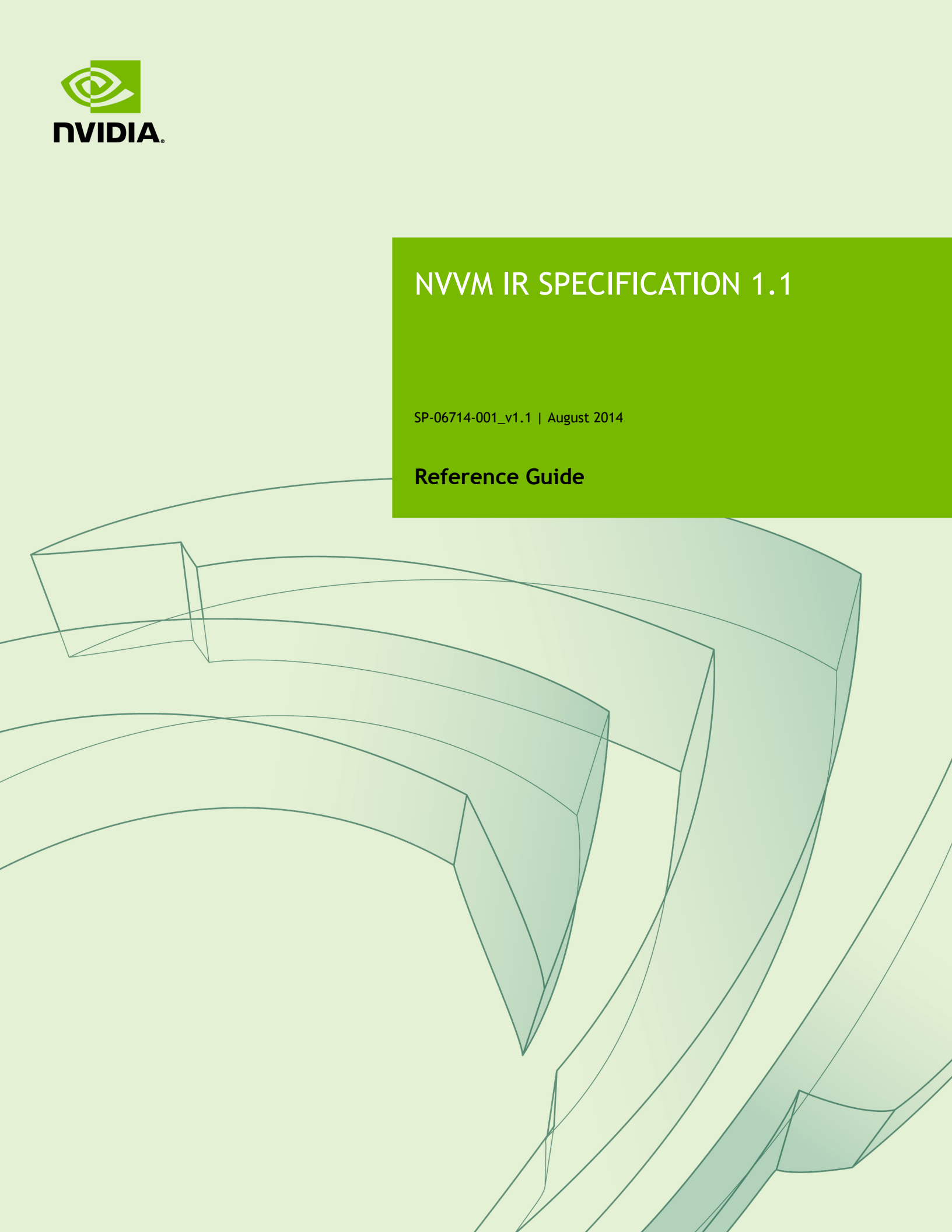

NVVM IR SPECIFICATION 1.1

SP-06714-001_v1.1 | August 2014

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).

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 LLVM 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.

The current NVVM IR is based on LLVM 3.2. For the complete semantics of the IR, readers of this document should check the official LLVM Language Reference Manual (http://llvm.org/releases/3.2/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

All linkage types (except for **dllimport** and **dllexport**) are supported. NVVM ABI for PTX provides details on how they 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 0x0008, i32 0x208};
```

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. Named Types

Fully supported.

3.5. 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.

3.6. Functions

The following are not supported on functions.

- **`explicit section`**
- **`alignment`**

3.7. Aliases

Fully supported.

3.8. Named Metadata

Accepted and ignored, except for the following:

- **`nvvm.annotations`**: see Global Property Annotation
- **`nvvmir.version`**
- debugging information

The NVVM IR version is specified using a named metadata called **`!nvvmir.version`**. The metadata node consists of two i32 values—the first denotes the 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}
```

3.9. Parameter Attributes

Fully supported, except that **`inreg`** is accepted and ignored.

See Calling Conventions for the use of the attributes.

3.10. Garbage Collector Names

Not supported.

3.11. Function Attributes

Supported:

- `alwaysinline`
- `inlinehint`
- `noinline`
- `noreturn`
- `nounwind`
- `optsize`
- `readnone`
- `readonly`

Not Supported:

- `address_safety`
- `alignstack`
- `nonlazybind`
- `naked`
- `noimplicitfloat`
- `noredzone`
- `returns_twice`
- `ssp`
- `sspreq`
- `uwtable`

3.12. Module-Level Inline Assembly

Supported.

3.13. 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
  ```

3.14. Pointer Aliasing Rules

Fully supported.

3.15. Volatile Memory Access

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

3.16. 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.

Chapter 4. TYPE SYSTEM

Fully supported, except for the following:

- Floating point types **`half`**, **`fp128`**, **`x86_fp80`**, and **`ppc_fp128`** are not supported.
- The **`x86mmx`** type is not supported.

Chapter 5. CONSTANTS

Fully supported, except for the following:

- **`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 following metadata are understood by the NVVM compiler:

- Debug information.
- **`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 = !{ !"unroll", i32 4}
```

- **`callalign`**

See Rules and Restrictions for *Calling Conventions*.

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**
- **unwind**
- **resume**

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.

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.
- Requires CUDA architecture **sm_11** or higher.
- Use of a shared pointer requires **sm_12** or higher.
- Use of a generic pointer requires **sm_20** or higher.
- **i64** type with a global pointer requires **sm_12** or higher.
- **i64** type with a shared pointer requires **sm_20** or higher.

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.
- Requires CUDA architecture **sm_11** or higher.
- Use of a shared pointer requires **sm_12** or higher.
- Use of a generic pointer requires **sm_20** or higher.
- **i64** type **xchg**, add and sub with a global pointer require **sm_12** or higher.
- **i64** type **xchg** , add and sub with a shared pointer require **sm_20** or higher.

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**
- **bitcast .. to**

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

8.8. Other Operations

Supported:

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

Unsupported:

- **va_arg**
- **landingpad**

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.log`**

  Not supported.

- **`llvm.fma`**

  Supported for float and vector of float for **`sm_20`** or higher, mapped to PTX **`fma.rn.f32`**.

  Supported for double and vector of double for **`sm_13`** or higher, mapped to PTX **`fma.rn.f64`**.

9.5. Bit Manipulations Intrinsics

- **`llvm.bswap`**

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

- **`llvm.ctpop`**

  Supported for **`i8`**, **`i16`**, **`i32`**, **`i64`**, and vectors of these types, for **`sm_20`** or higher.

- **`llvm.ctlz`**

  Supported for **`i8`**, **`i16`**, **`i32`**, **`i64`**, and vectors of these types, for **`sm_20`** or higher.

- **`llvm.cttz`**

  Supported for **`i8`**, **`i16`**, **`i32`**, **`i64`**, and vectors of these types, for **`sm_20`** or higher.

9.6. Arithmetic with Overflow Intrinsics

Not supported.

9.7. Half Precision Floating Point Intrinsics

Supported: **`llvm.convert.to.fp16`** and **`llvm.convert.from.fp16`**.

9.8. Debugger Intrinsics

Supported: **`llvm.dbg.declare`** and **`llvm.dbg.value`**.

9.9. Exception Handling Intrinsics

Not supported.

9.10. Trampoline Intrinsics

Not supported.

9.11. Memory Use Markers

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

9.12. General Intrinsics

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

  Accepted and ignored.

- **`llvm.annotation`**

  Accepted and ignored.

- **`llvm.trap`**

  Not supported.

- **`llvm.stackprotector`**

  Not supported.

- **`llvm.objectsize`**

  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.

A generic pointer can be converted into a non-generic pointer using one of the following NVVM intrinsic functions.

- **`llvm.nvvm.ptr.gen.to.global`**
- **`llvm.nvvm.ptr.gen.to.shared`**
- **`llvm.nvvm.ptr.gen.to.local`**
- **`llvm.nvvm.ptr.gen.to.constant`**

A non-generic pointer can be converted into a generic pointer using one of the following NVVM intrinsic functions.

- **`llvm.nvvm.ptr.global.to.gen`**
- **`llvm.nvvm.ptr.shared.to.gen`**
- **`llvm.nvvm.ptr.local.to.gen`**
- **`llvm.nvvm.ptr.constant.to.gen`**

The above conversion intrinsic functions are value changing functions, that is, the bit patterns of the output value may be different from the input value.

One can convert a non-generic pointer to a generic pointer and then convert back to the original non-generic space. But it is illegal to convert to a different non-generic space.

`inttoptr` and **`ptrtoint`** are supported. **`inttoptr`** and **`ptrtoin`** are value preserving instructions when the two operands are of the same size.

`bitcast` on pointers is supported. **`bitcast`** is a value preserving instruction. Although it is legal to bitcast a non-generic pointer to a non-generic pointer that points to a different address space, or bitcast between a generic pointer and a non-generic pointer, the producer of the NVVM IR code should make sure they are used correctly. For example, accessing a memory location via a generic pointer bitcasted from a non-generic pointer will result in an undefined result.

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)
```

reads the single 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 three 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

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 support accessing the texture memory and the surface memory.

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

See Accessing Texture Memory or Surface Memory for details.

Chapter 14. NVVM ABI FOR PTX

14.1. 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	
`available_externally`		`.extern`
`linkonce`, `linkonce_odr`, `weak`, `common`		`.weak`
all other linkage types		Not supported.

14.2. Argument for 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)

Source Type	Size in Bits	PTX Type
	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.