer points to memory in a specific address space.

In NVVM IR, a generic pointer has a pointer type with the address space `generic`, while a non-generic pointer has a pointer type with a non-generic address space.

Note that the address space number for the generic address space is 0—the default in both NVVM IR and LLVM IR. The address space number for the code address space is also 0. Function pointers are qualified by address space `code` (`addrspace(0)`).

Loads/stores via generic pointers are supported, as well as loads/stores via non-generic pointers. Loads/stores via function pointers are not supported

```
@a = addrspace(1) global i32 0, align 4 ; 'global' addrspace, @a holds a specific
↪value
@b = global i32 0, align 4              ; 'global' addrspace, @b holds a generic value
@c = addrspace(4) global i32 0, align 4 ; 'constant' addrspace, @c holds a specific
↪value

... = load i32 addrspace(1)* @a, align 4 ; Correct
... = load i32* @a, align 4              ; Wrong
... = load i32* @b, align 4              ; Correct
... = load i32 addrspace(1)* @b, align 4 ; Wrong
... = load i32 addrspace(4)* @c, align4  ; Correct
... = load i32* @c, align 4              ; Wrong
```

11.2.2. Conversion

The bit value of a generic pointer that points to a specific object may be different from the bit value of a specific pointer that points to the same object.

The `addrspacecast` IR instruction should be used to perform pointer casts across address spaces (generic to non-generic or non-generic to generic). Casting a non-generic pointer to a different non-generic pointer is not supported. Casting from a generic to a non-generic pointer is undefined if the generic pointer does not point to an object in the target non-generic address space.

`inttoptr` and `ptrtoint` are supported. `inttoptr` and `ptrtoint` are value preserving instructions when the two operands are of the same size. In general, using `ptrtoint` and `inttoptr` to implement an address space cast is undefined.

The following intrinsic can be used to query if the argument pointer was derived from the address of a kernel function parameter that has the grid_constant property:

```
i1 @llvm.nvvm.isspacep.grid_const(i8*)
```

The following intrinsic can be used to query if the input generic pointer was derived from the address of a variable allocated in the shared address space, in a CTA that is part of the same cluster as the parent CTA of the invoking thread. This intrinsic is only supported for Hopper+.

```
i1 @llvm.nvvm.isspacep.cluster_shared(i8*)
```

The following intrinsics can be used to query if a generic pointer can be safely cast to a specific non-generic address space:

- `i1 @llvm.nvvm.isspacep.const(i8*)`
- `i1 @llvm.nvvm.isspacep.global(i8*)`
- `i1 @llvm.nvvm.isspacep.local(i8*)`
- `i1 @llvm.nvvm.isspacep.shared(i8*)`

`bitcast` on pointers is supported, though LLVM IR forbids `bitcast` from being used to change the address space of a pointer.

11.2.3. No Aliasing between Two Different Specific Address Spaces

Two different specific address spaces do not overlap. NVVM compiler assumes two memory accesses via non-generic pointers that point to different address spaces are not aliased.

11.3. The alloca Instruction

The `alloca` instruction returns a generic pointer that only points to address space `local`.

Chapter 12. Global Property Annotation

12.1. Overview

NVVM uses Named Metadata to annotate IR objects with properties that are otherwise not representable in the IR. The NVVM IR producers can use the Named Metadata to annotate the IR with properties, which the NVVM compiler can process.

12.2. Representation of Properties

For each translation unit (that is, per bitcode file), there is a named metadata called `nvvm.annotations`.

This named metadata contains a list of MDNodes.

The first operand of each MDNode is an entity that the node is annotating using the remaining operands.

Multiple MDNodes may provide annotations for the same entity, in which case their first operands will be same.

The remaining operands of the MDNode are organized in order as <property-name, value>.

- The property-name operand is MDString, while the value is `i32`.
- Starting with the operand after the annotated entity, every alternate operand specifies a property.
- The operand after a property is its value.

  The following is an example.

```
!nvvm.annotations = !{!12, !13}
  !12 = !{void (i32, i32)* @_Z6kernelii, !"kernel", i32 1}
  !13 = !{void ()* @_Z7kernel2v, !"kernel", i32 1, !"maxntidx", i32 16}
```

If two bitcode files are being linked and both have a named metadata `nvvm.annotations`, the linked file will have a single merged named metadata. If both files define properties for the same entity foo , the linked file will have two MDNodes defining properties for foo . It is illegal for the files to have conflicting properties for the same entity.

12.3. Supported Properties

Property Name	Annotated On	Description
`maxntid{x, y, z}`	kernel function	Maximum expected CTA size from any launch.
`reqntid{x, y, z}`	kernel function	Minimum expected CTA size from any launch.
`cluster_dim_{x,y,z}`	kernel function	Support for cluster dimensions for Hopper+. If any dimension is specified as 0, then all dimensions must be specified as 0.
`cluster_max_blocks`	kernel function	Maximum number of blocks per cluster. Must be non-zero. Only supported for Hopper+.
`minctasm`	kernel function	Hint/directive to the compiler/driver, asking it to put at least these many CTAs on an SM.
`grid_constant`	kernel function	The argument is a metadata node, which contains a list of integers, where each integer n denotes that the nth parameter has the grid_constant annotation (numbering from 1). The parameter's type must be of pointer type with byval attribute set. It is undefined behavior to write to memory pointed to by the parameter. This property is only supported for Volta+.
`maxnreg`	function	Maximum number of registers for function.
`kernel`	function	Signifies that this function is a kernel function.
`align`	function	Signifies that the value in low 16-bits of the 32-bit value contains alignment of n th parameter type if its alignment is not the natural alignment. n is specified by high 16-bits of the value. For return type, n is 0.
`texture`	global variable	Signifies that variable is a texture.
`surface`	global variable	Signifies that variable is a surface.
`managed`	global variable	Signifies that variable is a UVM managed variable.

Chapter 13. Texture and Surface

13.1. Texture Variable and Surface Variable

A texture or a surface variable can be declared/defined as a global variable of `i64` type with annotation `texture` or `surface` in the `global` address space.

A texture or surface variable must have a name, which must follow identifier naming conventions.

It is illegal to store to or load from the address of a texture or surface variable. A texture or a surface variable may only have the following uses:

- In a metadata node
- As an intrinsic function argument as shown below
- In `llvm.used` Global Variable

13.2. Accessing Texture Memory or Surface Memory

Texture memory and surface memory can be accessed using texture or surface handles. NVVM provides the following intrinsic function to get a texture or surface handle from a texture or surface variable.

```
delcare i64 %llvm.nvvm.texsurf.handle.p1i64(metadata, i64 addrspace(1)*)
```

The first argument to the intrinsic is a metadata holding the texture or surface variable. Such a metadata may hold only one texture or one surface variable. The second argument to the intrinsic is the texture or surface variable itself. The intrinsic returns a handle of `i64` type.

The returned handle value from the intrinsic call can be used as an operand (with a constraint of l) in a PTX inline asm to access the texture or surface memory.

Chapter 14. NVVM Specific Intrinsic Functions

14.1. Atomic

Besides the atomic instructions, the following extra atomic intrinsic functions are supported.

```
declare float @llvm.nvvm.atomic.load.add.f32.p0f32(float* address, float val)
declare float @llvm.nvvm.atomic.load.add.f32.p1f32(float addrspace(1)* address, float
↪val)
declare float @llvm.nvvm.atomic.load.add.f32.p3f32(float addrspace(3)* address, float
↪val)
declare double @llvm.nvvm.atomic.load.add.f64.p0f64(double* address, double val)
declare double @llvm.nvvm.atomic.load.add.f64.p1f64(double addrspace(1)* address,
↪double val)
declare double @llvm.nvvm.atomic.load.add.f64.p3f64(double addrspace(3)* address,
↪double val)
```

reads the single/double precision floating point value `old` located at the address `address`, computes `old+val`, and stores the result back to memory at the same address. These operations are performed in one atomic transaction. The function returns `old`.

```
declare i32 @llvm.nvvm.atomic.load.inc.32.p0i32(i32* address, i32 val)
declare i32 @llvm.nvvm.atomic.load.inc.32.p1i32(i32 addrspace(1)* address, i32 val)
declare i32 @llvm.nvvm.atomic.load.inc.32.p3i32(i32 addrspace(3)* address, i32 val)
```

reads the 32-bit word `old` located at the address `address`, computes `((old >= val) ? 0 : (old+1))`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

```
declare i32 @llvm.nvvm.atomic.load.dec.32.p0i32(i32* address, i32 val)
declare i32 @llvm.nvvm.atomic.load.dec.32.p1i32(i32 addrspace(1)* address, i32 val)
declare i32 @llvm.nvvm.atomic.load.dec.32.p3i32(i32 addrspace(3)* address, i32 val)
```

reads the 32-bit word `old` located at the address `address`, computes `(((old == 0) | (old > val)) ? val : (old-1) )`, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns `old`.

14.2. Barrier and Memory Fence

```
declare void @llvm.nvvm.barrier0()
```

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `llvm.nvvm.barrier0()` are visible to all threads in the block.

```
declare i32 @llvm.nvvm.barrier0.popc(i32)
```

is identical to `llvm.nvvm.barrier0()` with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.

```
declare i32 @llvm.nvvm.barrier0.and(i32)
```

is identical to `llvm.nvvm.barrier0()` with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for all of them.

```
declare i32 @llvm.nvvm.barrier0.or(i32)
```

is identical to `llvm.nvvm.barrier0()` with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for any of them.

```
declare void @llvm.nvvm.cluster.barrier(i32 %flags)
```

Synchronize and communicate among threads in the same cluster. This intrinsic is only supported for Hopper+. The %flags is encoded according to the following table:

%flags bits	Meaning
31-8	Reserved
7-4	Memory ordering (See Cluster Barrier Memory Ordering Encoding below)
3-0	Operation mode (See Cluster Barrier Operation Mode Encoding below)

Cluster Barrier Operation Mode Encoding

Encoding	Mode	Description
0	Arrive	Arrive at cluster barrier
1	Wait	Wait at cluster barrier
2-15	RESERVED	RESERVED

Cluster Barrier Memory Ordering Encoding

Encoding	Mode	Description
0	Default	All synchronous memory accesses requested by the executing entry prior to arrive are performed and are visible to all the entrys in the cluster after wait.
1	Relaxed	All previously fenced memory accesses requested by the executing entry prior to arrive are performed and are visible to all the entrys in the cluster after wait. This ordering is only supported when the operation mode is Arrive.
2-15	RESERVED	RESERVED

```
declare void @llvm.nvvm.membar.cta()
```

is a memory fence at the thread block level. This intrinsic is deprecated. Please use nvvm.membar with flags as argument instead.

```
declare void @llvm.nvvm.membar.gl()
```

is a memory fence at the device level. This intrinsic is deprecated. Please use nvvm.membar with flags as argument instead.

```
declare void @llvm.nvvm.membar.sys()
```

is a memory fence at the system level. This intrinsic is deprecated. Please use nvvm.membar with flags as argument instead.

```
declare void @llvm.nvvm.membar(i32 %flags)
```

Wait for all prior memory accesses requested by this thread to be performed at a membar level defined by the membar mode below. The memory barrier enforces vertical ordering only. It makes no guarantees as to execution synchronization with other threads. For horizontal synchronization, a barrier should be used instead, or in addition to membar.

The %flags is encoded according to the following table:

%flags bits	Meaning
31-4	Reserved
3-0	Membar modes (See Membar Mode Encoding.)

Membar Mode Encoding

Encoding	Mode	Description
0	GLOBAL	Membar at the global level
1	CTA	Membar at the CTA level
2	SYSTEM	Membar at the system level
3	RESERVED	RESERVED
4	CLUSTER	Membar at the cluster level, only on Hopper+
5-15	RESERVED	RESERVED

14.3. Address space conversion

Note: Attention: Please use the `addrspacecast` IR instruction for address space conversion.

14.4. Special Registers

The following intrinsic functions are provided to support reading special PTX registers:

```
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.warpsize()
```

14.5. Texture/Surface Access

The following intrinsic function is provided to convert a global texture/surface variable into a texture/surface handle.

```
declare i64 %llvm.nvvm.texsurf.handle.p1i64(metadata, i64 addrspace(1)*)
```

See Accessing Texture Memory or Surface Memory for details.

The following IR definitions apply to all intrinsics in this section:

```
type %float4 = { float, float, float, float }
type %long2 = { i64, i64 }
type %int4 = { i32, i32, i32, i32 }
type %int2 = { i32, i32 }
type %short4 = { i16, i16, i16, i16 }
type %short2 = { i16, i16 }
```

14.5.1. Texture Reads

Sampling a 1D texture:

```
%float4 @llvm.nvvm.tex.unified.1d.v4f32.s32(i64 %tex, i32 %x)
%float4 @llvm.nvvm.tex.unified.1d.v4f32.f32(i64 %tex, float %x)
%float4 @llvm.nvvm.tex.unified.1d.level.v4f32.f32(i64 %tex, float %x,
                                                  float %level)
%float4 @llvm.nvvm.tex.unified.1d.grad.v4f32.f32(i64 %tex, float %x,
                                                 float %dPdx,
                                                 float %dPdy)

%int4 @llvm.nvvm.tex.unified.1d.v4s32.s32(i64 %tex, i32 %x)
%int4 @llvm.nvvm.tex.unified.1d.v4s32.f32(i64 %tex, float %x)
%int4 @llvm.nvvm.tex.unified.1d.level.v4s32.f32(i64 %tex, float %x,
                                                float %level)
%int4 @llvm.nvvm.tex.unified.1d.grad.v4s32.f32(i64 %tex, float %x,
                                               float %dPdx,
                                               float %dPdy)

%int4 @llvm.nvvm.tex.unified.1d.v4u32.s32(i64 %tex, i32 %x)
%int4 @llvm.nvvm.tex.unified.1d.v4u32.f32(i64 %tex, float %x)
%int4 @llvm.nvvm.tex.unified.1d.level.v4u32.f32(i64 %tex, float %x,
                                                float %level)
%int4 @llvm.nvvm.tex.unified.1d.grad.v4u32.f32(i64 %tex, float %x,
                                               float %dPdx,
                                               float %dPdy)
```

Sampling a 1D texture array:

```
%float4 @llvm.nvvm.tex.unified.1d.array.v4f32.s32(i64 %tex, i32 %idx, i32 %x)
%float4 @llvm.nvvm.tex.unified.1d.array.v4f32.f32(i64 %tex, i32 %idx, float %x)
%float4 @llvm.nvvm.tex.unified.1d.array.level.v4f32.f32(i64 %tex, i32 %idx,
                                                        float %x,
                                                        float %level)
%float4 @llvm.nvvm.tex.unified.1d.array.grad.v4f32.f32(i64 %tex, i32 %idx,
                                                       float %x,
                                                       float %dPdx,
                                                       float %dPdy)

%int4 @llvm.nvvm.tex.unified.1d.array.v4s32.s32(i64 %tex, i32 %idx, i32 %x)
%int4 @llvm.nvvm.tex.unified.1d.array.v4s32.f32(i64 %tex, i32 %idx, float %x)
%int4 @llvm.nvvm.tex.unified.1d.array.level.v4s32.f32(i64 %tex, i32 %idx,
                                                      float %x,
                                                      float %level)
%int4 @llvm.nvvm.tex.unified.1d.array.grad.v4s32.f32(i64 %tex, i32 %idx,
                                                     float %x,
                                                     float %dPdx,
                                                     float %dPdy)

%int4 @llvm.nvvm.tex.unified.1d.array.v4u32.s32(i64 %tex, i32 %idx, i32 %x)
%int4 @llvm.nvvm.tex.unified.1d.array.v4u32.f32(i64 %tex, i32 %idx, float %x)
%int4 @llvm.nvvm.tex.unified.1d.array.level.v4u32.f32(i64 %tex, i32 %idx,
                                                      float %x,
                                                      float %level)
%int4 @llvm.nvvm.tex.unified.1d.array.grad.v4u32.f32(i64 %tex, i32 %idx,
                                                     float %x,
```

(continues on next page)

(continued from previous page)

```
                                                        float %dPdx,
                                                        float %dPdy)
```

Sampling a 2D texture:

```
%float4 @llvm.nvvm.tex.unified.2d.v4f32.s32(i64 %tex, i32 %x, i32 %y)
%float4 @llvm.nvvm.tex.unified.2d.v4f32.f32(i64 %tex, float %x, float %y)
%float4 @llvm.nvvm.tex.unified.2d.level.v4f32.f32(i64 %tex, float %x, float %y,
                                                   float %level)
%float4 @llvm.nvvm.tex.unified.2d.grad.v4f32.f32(i64 %tex, float %x, float %y,
                                                  float %dPdx_x, float %dPdx_y,
                                                  float %dPdy_x, float %dPdy_y)

%int4 @llvm.nvvm.tex.unified.2d.v4s32.s32(i64 %tex, i32 %x, i32 %y)
%int4 @llvm.nvvm.tex.unified.2d.v4s32.f32(i64 %tex, float %x, float %y,)
%int4 @llvm.nvvm.tex.unified.2d.level.v4s32.f32(i64 %tex, float %x, float %y,
                                                 float %level)
%int4 @llvm.nvvm.tex.unified.2d.grad.v4s32.f32(i64 %tex, float %x, float %y,
                                                float %dPdx_x, float %dPdx_y,
                                                float %dPdy_x, float %dPdy_y)

%int4 @llvm.nvvm.tex.unified.2d.v4u32.s32(i64 %tex, i32 %x i32 %y)
%int4 @llvm.nvvm.tex.unified.2d.v4u32.f32(i64 %tex, float %x float %y)
%int4 @llvm.nvvm.tex.unified.2d.level.v4u32.f32(i64 %tex, float %x, float %y,
                                                 float %level)
%int4 @llvm.nvvm.tex.unified.2d.grad.v4u32.f32(i64 %tex, float %x, float %y,
                                                float %dPdx_x, float %dPdx_y,
                                                float %dPdy_x, float %dPdy_y)
```

Sampling a 2D texture array:

```
%float4 @llvm.nvvm.tex.unified.2d.array.v4f32.s32(i64 %tex, i32 %idx,
                                                   i32 %x, i32 %y)
%float4 @llvm.nvvm.tex.unified.2d.array.v4f32.f32(i64 %tex, i32 %idx,
                                                   float %x, float %y)
%float4 @llvm.nvvm.tex.unified.2d.array.level.v4f32.f32(i64 %tex, i32 %idx,
                                                         float %x, float %y,
                                                         float %level)
%float4 @llvm.nvvm.tex.unified.2d.array.grad.v4f32.f32(i64 %tex, i32 %idx,
                                                        float %x, float %y,
                                                        float %dPdx_x,
                                                        float %dPdx_y,
                                                        float %dPdy_x,
                                                        float %dPdy_y)

%int4 @llvm.nvvm.tex.unified.2d.array.v4s32.s32(i64 %tex, i32 %idx,
                                                 i32 %x, i32 %y)
%int4 @llvm.nvvm.tex.unified.2d.array.v4s32.f32(i64 %tex, i32 %idx,
                                                 float %x, float %y)
%int4 @llvm.nvvm.tex.unified.2d.array.level.v4s32.f32(i64 %tex, i32 %idx,
                                                       float %x, float %y,
                                                       float %level)
%int4 @llvm.nvvm.tex.unified.2d.array.grad.v4s32.f32(i64 %tex, i32 %idx,
                                                      float %x, float %y,
                                                      float %dPdx_x,
                                                      float %dPdx_y,
                                                      float %dPdy_x,
```

(continues on next page)

(continued from previous page)

```
                                                        float %dPdy_y)

%int4 @llvm.nvvm.tex.unified.2d.array.v4u32.s32(i64 %tex, i32 %idx,
                                                i32 %x i32 %y)
%int4 @llvm.nvvm.tex.unified.2d.array.v4u32.f32(i64 %tex, i32 %idx,
                                                float %x float %y)
%int4 @llvm.nvvm.tex.unified.2d.array.level.v4u32.f32(i64 %tex, i32 %idx,
                                                      float %x, float %y,
                                                      float %level)
%int4 @llvm.nvvm.tex.unified.2d.array.grad.v4u32.f32(i64 %tex, i32 %idx,
                                                     float %x, float %y,
                                                     float %dPdx_x,
                                                     float %dPdx_y,
                                                     float %dPdy_x,
                                                     float %dPdy_y)
```

Sampling a 3D texture:

```
%float4 @llvm.nvvm.tex.unified.3d.v4f32.s32(i64 %tex, i32 %x, i32 %y, i32 %z)
%float4 @llvm.nvvm.tex.unified.3d.v4f32.f32(i64 %tex, float %x, float %y,
                                            float %z)
%float4 @llvm.nvvm.tex.unified.3d.level.v4f32.f32(i64 %tex,float %x, float %y,
                                                  float %z, float %level)
%float4 @llvm.nvvm.tex.unified.3d.grad.v4f32.f32(i64 %tex, float %x, float %y,
                                                 float %z, float %dPdx_x,
                                                 float %dPdx_y, float %dPdx_z,
                                                 float %dPdy_x, float %dPdy_y,
                                                 float %dPdy_z)

%int4 @llvm.nvvm.tex.unified.3d.v4s32.s32(i64 %tex, i32 %x, i32 %y, i32 %z)
%int4 @llvm.nvvm.tex.unified.3d.v4s32.f32(i64 %tex, float %x, float %y,
                                          float %z)
%int4 @llvm.nvvm.tex.unified.3d.level.v4s32.f32(i64 %tex, float %x, float %y,
                                                float %z, float %level)
%int4 @llvm.nvvm.tex.unified.3d.grad.v4s32.f32(i64 %tex, float %x, float %y,
                                               float %z, float %dPdx_x,
                                               float %dPdx_y, float %dPdx_z,
                                               float %dPdy_x, float %dPdy_y,
                                               float %dPdy_z)

%int4 @llvm.nvvm.tex.unified.3d.v4u32.s32(i64 %tex, i32 %x i32 %y, i32 %z)
%int4 @llvm.nvvm.tex.unified.3d.v4u32.f32(i64 %tex, float %x, float %y,
                                          float %z)
%int4 @llvm.nvvm.tex.unified.3d.level.v4u32.f32(i64 %tex, float %x, float %y,
                                                float %z, float %level)
%int4 @llvm.nvvm.tex.unified.3d.grad.v4u32.f32(i64 %tex, float %x, float %y,
                                               float %z, float %dPdx_x,
                                               float %dPdx_y, float %dPdx_z,
                                               float %dPdy_x, float %dPdy_y,
                                               float %dPdy_z)
```

Sampling a cube texture:

```
%float4 @llvm.nvvm.tex.unified.cube.v4f32.f32(i64 %tex, float %x, float %y,
                                              float %z)
%float4 @llvm.nvvm.tex.unified.cube.level.v4f32.f32(i64 %tex,float %x, float %y,
                                                    float %z, float %level)
```

(continues on next page)

(continued from previous page)

```
%int4 @llvm.nvvm.tex.unified.cube.v4s32.f32(i64 %tex, float %x, float %y,
                                            float %z)
%int4 @llvm.nvvm.tex.unified.cube.level.v4s32.f32(i64 %tex, float %x, float %y,
                                                  float %z, float %level)

%int4 @llvm.nvvm.tex.unified.cube.v4u32.f32(i64 %tex, float %x, float %y,
                                            float %z)
%int4 @llvm.nvvm.tex.unified.cube.level.v4u32.f32(i64 %tex, float %x, float %y,
                                                  float %z, float %level)
```

Sampling a cube texture array:

```
%float4 @llvm.nvvm.tex.unified.cube.array.v4f32.f32(i64 %tex, i32 %idx,
                                                    float %x, float %y,
                                                    float %z)
%float4 @llvm.nvvm.tex.unified.cube.array.level.v4f32.f32(i64 %tex, i32 %idx,
                                                          float %x, float %y,
                                                          float %z,
                                                          float %level)

%int4 @llvm.nvvm.tex.unified.cube.array.v4s32.f32(i64 %tex, i32 %idx, float %x,
                                                  float %y, float %z)
%int4 @llvm.nvvm.tex.unified.cube.array.level.v4s32.f32(i64 %tex, i32 %idx,
                                                        float %x, float %y,
                                                        float %z, float %level)

%int4 @llvm.nvvm.tex.unified.cube.array.v4u32.f32(i64 %tex, i32 %idx, float %x,
                                                  float %y, float %z)
%int4 @llvm.nvvm.tex.unified.cube.array.level.v4u32.f32(i64 %tex, i32 %idx,
                                                        float %x, float %y,
                                                        float %z, float %level)
```

Fetching a four-texel bilerp footprint:

```
%float4 @llvm.nvvm.tld4.unified.r.2d.v4f32.f32(i64 %tex, float %x, float %y)
%float4 @llvm.nvvm.tld4.unified.g.2d.v4f32.f32(i64 %tex, float %x, float %y)
%float4 @llvm.nvvm.tld4.unified.b.2d.v4f32.f32(i64 %tex, float %x, float %y)
%float4 @llvm.nvvm.tld4.unified.a.2d.v4f32.f32(i64 %tex, float %x, float %y)

%int4 @llvm.nvvm.tld4.unified.r.2d.v4s32.f32(i64 %tex, float %x, float %y)
%int4 @llvm.nvvm.tld4.unified.g.2d.v4s32.f32(i64 %tex, float %x, float %y)
%int4 @llvm.nvvm.tld4.unified.b.2d.v4s32.f32(i64 %tex, float %x, float %y)
%int4 @llvm.nvvm.tld4.unified.a.2d.v4s32.f32(i64 %tex, float %x, float %y)

%int4 @llvm.nvvm.tld4.unified.r.2d.v4u32.f32(i64 %tex, float %x, float %y)
%int4 @llvm.nvvm.tld4.unified.g.2d.v4u32.f32(i64 %tex, float %x, float %y)
%int4 @llvm.nvvm.tld4.unified.b.2d.v4u32.f32(i64 %tex, float %x, float %y)
%int4 @llvm.nvvm.tld4.unified.a.2d.v4u32.f32(i64 %tex, float %x, float %y)
```

14.5.2. Surface Loads

In the following intrinsics, `<clamp>` represents the surface clamp mode and can be one of the following: `clamp`, `trap`, or `zero`.

For surface load instructions that operate on 8-bit data channels, the output operands are of type `i16`. The high-order eight bits are undefined.

Reading a 1D surface:

```
i16 @llvm.nvvm.suld.1d.i8.<clamp>(i64 %tex, i32 %x)
i16 @llvm.nvvm.suld.1d.i16.<clamp>(i64 %tex, i32 %x)
i32 @llvm.nvvm.suld.1d.i32.<clamp>(i64 %tex, i32 %x)
i64 @llvm.nvvm.suld.1d.i64.<clamp>(i64 %tex, i32 %x)

%short2 @llvm.nvvm.suld.1d.v2i8.<clamp>(i64 %tex, i32 %x)
%short2 @llvm.nvvm.suld.1d.v2i16.<clamp>(i64 %tex, i32 %x)
%int2 @llvm.nvvm.suld.1d.v2i32.<clamp>(i64 %tex, i32 %x)
%long2 @llvm.nvvm.suld.1d.v2i64.<clamp>(i64 %tex, i32 %x)

%short4 @llvm.nvvm.suld.1d.v4i8.<clamp>(i64 %tex, i32 %x)
%short4 @llvm.nvvm.suld.1d.v4i16.<clamp>(i64 %tex, i32 %x)
%int4 @llvm.nvvm.suld.1d.v4i32.<clamp>(i64 %tex, i32 %x)
```

Reading a 1D surface array:

```
i16 @llvm.nvvm.suld.1d.array.i8.<clamp>(i64 %tex, i32 %idx, i32 %x)
i16 @llvm.nvvm.suld.1d.array.i16.<clamp>(i64 %tex, i32 %idx, i32 %x)
i32 @llvm.nvvm.suld.1d.array.i32.<clamp>(i64 %tex, i32 %idx, i32 %x)
i64 @llvm.nvvm.suld.1d.array.i64.<clamp>(i64 %tex, i32 %idx, i32 %x)

%short2 @llvm.nvvm.suld.1d.array.v2i8.<clamp>(i64 %tex, i32 %idx, i32 %x)
%short2 @llvm.nvvm.suld.1d.array.v2i16.<clamp>(i64 %tex, i32 %idx, i32 %x)
%int2 @llvm.nvvm.suld.1d.array.v2i32.<clamp>(i64 %tex, i32 %idx, i32 %x)
%long2 @llvm.nvvm.suld.1d.array.v2i64.<clamp>(i64 %tex, i32 %idx, i32 %x)

%short4 @llvm.nvvm.suld.1d.array.v4i8.<clamp>(i64 %tex, i32 %idx, i32 %x)
%short4 @llvm.nvvm.suld.1d.array.v4i16.<clamp>(i64 %tex, i32 %idx, i32 %x)
%int4 @llvm.nvvm.suld.1d.array.v4i32.<clamp>(i64 %tex, i32 %idx, i32 %x)
```

Reading a 2D surface:

```
i16 @llvm.nvvm.suld.2d.i8.<clamp>(i64 %tex, i32 %x, i32 %y)
i16 @llvm.nvvm.suld.2d.i16.<clamp>(i64 %tex, i32 %x, i32 %y)
i32 @llvm.nvvm.suld.2d.i32.<clamp>(i64 %tex, i32 %x, i32 %y)
i64 @llvm.nvvm.suld.2d.i64.<clamp>(i64 %tex, i32 %x, i32 %y)

%short2 @llvm.nvvm.suld.2d.v2i8.<clamp>(i64 %tex, i32 %x, i32 %y)
%short2 @llvm.nvvm.suld.2d.v2i16.<clamp>(i64 %tex, i32 %x, i32 %y)
%int2 @llvm.nvvm.suld.2d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %y)
%long2 @llvm.nvvm.suld.2d.v2i64.<clamp>(i64 %tex, i32 %x, i32 %y)

%short4 @llvm.nvvm.suld.2d.v4i8.<clamp>(i64 %tex, i32 %x, i32 %y)
%short4 @llvm.nvvm.suld.2d.v4i16.<clamp>(i64 %tex, i32 %x, i32 %y)
%int4 @llvm.nvvm.suld.2d.v4i32.<clamp>(i64 %tex, i32 %x, i32 %y)
```

Reading a 2D surface array:

```
i16 @llvm.nvvm.suld.2d.array.i8.<clamp>(i64 %tex, i32 %idx, i32 %x, i32 %y)
i16 @llvm.nvvm.suld.2d.array.i16.<clamp>(i64 %tex, i32 %idx, i32 %x, i32 %y)
i32 @llvm.nvvm.suld.2d.array.i32.<clamp>(i64 %tex, i32 %idx, i32 %x, i32 %y)
i64 @llvm.nvvm.suld.2d.array.i64.<clamp>(i64 %tex, i32 %idx, i32 %x, i32 %y)

%short2 @llvm.nvvm.suld.2d.array.v2i8.<clamp>(i64 %tex, i32 %idx,
                                              i32 %x, i32 %y)
%short2 @llvm.nvvm.suld.2d.array.v2i16.<clamp>(i64 %tex, i32 %idx,
                                               i32 %x, i32 %y)
%int2 @llvm.nvvm.suld.2d.array.v2i32.<clamp>(i64 %tex, i32 %idx,
                                             i32 %x, i32 %y)
%long2 @llvm.nvvm.suld.2d.array.v2i64.<clamp>(i64 %tex, i32 %idx,
                                              i32 %x, i32 %y)

%short4 @llvm.nvvm.suld.2d.array.v4i8.<clamp>(i64 %tex, i32 %idx,
                                              i32 %x, i32 %y)
%short4 @llvm.nvvm.suld.2d.array.v4i16.<clamp>(i64 %tex, i32 %idx,
                                               i32 %x, i32 %y)
%int4 @llvm.nvvm.suld.2d.array.v4i32.<clamp>(i64 %tex, i32 %idx,
                                             i32 %x, i32 %y)
```

Reading a 3D surface:

```
i16 @llvm.nvvm.suld.3d.i8.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)
i16 @llvm.nvvm.suld.3d.i16.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)
i32 @llvm.nvvm.suld.3d.i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)
i64 @llvm.nvvm.suld.3d.i64.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)

%short2 @llvm.nvvm.suld.3d.v2i8.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)
%short2 @llvm.nvvm.suld.3d.v2i16.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)
%int2 @llvm.nvvm.suld.3d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)
%long2 @llvm.nvvm.suld.3d.v2i64.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z)

%short4 @llvm.nvvm.suld.3d.v4i8.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i32 %z)
%short4 @llvm.nvvm.suld.3d.v4i16.<clamp>(i64 %tex, i32 %x, i32 %y,
                                         i32 %z)
%int4 @llvm.nvvm.suld.3d.v4i32.<clamp>(i64 %tex, i32 %x, i32 %y,
                                       i32 %z)
```

14.5.3. Surface Stores

In the following intrinsics, `<clamp>` represents the surface clamp mode. It is `trap` for the formatted stores, and can be one of the following for unformatted stores: `clamp`, `trap`, or `zero`.

For surface store instructions that operate on 8-bit data channels, the input operands are of type `i16`. The high-order eight bits are ignored.

Writing a 1D surface:

```
;; Unformatted
void @llvm.nvvm.sust.b.1d.i8.<clamp>(i64 %tex, i32 %x, i16 %r)
void @llvm.nvvm.sust.b.1d.i16.<clamp>(i64 %tex, i32 %x, i16 %r)
void @llvm.nvvm.sust.b.1d.i32.<clamp>(i64 %tex, i32 %x, i32 %r)
void @llvm.nvvm.sust.b.1d.i64.<clamp>(i64 %tex, i32 %x, i64 %r)
```

(continues on next page)

(continued from previous page)

```
void @llvm.nvvm.sust.b.1d.v2i8.<clamp>(i64 %tex, i32 %x, i16 %r, i16 %g)
void @llvm.nvvm.sust.b.1d.v2i16.<clamp>(i64 %tex, i32 %x, i16 %r, i16 %g)
void @llvm.nvvm.sust.b.1d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %r, i32 %g)
void @llvm.nvvm.sust.b.1d.v2i64.<clamp>(i64 %tex, i32 %x, i64 %r, i64 %g)

void @llvm.nvvm.sust.b.1d.v4i8.<clamp>(i64 %tex, i32 %x,
                                       i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.1d.v4i16.<clamp>(i64 %tex, i32 %x,
                                        i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.1d.v4i32.<clamp>(i64 %tex, i32 %x,
                                        i32 %r, i32 %g, i32 %b, i32 %a)

;; Formatted
void @llvm.nvvm.sust.p.1d.i32.<clamp>(i64 %tex, i32 %x, i32 %r)

void @llvm.nvvm.sust.p.1d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %r, i32 %g)

void @llvm.nvvm.sust.p.1d.v4i32.<clamp>(i64 %tex, i32 %x,
                                        i32 %r, i32 %g, i32 %b, i32 %a)
```

Writing a 1D surface array:

```
;; Unformatted
void @llvm.nvvm.sust.b.1d.array.i8.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i16 %r)
void @llvm.nvvm.sust.b.1d.array.i16.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                     i16 %r)
void @llvm.nvvm.sust.b.1d.array.i32.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                     i32 %r)
void @llvm.nvvm.sust.b.1d.array.i64.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                     i64 %r)

void @llvm.nvvm.sust.b.1d.array.v2i8.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                   i16 %r, i16 %g)
void @llvm.nvvm.sust.b.1d.array.v2i16.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i16 %r, i16 %g)
void @llvm.nvvm.sust.b.1d.array.v2i32.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i32 %r, i32 %g)
void @llvm.nvvm.sust.b.1d.array.v2i64.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i64 %r, i64 %g)

void @llvm.nvvm.sust.b.1d.array.v4i8.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                  i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.1d.array.v4i16.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.1d.array.v4i32.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i32 %r, i32 %g, i32 %b, i32 %a)

;; Formatted
void @llvm.nvvm.sust.p.1d.array.i32.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                     i32 %r)

void @llvm.nvvm.sust.p.1d.array.v2i32.<clamp>(i64 %tex, i32 %idx, i32 %x,
                                                    i32 %r, i32 %g)

void @llvm.nvvm.sust.p.1d.array.v4i32.<clamp>(i64 %tex, i32 %idx, i32 %x,
```

(continues on next page)

(continued from previous page)

```
                                                i32 %r, i32 %g, i32 %b, i32 %a)
```

Writing a 2D surface:

```
;; Unformatted
void @llvm.nvvm.sust.b.2d.i8.<clamp>(i64 %tex, i32 %x, i32 %y, i16 %r)
void @llvm.nvvm.sust.b.2d.i16.<clamp>(i64 %tex, i32 %x, i32 %y, i16 %r)
void @llvm.nvvm.sust.b.2d.i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %r)
void @llvm.nvvm.sust.b.2d.i64.<clamp>(i64 %tex, i32 %x, i32 %y, i64 %r)

void @llvm.nvvm.sust.b.2d.v2i8.<clamp>(i64 %tex, i32 %x, i32 %y,
                                       i16 %r, i16 %g)
void @llvm.nvvm.sust.b.2d.v2i16.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i16 %r, i16 %g)
void @llvm.nvvm.sust.b.2d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i32 %r, i32 %g)
void @llvm.nvvm.sust.b.2d.v2i64.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i64 %r, i64 %g)

void @llvm.nvvm.sust.b.2d.v4i8.<clamp>(i64 %tex, i32 %x, i32 %y,
                                       i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.2d.v4i16.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.2d.v4i32.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i32 %r, i32 %g, i32 %b, i32 %a)

;; Formatted
void @llvm.nvvm.sust.p.2d.i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %r)

void @llvm.nvvm.sust.p.2d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i32 %r, i32 %g)

void @llvm.nvvm.sust.p.2d.v4i32.<clamp>(i64 %tex, i32 %x, i32 %y,
                                        i32 %r, i32 %g, i32 %b, i32 %a)
```

Writing a 2D surface array:

```
;; Unformatted
void @llvm.nvvm.sust.b.2d.array.i8.<clamp>(i64 %tex, i32 %idx,
                                           i32 %x, i32 %y, i16 %r)
void @llvm.nvvm.sust.b.2d.array.i16.<clamp>(i64 %tex, i32 %idx,
                                            i32 %x, i32 %y, i16 %r)
void @llvm.nvvm.sust.b.2d.array.i32.<clamp>(i64 %tex, i32 %idx,
                                            i32 %x, i32 %y, i32 %r)
void @llvm.nvvm.sust.b.2d.array.i64.<clamp>(i64 %tex, i32 %idx,
                                            i32 %x, i32 %y, i64 %r)

void @llvm.nvvm.sust.b.2d.array.v2i8.<clamp>(i64 %tex, i32 %idx,
                                             i32 %x, i32 %y,
                                             i16 %r, i16 %g)
void @llvm.nvvm.sust.b.2d.array.v2i16.<clamp>(i64 %tex, i32 %idx,
                                              i32 %x, i32 %y,
                                              i16 %r, i16 %g)
void @llvm.nvvm.sust.b.2d.array.v2i32.<clamp>(i64 %tex, i32 %idx,
                                              i32 %x, i32 %y,
                                              i32 %r, i32 %g)
void @llvm.nvvm.sust.b.2d.array.v2i64.<clamp>(i64 %tex, i32 %idx,
```

(continues on next page)

(continued from previous page)

```
                                                i32 %x, i32 %y,
                                                i64 %r, i64 %g)

void @llvm.nvvm.sust.b.2d.array.v4i8.<clamp>(i64 %tex, i32 %idx,
                                               i32 %x, i32 %y,
                                               i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.2d.array.v4i16.<clamp>(i64 %tex, i32 %idx,
                                                i32 %x, i32 %y,
                                                i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.2d.array.v4i32.<clamp>(i64 %tex, i32 %idx,
                                                i32 %x, i32 %y,
                                                i32 %r, i32 %g, i32 %b, i32 %a)

;; Formatted
void @llvm.nvvm.sust.p.2d.array.i32.<clamp>(i64 %tex, i32 %idx,
                                              i32 %x, i32 %y, i32 %r)

void @llvm.nvvm.sust.p.2d.array.v2i32.<clamp>(i64 %tex, i32 %idx,
                                                i32 %x, i32 %y,
                                                i32 %r, i32 %g)

void @llvm.nvvm.sust.p.2d.array.v4i32.<clamp>(i64 %tex, i32 %idx,
                                                i32 %x, i32 %y,
                                                i32 %r, i32 %g, i32 %b, i32 %a)
```

Writing a 3D surface:

```
;; Unformatted
void @llvm.nvvm.sust.b.3d.i8.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z, i16 %r)
void @llvm.nvvm.sust.b.3d.i16.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z, i16 %r)
void @llvm.nvvm.sust.b.3d.i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z, i32 %r)
void @llvm.nvvm.sust.b.3d.i64.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z, i64 %r)

void @llvm.nvvm.sust.b.3d.v2i8.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                       i16 %r, i16 %g)
void @llvm.nvvm.sust.b.3d.v2i16.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i16 %r, i16 %g)
void @llvm.nvvm.sust.b.3d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i32 %r, i32 %g)
void @llvm.nvvm.sust.b.3d.v2i64.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i64 %r, i64 %g)

void @llvm.nvvm.sust.b.3d.v4i8.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                       i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.3d.v4i16.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i16 %r, i16 %g, i16 %b, i16 %a)
void @llvm.nvvm.sust.b.3d.v4i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i32 %r, i32 %g, i32 %b, i32 %a)

;; Formatted
void @llvm.nvvm.sust.p.3d.i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z, i32 %r)

void @llvm.nvvm.sust.p.3d.v2i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i32 %r, i32 %g)

void @llvm.nvvm.sust.p.3d.v4i32.<clamp>(i64 %tex, i32 %x, i32 %y, i32 %z,
                                        i32 %r, i32 %g, i32 %b, i32 %a)
```

14.6. Warp-level Operations

14.6.1. Barrier Synchronization

The following intrinsic performs a barrier synchronization among a subset of threads in a warp.

```
declare void @llvm.nvvm.bar.warp.sync(i32 %membermask)
```

This intrinsic causes executing thread to wait until all threads corresponding to `%membermask` have executed the same intrinsic with the same `%membermask` value before resuming execution.

The argument `%membership` is a 32bit mask, with each bit corresponding to a lane in the warp. 1 means the thread is in the subset.

The behavior of this intrinsic is undefined if the executing thread is not in the `%membermask`.

14.6.2. Data Movement

The following intrinsics synchronize a subset of threads in a warp and then perform data movement among these threads. Note that the old intrinsic with the `%mode` parameter is deprecated on the modern NVVM IR dialect.

```
; New intrinsics for the modern dialect
declare {i32, i1} @llvm.nvvm.shfl.sync.idx.i32p(i32 %membermask, i32 %a, i32 %b, i32
↪%c)
declare {i32, i1} @llvm.nvvm.shfl.sync.up.i32p(i32 %membermask, i32 %a, i32 %b, i32
↪%c)
declare {i32, i1} @llvm.nvvm.shfl.sync.down.i32p(i32 %membermask, i32 %a, i32 %b, i32
↪%c)
declare {i32, i1} @llvm.nvvm.shfl.sync.bfly.i32p(i32 %membermask, i32 %a, i32 %b, i32
↪%c)

; Old intrinsic for the LLVM 7 dialect
declare {i32, i1} @llvm.nvvm.shfl.sync.i32(i32 %membermask, i32 %mode, i32 %a, i32 %b,
↪ i32 %c)
```

This intrinsic causes executing thread to wait until all threads corresponding to `%membermask` have executed the same intrinsic with the same `%membermask` value before reading data from other threads in the same warp.

The argument `%membership` is a 32bit mask, with each bit corresponding to a lane in the warp. 1 means the thread is in the subset.

Each thread in the currently executing warp will compute a source lane index j based on input arguments %b, %c, and the shuffle mode (implicit for new intrinsics, explicit via %mode for the old intrinsic). If the computed source lane index j is in range, the returned `i32` value will be the value of %a from lane j; otherwise, it will be the the value of %a from the current thread. If the thread corresponding to lane j is inactive, then the returned `i32` value is undefined. The returned `i1` value is set to 1 if the source lane j is in range, and otherwise set to 0.

In the LLVM 7 dialect intrinsic, the argument `%mode` must be a constant and its encoding is specified in the following table.

Encoding	Meaning	Corresponding Suffix
0	IDX	.idx
1	UP	.up
2	DOWN	.down
3	BFLY	.bfly

Argument %b specifies a source lane or source lane offset, depending on %mode.

Argument %c contains two packed values specifying a mask for logically splitting warps into sub-segments and an upper bound for clamping the source lane index.

The following pseudo code illustrates the semantics of this intrinsic. The %mode in the switch would be determined by the intrinsic suffix in the modern dialect intrinsics and by the %mode parameter in the LLVM 7 dialect intrinsic.

```
wait until all threads in %membermask have arrived;

%lane[4:0] = current_lane_id; // position of thread in warp
%bval[4:0] = %b[4:0]; // source lane or lane offset (0..31)
%cval[4:0] = %c[4:0]; // clamp value
%mask[4:0] = %c[12:8];

%maxLane = (%lane[4:0] & %mask[4:0]) | (%cval[4:0] & ~%mask[4:0]);
%minLane = (%lane[4:0] & %mask[4:0]);
switch (%mode) {
case UP: %j = %lane - %bval; %pval = (%j >= %maxLane); break;
case DOWN: %j = %lane + %bval; %pval = (%j <= %maxLane); break;
case BFLY: %j = %lane ^ %bval; %pval = (%j <= %maxLane); break;
case IDX: %j = %minLane | (%bval[4:0] & ~%mask[4:0]); %pval = (%j <= %maxLane); break;
}
if (!%pval) %j = %lane; // copy from own lane
if (thread at lane %j is active)
   %d = %a from lane %j
else
   %d = undef
return {%d, %pval}
```

Note that the return values are undefined if the thread at the source lane is not in %membermask.

The behavior of this intrinsic is undefined if the executing thread is not in the %membermask.

14.6.3. Vote

The following intrinsic synchronizes a subset of threads in a warp and then performs a reduce-and-broadcast of a predicate over all threads in the subset.

```
declare {i32, i1} @llvm.nvvm.vote.sync(i32 %membermask, i32 %mode, i1 %predicate)
```

This intrinsic causes executing thread to wait until all threads corresponding to %membermask have executed the same intrinsic with the same %membermask value before performing a reduce-and-broadcast of a predicate over all threads in the subset.

The argument `%membermask` is a 32-bit mask, with each bit corresponding to a lane in the warp. 1 means the thread is in the subset.

`@llvm.nvvm.vote.sync()` performs a reduction of the source `%predicate` across all threads in `%membermask` after the synchronization. The return value is the same across all threads in the `%membermask`. The element in the returned aggregate that holds the return value depends on `%mode`.

The argument `%mode` must be a constant and its encoding is specified in the following table.

Encoding	Meaning	return value
0	ALL	`i1` : 1 if the source predicates is 1 for all thread in `%membermask`, 0 otherwise
1	ANY	`i1` : 1 if the source predicate is 1 for any thread in `%membermask`, 0 otherwise
2	EQ	`i1` : 1 if the source predicates are the same for all thread in `%membermask`, 0 otherwise
3	BALLOT	`i32` : ballot data, containing the `%predicate` value from each thread in `%membermask`

For the `BALLOT` mode, the `i32` value represents the ballot data, which contains the `%predicate` value from each thread in `%membermask` in the bit position corresponding to the thread's land id. The bit value corresponding to a thread not in `%membermask` is 0.

Note that the return values are undefined if the thread at the source lane is not in `%membermask`.

The behavior of this intrinsic is undefined if the executing thread is not in the `%membermask`.

14.6.4. Match

The following intrinsics synchronize a subset of threads in a warp and then broadcast and compare a value across threads in the subset.

```
declare i32 @llvm.nvvm.match.any.sync.i32(i32 %membermask, i32 %value)
declare i32 @llvm.nvvm.match.any.sync.i64(i32 %membermask, i64 %value)
declare {i32, i1} @llvm.nvvm.match.all.sync.i32(i32 %membermask, i32 %value)
declare {i32, i1} @llvm.nvvm.match.all.sync.i64(i32 %membermask, i64 %value)
```

These intrinsics cause executing thread to wait until all threads corresponding to `%membermask` have executed the same intrinsic with the same `%membermask` value before performing broadcast and compare of operand `%value` across all threads in the subset.

The argument `%membership` is a 32bit mask, with each bit corresponding to a lane in the warp. 1 means the thread is in the subset.

The `i32` return value is a 32-bit mask where bit position in mask corresponds to thread's laneid.

In the `any` version, the `i32` return value is set to the mask of active threads in `%membermask` that have same value as operand `%value`.

In the `all` version, if all active threads in `%membermask` have same value as operand `%value`, the `i32` return value is set to `%membermask`, and the `i1` value is set to 1. Otherwise, the `i32` return value is set to 0 and the `i1` return value is also set to 0.

The behavior of this intrinsic is undefined if the executing thread is not in the `%membermask`.

14.6.5. Matrix Operation

THIS IS PREVIEW FEATURE. SUPPORT MAY BE REMOVED IN FUTURE RELEASES.

NVVM provides warp-level intrinsics for matrix multiply operations. The core operation is a matrix multiply and accumulate of the form:

```
D = A*B + C, or
C = A*B + C
```

where `A` is an `MxK` matrix, `B` is a `KxN` matrix, while `C` and `D` are `MxN` matrices. `C` and `D` are also called accumulators. The element type of the `A` and `B` matrices is 16-bit floating point. The element type of the accumulators can be either 32-bit floating point or 16-bit floating point.

All threads in a warp will collectively hold the contents of each matrix `A`, `B`, `C` and `D`. Each thread will hold only a fragment of matrix `A`, a fragment of matrix `B`, a fragment of matrix `C`, and a fragment of the result matrix `D`. How the elements of a matrix are distributed among the fragments is opaque to the user and is different for matrix `A`, `B` and the accumulator.

A fragment is represented by a sequence of element values. For fp32 matrices, the element type is `float`. For fp16 matrices, the element type is `i32` (each `i32` value holds two fp16 values). The number of elements varies with the shape of the matrix.

14.6.5.1 Load Fragments

The following intrinsics synchronize all threads in a warp and then load a fragment of a matrix for each thread.

```
; load fragment A
declare {i32, i32, i32, i32, i32, i32, i32, i32} @llvm.nvvm.hmma.m16n16k16.ld.a.p<n>
↪i32(i32 addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {i32, i32, i32, i32, i32, i32, i32, i32} @llvm.nvvm.hmma.m32n8k16.ld.a.p<n>
↪i32(i32 addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {i32, i32, i32, i32, i32, i32, i32, i32} @llvm.nvvm.hmma.m8n32k16.ld.a.p<n>
↪i32(i32 addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);

; load fragment B
declare {i32, i32, i32, i32, i32, i32, i32, i32} @llvm.nvvm.hmma.m16n16k16.ld.b.p<n>
↪i32(i32 addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {i32, i32, i32, i32, i32, i32, i32, i32} @llvm.nvvm.hmma.m32n8k16.ld.b.p<n>
↪i32(i32 addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {i32, i32, i32, i32, i32, i32, i32, i32} @llvm.nvvm.hmma.m8n32k16.ld.b.p<n>
↪i32(i32 addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);

; load fragment C
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m16n16k16.ld.c.f32.p<n>f32(float addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m32n8k16.ld.c.f32.p<n>f32(float addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m8n32k16.ld.c.f32.p<n>f32(float addrspace(<n>)* %ptr, i32 %ldm, i32 %rowcol);

; load fragment C
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m16n16k16.ld.c.f16.p<n>i32(i32 addrspace(
↪<n>)* %ptr, i32 %ldm, i32 %rowcol);
```

(continues on next page)

(continued from previous page)

```
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m32n8k16.ld.c.f16.p<n>i32(i32 addrspace(
↪<n>)* %ptr, i32 %ldm, i32 %rowcol);
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m8n32k16.ld.c.f16.p<n>i32(i32 addrspace(
↪<n>)* %ptr, i32 %ldm, i32 %rowcol);
```

These intrinsics load and return a matrix fragment from memory at location `%ptr`. The matrix in memory must be in a canonical matrix layout with leading dimension `%ldm`. `%rowcol` specifies which the matrix in memory is row-major (0) or column-major (1). `%rowcol` must be a constant value.

The returned sequence of values represent the fragment held by the calling thread. How the elements of a matrix are distributed among the fragments is opaque to the user and is different for matrix A, B and the accumulator. Therefore, three variants (i.e. `ld.a`, `ld.b`, and `ld.c`) are provided.

These intrinsics are overloaded based on the address spaces. The address space number <n> must be either 0 (generic), 1 (global) or 3 (shared).

The behavior of this intrinsic is undefined if any thread in the warp has exited.

14.6.5.2 Store Fragments

The following intrinsics synchronize all threads in a warp and then store a fragment of a matrix for each thread.

```
; The last 8 arguments are the elements of the C fragment
declare void @llvm.nvvm.hmma.m16n16k16.st.c.f32.p<n>float(float addrspace(<n>)* %ptr,
↪i32 %ldm, i32 %rowcol, float, float, float, float, float, float, float, float);
declare void @llvm.nvvm.hmma.m32n8k16.st.c.f32.p<n>float(float addrspace(<n>)* %ptr,
↪i32 %ldm, i32 %rowcol, float, float, float, float, float, float, float, float);
declare void @llvm.nvvm.hmma.m8n32k16.st.c.f32.p<n>float(float addrspace(<n>)* %ptr,
↪i32 %ldm, i32 %rowcol, float, float, float, float, float, float, float, float);

; The last 4 arguments are the elements of the C fragment
declare void @llvm.nvvm.hmma.m16n16k16.st.c.f16.p<n>i32(i32 addrspace(<n>)* %ptr, i32
↪%ldm, i32 %rowcol, i32, i32, i32, i32);
declare void @llvm.nvvm.hmma.m32n8k16.st.c.f16.p<n>i32(i32 addrspace(<n>)* %ptr, i32
↪%ldm, i32 %rowcol, i32, i32, i32, i32);
declare void @llvm.nvvm.hmma.m8n32k16.st.c.f16.p<n>i32(i32 addrspace(<n>)* %ptr, i32
↪%ldm, i32 %rowcol, i32, i32, i32, i32);
```

These intrinsics store an accumulator fragment to memory at location `%ptr`. The matrix in memory must be in a canonical matrix layout with leading dimension `%ldm`. `%rowcol` specifies which the matrix in memory is row-major (0) or column-major (1). `%rowcol` must be a constant value.

These intrinsics are overloaded based on the address spaces. The address space number <n> must be either 0 (generic), 1 (global) or 3 (shared).

The behavior of this intrinsic is undefined if any thread in the warp has exited.

14.6.5.3 Matrix Multiply-and-Accumulate

The following intrinsics synchronize all threads in a warp and then perform a matrix multiply-and-accumulate operation.

```
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m16n16k16.mma.f16.f16(i32 %rowcol, i32
↪%satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32
↪%b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4, i32 %b5, i32 %b6, i32 %b7, i32 %c0, i32
↪%c1, i32 %c2, i32 %c3);
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m32n8k16.mma.f16.f16(i32 %rowcol, i32
↪%satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32
↪%b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4, i32 %b5, i32 %b6, i32 %b7, i32 %c0, i32
↪%c1, i32 %c2, i32 %c3);
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m8n32k16.mma.f16.f16(i32 %rowcol, i32
↪%satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32
↪%b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4, i32 %b5, i32 %b6, i32 %b7, i32 %c0, i32
↪%c1, i32 %c2, i32 %c3);

declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m16n16k16.mma.f32.f16(i32 %rowcol, i32 %satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3,
↪i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4,
↪i32 %b5, i32 %b6, i32 %b7, i32 %c0, i32 %c1, i32 %c2, i32 %c3);
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m32n8k16.mma.f32.f16(i32 %rowcol, i32 %satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3,
↪i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4,
↪i32 %b5, i32 %b6, i32 %b7, i32 %c0, i32 %c1, i32 %c2, i32 %c3);
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m8n32k16.mma.f32.f16(i32 %rowcol, i32 %satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3,
↪i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4,
↪i32 %b5, i32 %b6, i32 %b7, i32 %c0, i32 %c1, i32 %c2, i32 %c3);

declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m16n16k16.mma.f32.f32(i32 %rowcol, i32 %satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3,
↪i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4,
↪i32 %b5, i32 %b6, i32 %b7, float %c0, float %c1, float %c2, float %c3, float %c4,
↪float %c5, float %c6, float %c7);
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m32n8k16.mma.f32.f32(i32 %rowcol, i32 %satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3,
↪i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4,
↪i32 %b5, i32 %b6, i32 %b7, float %c0, float %c1, float %c2, float %c3, float %c4,
↪float %c5, float %c6, float %c7);
declare {float, float, float, float, float, float, float, float} @llvm.nvvm.hmma.
↪m8n32k16.mma.f32.f32(i32 %rowcol, i32 %satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3,
↪i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4,
↪i32 %b5, i32 %b6, i32 %b7, float %c0, float %c1, float %c2, float %c3, float %c4,
↪float %c5, float %c6, float %c7);

declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m16n16k16.mma.f16.f32(i32 %rowcol, i32
↪%satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32
↪%b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4, i32 %b5, i32 %b6, i32 %b7, float %c0,
↪float %c1, float %c2, float %c3, float %c4, float %c5, float %c6, float %c7);
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m32n8k16.mma.f16.f32(i32 %rowcol, i32
↪%satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32
↪%b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4, i32 %b5, i32 %b6, i32 %b7, float %c0,
↪float %c1, float %c2, float %c3, float %c4, float %c5, float %c6, float %c7);
declare {i32, i32, i32, i32} @llvm.nvvm.hmma.m8n32k16.mma.f16.f32(i32 %rowcol, i32
↪%satf, i32 %a0, i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32
↪%b0, i32 %b1, i32 %b2, i32 %b3, i32 %b4, i32 %b5, i32 %b6, i32 %b7, float %c0,
↪float %c1, float %c2, float %c3, float %c4, float %c5, float %c6, float %c7);
```

(continues on next page)

(continued from previous page)

These intrinsics perform a matrix multiply-and-accumulate operation. `%rowcol` specifies the layout of A and B fragments. It must be a constant value, which can have the following values and semantics.

Encoding	Meaning
0	A fragment is row-major, B fragment is row-major
1	A fragment is row-major, B fragment is column-major
2	A fragment is column-major, B fragment is row-major
3	A fragment is column-major, B fragment is column-major

Support for `%satf` has been removed and this operand must be a constant zero.

The behavior of these intrinsics are undefined if any thread in the warp has exited.

Chapter 15. Source Level Debugging Support

To enable source level debugging of an IR module, NVVM IR supports debug intrinsics and debug information descriptors to express the debugging information. Debug information descriptors are represented using specialized metadata nodes. The current NVVM IR debug metadata version is 3.1.

NVVM IR debugging support is based on that in LLVM 7.0.1 for pre-Blackwell targets and LLVM 20.1.0 for Blackwell and later targets. For the complete semantics of the IR, readers of this chapter should refer to the official LLVM IR Specialized Metadata Nodes and the Source Level Debugging documents. Blackwell and later targets should refer to this and this document respectively.

The following metadata nodes need to be present in the module when debugging support is requested:

- Named metadata node `!llvm.dbg.cu`
- Module flags metadata for "`Debug Info Version`" flag: The *behavior* flag should be `Error`. The value of the flag should be `DEBUG_METADATA_VERSION`, which is 3.
- Named metadata `!nvvmir.version` containing a metadata node with the NVVM IR major and minor version values followed by the NVVM IR debug metadata major and minor version values. The current NVVM IR debug metadata version is 3.1.
- The debug resolution (e.g., full, line info only) is controlled by the DICompileUnit's `emissionKind` field:
  - `FullDebug (value: 1)`: Generate symbolic debug and line information. This requires the libNVVM `-g` option to be specified at compile time.
  - `DebugDirectivesOnly (value: 3)`: Generate line information.

Source level debugging is supported only for a single debug compile unit. If there are multiple input NVVM IR modules, at most one module may have a single debug compile unit.

Chapter 16. NVVM ABI for PTX

16.1. Linkage Types

The following table provides the mapping of NVVM IR linkage types associated with functions and global variables to PTX linker directives .

LLVM Linkage Type		**PTX Linker Directive**
`private`, `internal`		This is the default linkage type and does not require a linker directive.
`external`	Function with definition	`.visible`
	Global variable with initialization	
	Function without definition	`.extern`
	Global variable without initialization	
`common`		`.common` for the global address space, otherwise `.weak`
`available_externally`, `linkonce`, `linkonce_odr`, `weak`, `weak_odr`		`.weak`
All other linkage types		Not supported.

16.2. Parameter Passing and Return

The following table shows the mapping of function argument and return types in NVVM IR to PTX types.

Source Type	Size in Bits	PTX Type
Integer types	<= 32	`.u32` or `.b32` (zero-extended if unsigned) `.s32` or `.b32` (sign-extended if signed)
	64	`.u64` or `.b64` (if unsigned) `.s64` or `.b64` (if signed)
Pointer types (without `byval` attribute)	32	`.u32` or `.b32`
	64	`.u64` or `.b64`
Floating-point types	32	`.f32` or `.b32`
	64	`.f64` or `.b64`
Aggregate types	Any size	`.align`*align* `.b8`*name*[*size*] Where *align* is overall aggregate or vector alignment in bytes, *name* is variable name associated with aggregate or vector, and *size* is the aggregate or vector size in bytes.
Pointer types to aggregate with `byval` attribute	32 or 64	
Vector type	Any size	

Chapter 17. Revision History

Version 1.0

- Initial Release.

Version 1.1

- Added support for UVM managed variables in global property annotation. See Supported Properties.

Version 1.2

- Update to LLVM 3.4 for CUDA 7.0.
- Remove address space intrinsics in favor of `addrspacecast`.
- Add information about source level debugging support.

Version 1.3

- Add support for LLVM 3.8 for CUDA 8.0.

Version 1.4

- Add support for warp-level intrinsics.

Version 1.5

- Add support for LLVM 5.0 for CUDA 9.2.

Version 1.6

- Update to LLVM 7.0.1 for CUDA 11.2.

Version 1.7

- Add support for alloca with dynamic size.

Version 1.8

- Add support for i128 in data layout.

Version 1.9

- Modified text about ignoring shared variable initializations.

Version 1.10

- Added support for grid_constant kernel parameters for CUDA 11.7.

Version 1.11

- Added support for Hopper+ cluster intrinsics and max_blocks_per_cluster kernel property for CUDA 11.8.

- Deprecated support for 32-bit compilation.

Version 2.0

- Updated the NVVM IR to version 2.0 which is incompatible with NVVM IR version 1.x
- Removed address space conversion intrinsics. The IR verifier on 2.0 IR will give an error when these intrinsics are present. Clients of libNVVM are advised to use addrspacecast instruction instead.
- Stricter error checking on the supported datalayouts.
- Older style loop unroll pragma metadata on loop backedges is no longer supported. Clients are advised to use the newer loop pragma metadata defined by the LLVM framework.
- Shared variable initialization with non-undef values is no longer supported. In 1.x versions these initializers were ignored silently. This feature makes the 2.0 version incompatible with 1.x versions.

Chapter 18. Notices

18.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

18.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

18.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright