NVVM IR SPECIFICATION 1.4

SP-06714-001_v1.4 | August 2020

Reference Guide

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

NVVM IR is a compiler IR (internal 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 **`va_arg`** function. Future versions of NVVM may either support or accept and ignore IRs that are illegal in the current version.

This document describes version 1.5 of the NVVM IR and version 2.0 of the NVVM debug metadata (see Source Level Debugging Support).

The current NVVM IR is based on LLVM 5.0. For the complete semantics of the IR, readers of this document should check the official LLVM Language Reference Manual (http://llvm.org/releases/5.0.0/docs/LangRef.html). The NVVM IR version 1.2 and the NVVM debug metadata version 2.0 are based on LLVM 3.4. The Language Reference Manual for LLVM 3.4 is at http://llvm.org/releases/3.4/docs/LangRef.html.

Chapter 2.
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 3. HIGH LEVEL STRUCTURE

3.1. Linkage Types

Supported:

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

Not supported:

- **`linker_private`**
- **`linker_private_weak`**
- **`appending`**
- **`extern_weak`**
- **`dllimport`**
- **`dllexport`**

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

3.2. Calling Conventions

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

3.2.1. Rules and Restrictions

1. **va_arg** is not supported.
2. 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.
3. 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.
4. 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).
5. 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.
6. 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.
7. 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.
8. 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 = metadata !{i32 8, i32 520};
   ```

9. 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**.
10. 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.
11. The absence of the metadata in an indirect call instruction means using natural alignment or the align attribute in the **by-pointer-byval** case.

3.3. Visibility Styles

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

3.4. DLL Storage Classes

DLL storage classes are not supported.

3.5. Thread Local Storage Models

Not supported.

3.6. Comdats

Not supported.

3.7. Named Types

Fully supported.

3.8. source_filename

Accepted and ignored.

3.9. 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 ignored.

3.10. Functions

The following are not supported on functions:

- Alignment
- Explicit section
- Garbage collector name
- Prefix data
- Prologue
- Personality
- Optional list of attached metadata

3.11. Aliases

Supported only as aliases of non-kernel functions.

3.12. Ifuncs

Not supported.

3.13. 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 = metadata !{ 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 = metadata !{ i32 1, i32 0, i32 1, i32 0}
```

The version of NVVM IR described in this document is 1.5. The version of NVVM IR debug metadata described in this document is 2.0.

3.14. Parameter Attributes

Fully supported, except the following:

Accepted and ignored:

- **`inreg`**
- **`nest`**
- **`nonnull`**
- **`dereferenceable(<n>)`**
- **`dereferenceable_or_null(<n>)`**

Not supported:

- **`inalloca`**
- **`swiftself`**
- **`swifterror`**
- **`align <n>`**

See Calling Conventions for the use of the attributes.

3.15. Garbage Collector Names

Not supported.

3.16. Prefix Data

Not supported.

3.17. Attribute Groups

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

3.18. Function Attributes

Supported:

- `alwaysinline`
- `cold`
- `inlinehint`
- `minsize`
- `noduplicate`
- `noinline`
- `noreturn`
- `nounwind`
- `optnone`
- `optsize`
- `readnone`
- `readonly`

Accepted and ignored:

- `allocsize`
- `argmemonly`
- `inaccessiblememonly`
- `inaccessiblemem_or_argmemonly`
- `norecurse`
- `speculatable`
- `writeonly`

Not Supported:

- `alignstack`
- `builtin`
- `nonlazybind`
- `naked`
- `nobuiltin`
- `noimplicitfloat`
- `noredzone`
- `probe-stack`
- `returns_twice`
- `sanitize_address`
- `sanitize_memory`
- `sanitize_thread`
- `ssp`
- `sspreq`
- `sspstrong`
- `stack-probe-size`
- `uwtable`
- `convergent`
- `jumptable`
- `safestack`
- `"thunk"`

3.19. Global Attributes

Not supported.

3.20. Operand Bundles

Not supported.

3.21. Module-Level Inline Assembly

Supported.

3.22. Data Layout

Only the following data layouts are supported,

- 32-bit

  ```
  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
  ```

3.23. Target Triple

Only the following target triples are supported, where * can be any name:

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

3.24. Pointer Aliasing Rules

Fully supported.

3.25. Volatile Memory Access

Fully supported. Note that for code generation: **`ld.volatile`** and **`st.volatile`** will be generated.

3.26. Memory Model for Concurrent Operations

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

3.27. 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.

3.28. Fast Math Flags

Accepted and ignored.

3.29. Use-list Order Directives

Not supported.

Chapter 4. TYPE SYSTEM

Fully supported, except for the following:

- Floating point types **`half`**, **`fp128`**, **`x86_fp80`**, **`fp128`** 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 5. CONSTANTS

Fully supported, except for the following:

- **`Token constants`** is 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 6. OTHER VALUES

6.1. 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.

6.2. Metadata Nodes and Metadata Strings

Fully supported. The **`distinct`** keyword is accepted and ignored. The specialized metadata nodes are not supported.

The following metadata are understood by the NVVM compiler:

- Debug information (using LLVM 3.4 debug metadata)
- **`llvm.loop.unroll.count`**
- **`llvm.loop.unroll.disable`**
- **`llvm.loop.unroll.full`**
- **`pragma unroll`**

  Attached to the branch instruction corresponding to the backedge of a loop.

The kind of the MDNode is **`pragma`**. The first operand is a metadata string **`!"unroll"`** and the second operand is an **`i32`** value which specifies the unroll factor. For example,

```
br i1 %cond, label %BR1, label %BR2, !pragma !42
!42 = metadata !{metadata !"unroll", i32 4}
```

- **`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. 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 8. INSTRUCTIONS

8.1. Terminator Instructions

Supported:

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

Unsupported:

- **`indirectbr`**
- **`invoke`**
- **`resume`**
- **`catchswitch`**
- **`catchret`**
- **`cleanupret`**

8.2. Binary Operations

Supported:

- **`add`**
- **`fadd`**
- **`sub`**
- **`fsub`**
- **`mul`**
- **`fmul`**
- **`udiv`**
- **`sdiv`**
- **`fdiv`**

- **`urem`**
- **`srem`**
- **`frem`**

8.3. Bitwise Binary Operations

Supported:

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

8.4. Vector Operations

Supported:

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

8.5. Aggregate Operations

Supported:

- **`extractvalue`**
- **`insertvalue`**

8.6. Memory Access and Addressing Operations

8.6.1. alloca Instruction

The **`alloca`** instruction returns a generic pointer to the local address space. The number of elements, if specified, must be a compile-time constant, otherwise it is not supported. The **`inalloca`** attribute is not supported.

8.6.2. load Instruction

`load atomic` is not supported.

8.6.3. store Instruction

`store atomic` is not supported.

8.6.4. fence Instruction

Not supported. Use NVVM intrinsic functions instead.

8.6.5. cmpxchg Instruction

Supported for **`i32`** and **`i64`** 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.

8.6.6. atomicrmw Instruction

`nand` is not supported. The other keywords are supported for **`i32`** and **`i64`** 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.

8.6.7. getelementptr Instruction

Fully supported.

8.7. 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`**.

8.8. Other Operations

Supported:

- **`icmp`**
- **`fcmp`**
- **`phi`**
- **`select`**
- **`call`** (The **`musttail`** and **`notail`** markers are not supported. Optional fast math flags are accepted and ignored. See Calling Conventions for other rules and restrictions.)

Unsupported:

- **`va_arg`**
- **`landingpad`**
- **`catchpad`**
- **`cleanuppad`**

Chapter 9. INTRINSIC FUNCTIONS

9.1. Variable Argument Handling Intrinsics

Not supported.

9.2. Accurate Garbage Collection Intrinsics

Not supported.

9.3. Code Generator Intrinsics

Not supported.

9.4. Standard C Library Intrinsics

- **`llvm.memcpy`**

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

- **`llvm.memmove`**

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

- **`llvm.memset`**

  Supported. 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.powi`**

  Not supported.

- **`llvm.sin`**

  Not supported.

- **`llvm.cos`**

  Not supported.

- **`llvm.pow`**

  Not supported.

- **`llvm.exp`**

  Not supported.

- **`llvm.exp2`**

  Not supported.

- **`llvm.log`**

  Not supported.

- **`llvm.log10`**

  Not supported.

- **`llvm.log2`**

  Not supported.

- **`llvm.fma`**

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

- **`llvm.fabs`**

  Not supported.

- **`llvm.copysign`**

  Not supported.

- **`llvm.floor`**

  Not supported.

- **`llvm.ceil`**

  Not supported.

- **`llvm.trunc`**

  Not supported.

- **`llvm.rint`**

Not supported.

- **`llvm.nearbyint`**

  Not supported.

- **`llvm.round`**

  Not supported.

- **`llvm.minnum`**

  Not supported.

- **`llvm.maxnum`**

  Not supported.

9.5. Bit Manipulations Intrinsics

- **`llvm.bitreverse`**

  Not supported.

- **`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.

9.6. Specialised Arithmetic Intrinsics

Supported: **`llvm.fmuladd`**

9.7. Half Precision Floating Point Intrinsics

Supported: **`llvm.convert.to.fp16.f32`**, **`llvm.convert.from.fp16.f32`**, and the LLVM 3.4 versions **`llvm.convert.to.fp16`**, **`llvm.convert.from.fp16`**

9.8. Debugger Intrinsics

Supported: LLVM 3.4 versions of **`llvm.dbg.declare`** and **`llvm.dbg.value`**.

9.9. Exception Handling Intrinsics

Not supported.

9.10. Trampoline Intrinsics

Not supported.

9.11. Masked Vector Load and Store Intrinsics

Not supported.

9.12. Vector Reduction Intrinsics

Not supported.

9.13. Constrained Floating Point Intrinsics

Not supported.

9.14. Constrained libm-equivalent Intrinsics

Not supported.

9.15. Masked Vector Gather and Scatter Intrinsics

Not supported.

9.16. Memory Use Markers

Supported: **`llvm.lifetime.start`**, **`llvm.lifetime.end`**, **`llvm.invariant.start`**, and **`llvm.invariant.end`**.

9.17. General Intrinsics

- **`llvm.var.annotation`**

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

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

  Accepted and ignored.
- **`llvm.debugtrap`**

  Not supported.
- **`llvm.stackguard`**

  Not supported.
- **`llvm.stackprotector`**

  Not supported.
- **`llvm.stackprotectorcheck`**

  Not supported.
- **`llvm.objectsize`**

  Not supported.
- **`llvm.expect`**

  Supported.
- **`llvm.assume`**

  Not supported.
- **`llvm.type.test`**

  Not supported.
- **`llvm.checked.load`**

  Not supported.
- **`llvm.bitset.test`**

  Not supported.
- **`llvm.donothing`**

  Supported.
- **`llvm.experimental.deoptimize`**

Not Supported.

- **`llvm.experimental.guard`**

  Not Supported.

- **`llvm.load.relative`**

  Not Supported.

9.18. Element Wise Atomic Memory Intrinsics

Not supported.

Chapter 10. ADDRESS SPACE

10.1. 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.

10.2. Generic Pointers and Non-Generic Pointers

10.2.1. 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
```

10.2.2. 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 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.

10.2.3. 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.

10.3. The alloca Instruction

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

Chapter 11. GLOBAL PROPERTY ANNOTATION

11.1. 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.

11.2. 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 = metadata !{void (i32, i32)* @_Z6kernelii, metadata !"kernel", i32 1}
  !13 = metadata !{void ()* @_Z7kernel2v, metadata !"kernel", i32 1,
 metadata !"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.

11.3. 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.
`minctasm`	kernel function	Hint/directive to the compiler/ driver, asking it to put at least these many CTAs on an SM.
`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 12. TEXTURE AND SURFACE

12.1. 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

12.2. 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 13. NVVM SPECIFIC INTRINSIC FUNCTIONS

13.1. 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`**.

13.2. 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.membar.cta()
```

is a memory fence at the thread block level.

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

is a memory fence at the device level.

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

is a memory fence at the system level.

13.3. Address space conversion

Attention The NVVM address space conversion intrinsics are deprecated and may be removed from future versions of the NVVM IR specification. Please use the **`addrspacecast`** IR instruction instead.

The following intrinsic functions are provided to support converting pointers from specific address spaces to the generic address space.

```
declare i8* @llvm.nvvm.ptr.global.to.gen.p0i8.p1i8(i8 addrspace(1)*)
declare i8* @llvm.nvvm.ptr.shared.to.gen.p0i8.p3i8(i8 addrspace(3)*)
declare i8* @llvm.nvvm.ptr.constant.to.gen.p0i8.p4i8(i8 addrspace(4)*)
declare i8* @llvm.nvvm.ptr.local.to.gen.p0i8.p5i8(i8 addrspace(5)*)

declare i16* @llvm.nvvm.ptr.global.to.gen.p0i16.p1i16(i16 addrspace(1)*)
declare i16* @llvm.nvvm.ptr.shared.to.gen.p0i16.p3i16(i16 addrspace(3)*)
declare i16* @llvm.nvvm.ptr.constant.to.gen.p0i16.p4i16(i16 addrspace(4)*)
declare i16* @llvm.nvvm.ptr.local.to.gen.p0i16.p5i16(i16 addrspace(5)*)

declare i32* @llvm.nvvm.ptr.global.to.gen.p0i32.p1i32(i32 addrspace(1)*)
declare i32* @llvm.nvvm.ptr.shared.to.gen.p0i32.p3i32(i32 addrspace(3)*)
declare i32* @llvm.nvvm.ptr.constant.to.gen.p0i32.p4i32(i32 addrspace(4)*)
declare i32* @llvm.nvvm.ptr.local.to.gen.p0i32.p5i32(i32 addrspace(5)*)

declare i64* @llvm.nvvm.ptr.global.to.gen.p0i64.p1i64(i64 addrspace(1)*)
declare i64* @llvm.nvvm.ptr.shared.to.gen.p0i64.p3i64(i64 addrspace(3)*)
declare i64* @llvm.nvvm.ptr.constant.to.gen.p0i64.p4i64(i64 addrspace(4)*)
declare i64* @llvm.nvvm.ptr.local.to.gen.p0i64.p5i64(i64 addrspace(5)*)

declare f32* @llvm.nvvm.ptr.global.to.gen.p0f32.p1f32(f32 addrspace(1)*)
declare f32* @llvm.nvvm.ptr.shared.to.gen.p0f32.p3f32(f32 addrspace(3)*)
declare f32* @llvm.nvvm.ptr.constant.to.gen.p0f32.p4f32(f32 addrspace(4)*)
declare f32* @llvm.nvvm.ptr.local.to.gen.p0f32.p5f32(f32 addrspace(5)*)

declare f64* @llvm.nvvm.ptr.global.to.gen.p0f64.p1f64(f64 addrspace(1)*)
declare f64* @llvm.nvvm.ptr.shared.to.gen.p0f64.p3f64(f64 addrspace(3)*)
declare f64* @llvm.nvvm.ptr.constant.to.gen.p0f64.p4f64(f64 addrspace(4)*)
declare f64* @llvm.nvvm.ptr.local.to.gen.p0f64.p5f64(f64 addrspace(5)*)
```

The following intrinsic functions are provided to support converting pointers from the generic address spaces to specific address spaces.

```
declare i8 addrspace(1)* @llvm.nvvm.ptr.gen.to.global.p1i8.p0i8(i8*)
declare i8 addrspace(3)* @llvm.nvvm.ptr.gen.to.shared.p3i8.p0i8(i8*)
declare i8 addrspace(4)* @llvm.nvvm.ptr.gen.to.constant.p4i8.p0i8(i8*)
declare i8 addrspace(5)* @llvm.nvvm.ptr.gen.to.local.p5i8.p0i8(i8*)

declare i16 addrspace(1)* @llvm.nvvm.ptr.gen.to.global.p1i16.p0i16(i16*)
declare i16 addrspace(3)* @llvm.nvvm.ptr.gen.to.shared.p3i16.p0i16(i16*)
declare i16 addrspace(4)* @llvm.nvvm.ptr.gen.to.constant.p4i16.p0i16(i16*)
declare i16 addrspace(5)* @llvm.nvvm.ptr.gen.to.local.p5i16.p0i16(i16*)

declare i32 addrspace(1)* @llvm.nvvm.ptr.gen.to.global.p1i32.p0i32(i32*)
declare i32 addrspace(3)* @llvm.nvvm.ptr.gen.to.shared.p3i32.p0i32(i32*)
declare i32 addrspace(4)* @llvm.nvvm.ptr.gen.to.constant.p4i32.p0i32(i32*)
declare i32 addrspace(5)* @llvm.nvvm.ptr.gen.to.local.p5i32.p0i32(i32*)

declare i64 addrspace(1)* @llvm.nvvm.ptr.gen.to.global.p1i64.p0i64(i64*)
declare i64 addrspace(3)* @llvm.nvvm.ptr.gen.to.shared.p3i64.p0i64(i64*)
declare i64 addrspace(4)* @llvm.nvvm.ptr.gen.to.constant.p4i64.p0i64(i64*)
declare i64 addrspace(5)* @llvm.nvvm.ptr.gen.to.local.p5i64.p0i64(i64*)

declare f32 addrspace(1)* @llvm.nvvm.ptr.gen.to.global.p1f32.p0f32(f32*)
declare f32 addrspace(3)* @llvm.nvvm.ptr.gen.to.shared.p3f32.p0f32(f32*)
declare f32 addrspace(4)* @llvm.nvvm.ptr.gen.to.constant.p4f32.p0f32(f32*)
declare f32 addrspace(5)* @llvm.nvvm.ptr.gen.to.local.p5f32.p0f32(f32*)

declare f64 addrspace(1)* @llvm.nvvm.ptr.gen.to.global.p1f64.p0f64(f64*)
declare f64 addrspace(3)* @llvm.nvvm.ptr.gen.to.shared.p3f64.p0f64(f64*)
declare f64 addrspace(4)* @llvm.nvvm.ptr.gen.to.constant.p4f64.p0f64(f64*)
declare f64 addrspace(5)* @llvm.nvvm.ptr.gen.to.local.p5f64.p0f64(f64*)
```

13.4. 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()
```

13.5. 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 }
```

13.5.1. 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,
                                                      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,
                                                     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)

%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)
```

13.5.2. 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)
```

13.5.3. 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)

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,
                                              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,
                                              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)
```

13.6. Warp-level Operations

13.6.1. 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 any thread participating in the intrinsic has exited or the executing thread is not in the **`%membermask`**.

For **`compute_62`** or below, all threads in **`%membermask`** must call the same **`@llvm.nvvm.bar.warp.sync()`** in convergence, and only threads belonging to the **`%membermask`** can be active when the intrinsic is called. Otherwise, the behavior is undefined.

13.6.2. Data Movement

The following intrinsic synchronizes a subset of threads in a warp and then performs data movement among these threads.

```
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 **`%mode`**. 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.

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

Encoding	Meaning
0	IDX
1	UP
2	DOWN
3	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.

```
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 any thread participating in the intrinsic has exited or the executing thread is not in the **`%membermask`**.

For **`compute_62`** or below, all threads in **`%membermask`** must call the same **`@llvm.nvvm.shfl.sync.i32()`** in convergence, and only threads belonging to the **`%membermask`** can be active when the intrinsic is called. Otherwise, the behavior is undefined.

13.6.3. 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 any thread participating in the intrinsic has exited or the executing thread is not in the **%membermask**.

For **compute_62** or below, all threads in **%membermask** must call the same **@llvm.nvvm.vote.sync()** in convergence, and only threads belonging to the **%membermask** can be active when the intrinsic is called. Otherwise, the behavior is undefined.

13.6.4. 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 any thread participating in the intrinsic has exited or the executing thread is not in the **%membermask**.

These intrinsics are only available on **compute_70** or higher.

13.6.5. 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.

13.6.5.1. 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);
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.

These intrinsics are only available on **`compute_70`** or higher.

13.6.5.2. 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.

These intrinsics are only available on **`compute_70`** or higher.

13.6.5.3. 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);
```

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.

These intrinsics are only available on **`compute_70`** or higher.

Chapter 14.
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 metadata. The current NVVM IR debug metadata version is 2.0.

The current NVVM IR debugging support is based on that in LLVM 3.4. For the complete semantics of the IR, readers of this chapter should check the official Source Level Debugging with LLVM Manual (http://llvm.org/releases/3.4/docs/SourceLevelDebugging.html). Additionally, LLVM 5.0 bitcode format is not supported for source level debugging.

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`** in LLVM 3.4, which is 1.

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.

For the following debug information descriptors, NVVM IR supports a modified form of the descriptors:

- **`DW_TAG_pointer_type`**

```
!5 = metadata !{
  i32,       ;; Tag = DW_TAG_pointer_type
  metadata, ;; Source directory (including trailing slash) and
            ;; file pair (may be null)
  metadata, ;; Reference to context
  metadata, ;; Name (may be "" for anonymous types)
  i32,       ;; Line number where defined (may be 0)
  i64,       ;; Size in bits
  i64,       ;; Alignment in bits
  i64,       ;; Offset in bits
  i32,       ;; Flags to encode attributes, e.g. private
  metadata, ;; Reference to type derived from
  i32        ;; (optional) numeric value of address space for memory
             ;; pointed to
}
```

- **`DW_TAG_subrange_type`**

```
!42 = metadata !{
  i32,       ;; Tag = DW_TAG_subrange_type
  i64,       ;; Low value
  i64,       ;; Count of number of elements
  metadata, ;; (optional) reference to variable holding low value.
            ;; If present, takes precedence over i64 constant
  metadata  ;; (optional) reference to variable holding count.
            ;; If present, takes precedence over i64 constant
}
```

NVVM IR supports the following additional debug information descriptor for **`DW_TAG_module`**:

```
!45 = metadata !{
  i32,       ;; Tag = DW_TAG_module
  metadata, ;; Source directory (including trailing slash) and
            ;; file pair (may be null)
  metadata, ;; Reference to context descriptor
  metadata, ;; Name
  i32        ;; Line number
}
```

Chapter 15. NVVM ABI FOR PTX

15.1. 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.

15.2. 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)

Source Type	Size in Bits	PTX Type
		`.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	

Appendix A. 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