PARALLEL THREAD EXECUTION ISA VERSION 3.1

13 September 2012

NVIDIA Compute

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. INTRODUCTION

This document describes PTX, a low-level *parallel thread execution* virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing *device*.

1.1 SCALABLE DATA-PARALLEL COMPUTING USING GPUS

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable GPU has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth. The GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing

are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

PTX defines a virtual machine and ISA for general purpose parallel thread execution. PTX programs are translated at install time to the target hardware instruction set. The PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers.

1.2 GOALS OF PTX

PTX provides a stable programming model and instruction set for general purpose parallel programming. It is designed to be efficient on NVIDIA GPUs supporting the computation features defined by the NVIDIA Tesla architecture. High level language compilers for languages such as CUDA and C/C++ generate PTX instructions, which are optimized for and translated to native target-architecture instructions.

The goals for PTX include the following:

- Provide a stable ISA that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.
- Provide a machine-independent ISA for C/C++ and other compilers to target.
- Provide a code distribution ISA for application and middleware developers.
- Provide a common source-level ISA for optimizing code generators and translators, which map PTX to specific target machines.
- Facilitate hand-coding of libraries, performance kernels, and architecture tests.
- Provide a scalable programming model that spans GPU sizes from a single unit to many parallel units.

1.3 PTX ISA VERSION 3.1

PTX ISA version 3.1 introduces the following new features:

- Support for sm_35 target architecture.
- Support for CUDA Dynamic Parallelism, which enables a CUDA kernel to create and synchronize new work.
- ld.global.nc for loading read-only global data though the non-coherent texture cache.
- A new funnel shift instruction, shf.

- Extends atomic and reduction instructions to perform 64-bit {and, or, xor} operations, and 64-bit integer {min, max} operations.
- Adds support for mipmaps.
- Adds support for indirect access to textures and surfaces.
- Extends support for generic addressing to include the .const state space, and adds a new operator, generic(), to form a generic address for .global or .const variables used in initializers.
- A new .weak directive to permit linking multiple object files containing declarations of the same symbol.

1.4 DOCUMENT STRUCTURE

The information in this document is organized into the following Chapters:

- Chapter 2 outlines the programming model.
- Chapter 3 gives an overview of the PTX virtual machine model.
- Chapter 4 describes the basic syntax of the PTX language.
- Chapter 5 describes state spaces, types, and variable declarations.
- Chapter 6 describes instruction operands.
- Chapter 7 describes the function and call syntax, calling convention, and PTX support for abstracting the Application Binary Interface (ABI).
- Chapter 8 describes the instruction set.
- Chapter 9 lists special registers.
- Chapter 10 lists the assembly directives supported in PTX.
- Chapter 11 provides release notes for PTX ISA versions 2.x and 3.x.

References

- *754-2008 IEEE Standard for Floating-Point Arithmetic*. ISBN 978-0-7381-5752-8, 2008. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933
- *The OpenCL Specification*, Version: 1.1, Document Revision: 44, June 1, 2011. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
- *CUDA Dynamic Parallelism Programming Guide*. 2012

Chapter 2. PROGRAMMING MODEL

2.1 A HIGHLY MULTITHREADED COPROCESSOR

The GPU is a compute device capable of executing a very large number of threads in parallel. It operates as a coprocessor to the main CPU, or host: In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device.

More precisely, a portion of an application that is executed many times, but independently on different data, can be isolated into a *kernel* function that is executed on the GPU as many different threads. To that effect, such a function is compiled to the PTX instruction set and the resulting kernel is translated at install time to the target GPU instruction set.

2.2 THREAD HIERARCHY

The batch of threads that executes a kernel is organized as a grid of cooperative thread arrays as described in this section and illustrated in Figure 1. Cooperative thread arrays (CTAs) implement CUDA thread blocks.

2.2.1 Cooperative Thread Arrays

The Parallel Thread Execution (PTX) programming model is explicitly parallel: a PTX program specifies the execution of a given thread of a parallel thread array. A cooperative *thread array*, or CTA, is an array of threads that execute a kernel concurrently or in parallel.

Threads within a CTA can communicate with each other. To coordinate the communication of the threads within the CTA, one can specify synchronization points where threads wait until all threads in the CTA have arrived.

Each thread has a unique thread identifier within the CTA. Programs use a data parallel decomposition to partition inputs, work, and results across the threads of the CTA. Each CTA thread uses its thread identifier to determine its assigned role, assign specific input and output positions, compute addresses, and select work to perform. The thread identifier is a three-element vector tid, (with elements tid.x, tid.y, and tid.z) that specifies the thread's position within a 1D, 2D, or 3D CTA. Each thread identifier component ranges from zero up to the number of thread ids in that CTA dimension.

Each CTA has a 1D, 2D, or 3D shape specified by a three-element vector ntid (with elements ntid.x, ntid.y, and ntid.z). The vector ntid specifies the number of threads in each CTA dimension.

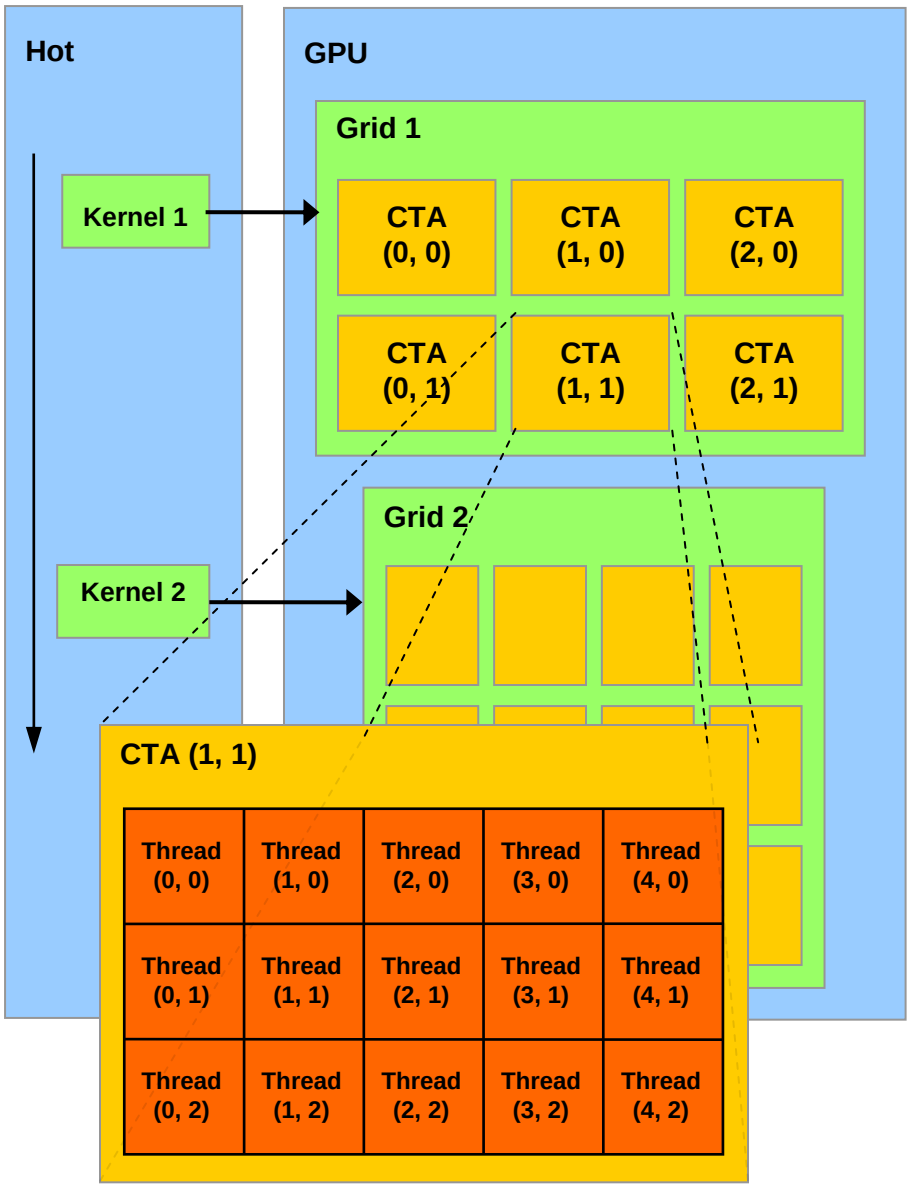
Threads within a CTA execute in SIMT (single-instruction, multiple-thread) fashion in groups called warps. A warp is a maximal subset of threads from a single CTA, such that the threads execute the same instructions at the same time. Threads within a warp are sequentially numbered. The warp size is a machine-dependent constant. Typically, a warp has 32 threads. Some applications may be able to maximize performance with knowledge of the warp size, so PTX includes a run-time immediate constant, WARP_SZ, which may be used in any instruction where an immediate operand is allowed.

2.2.2 Grid of Cooperative Thread Arrays

There is a maximum number of threads that a CTA can contain. However, CTAs that execute the same kernel can be batched together into a grid of CTAs, so that the total number of threads that can be launched in a single kernel invocation is very large. This comes at the expense of reduced thread communication and synchronization, because threads in different CTAs cannot communicate and synchronize with each other.

Multiple CTAs may execute concurrently and in parallel, or sequentially, depending on the platform. Each CTA has a unique CTA identifier (ctaid) within a grid of CTAs. Each grid of CTAs has a 1D, 2D , or 3D shape specified by the parameter nctaid. Each grid also has a unique temporal grid identifier (gridid). Threads may read and use these values through predefined, read-only special registers %tid, %ntid, %ctaid, %nctaid, and %gridid.

The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of CTAs (Figure 1).



A cooperative thread array (CTA) is a set of concurrent threads that execute the same kernel program. A grid is a set of CTAs that execute independently.

Figure 1. Thread Batching

2.3 MEMORY HIERARCHY

PTX threads may access data from multiple memory spaces during their execution as illustrated by Figure 2. Each thread has a private local memory. Each thread block (CTA) has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory.

There are additional memory spaces accessible by all threads: the constant, texture, and surface memory spaces. Constant and texture memory are read-only; surface memory is readable and writable. The global, constant, texture, and surface memory spaces are optimized for different memory usages. For example, texture memory offers different addressing modes as well as data filtering for specific data formats. Note that texture and surface memory is cached, and within the same kernel call, the cache is not kept coherent with respect to global memory writes and surface memory writes, so any texture fetch or surface read to an address that has been written to via a global or a surface write in the same kernel call returns undefined data. In other words, a thread can safely read some texture or surface memory location only if this memory location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread from the same kernel call.

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Both the host and the device maintain their own local memory, referred to as *host memory* and *device memory*, respectively. The device memory may be mapped and read or written by the host, or, for more efficient transfer, copied from the host memory through optimized API calls that utilize the device's high-performance Direct Memory Access (DMA) engine.

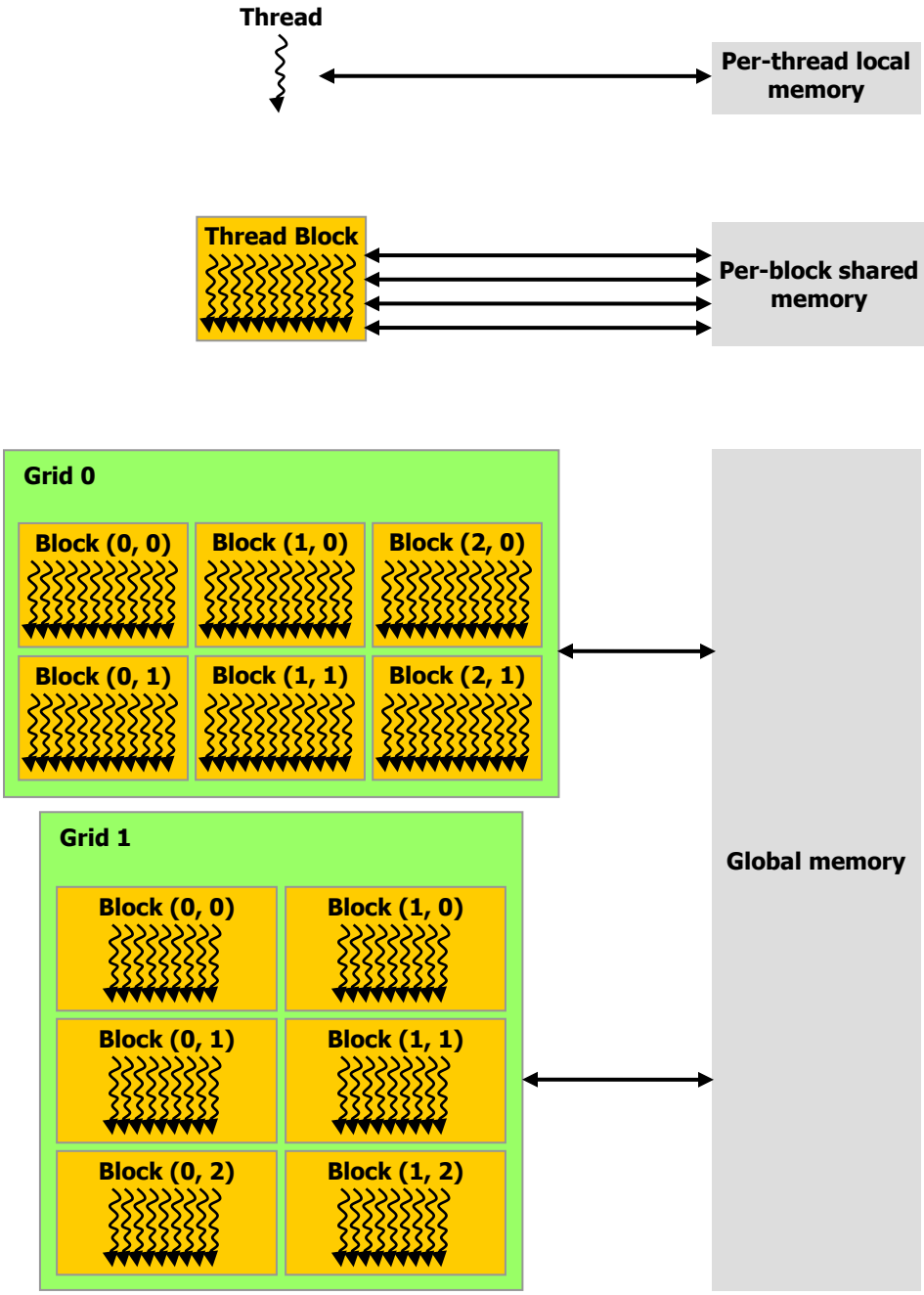


Figure 2. Memory Hierarchy

Chapter 3. PARALLEL THREAD EXECUTION MACHINE MODEL

3.1 A SET OF SIMT MULTIPROCESSORS WITH ON-CHIP SHARED MEMORY

The NVIDIA Tesla architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a host program invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor consists of multiple Scalar Processor (SP) cores, a multithreaded instruction unit, and on-chip shared memory. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. It implements a single-instruction barrier synchronization. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support very fine-grained parallelism, allowing, for example, a low granularity decomposition of problems by assigning one thread to each data element (such as a pixel in an image, a voxel in a volume, a cell in a grid-based computation).

To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture we call SIMT (single-instruction, multiple-thread). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of parallel threads called *warps*. (This term originates from weaving, the first parallel thread technology.) Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.

When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit. The way a block is split into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

At every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths.

SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

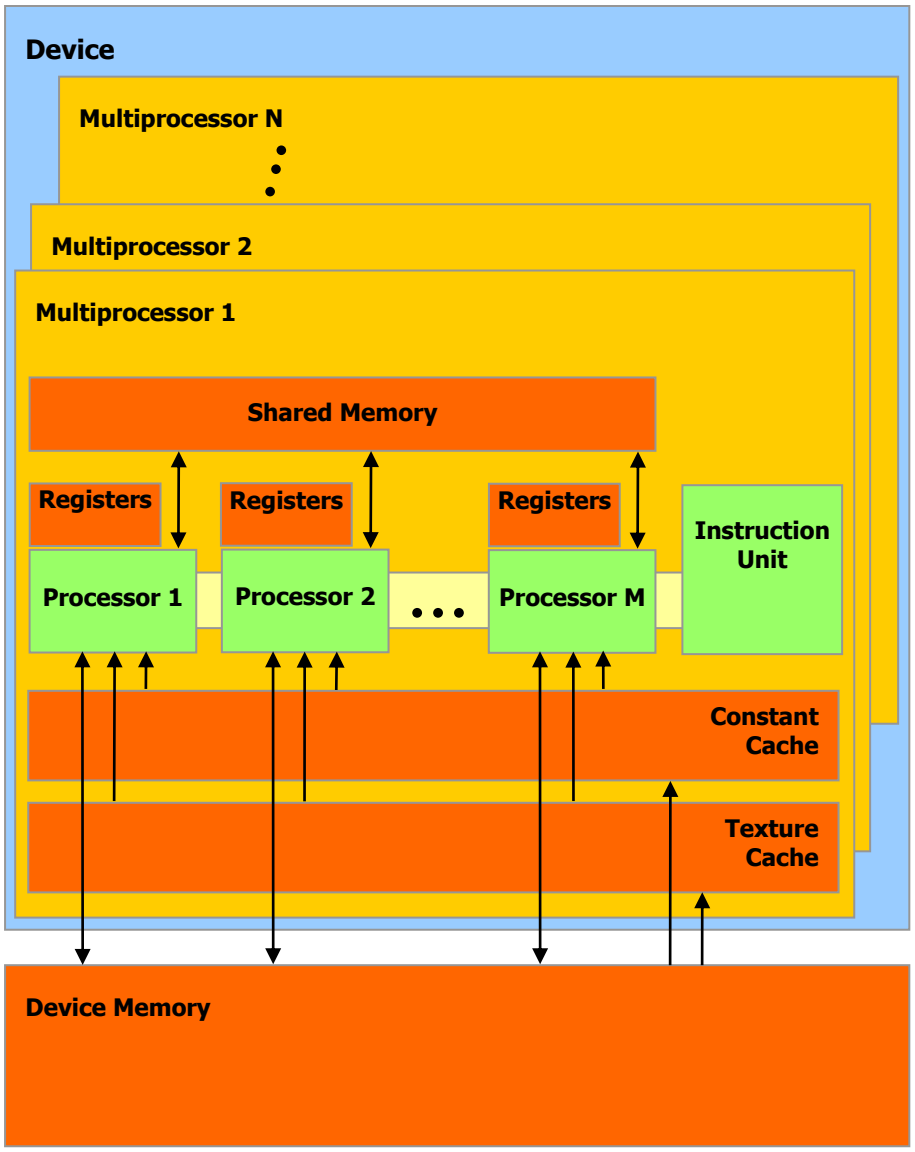
As illustrated by Figure 3, each multiprocessor has on-chip memory of the four following types:

- One set of local 32-bit *registers* per processor,
- A parallel data cache or *shared memory* that is shared by all scalar processor cores and is where the shared memory space resides,
- A read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory,
- A read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory; each multiprocessor accesses the texture cache via a *texture unit* that implements the various addressing modes and data filtering.

The local and global memory spaces are read-write regions of device memory and are not cached.

How many blocks a multiprocessor can process at once depends on how many registers per thread and how much shared memory per block are required for a given kernel since the multiprocessor's registers and shared memory are split among all the threads of the batch of blocks. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch. A multiprocessor can execute as many as eight thread blocks concurrently.

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location and the order in which they occur is undefined, but one of the writes is guaranteed to succeed. If an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read, modify, write to that location occurs and they are all serialized, but the order in which they occur is undefined.



A set of SIMT multiprocessors with on-chip shared memory.

Figure 3. Hardware Model

Chapter 4. SYNTAX

PTX programs are a collection of text source modules (files). PTX source modules have an assembly-language style syntax with instruction operation codes and operands. Pseudo-operations specify symbol and addressing management. The ptxas optimizing backend compiler optimizes and assembles PTX source modules to produce corresponding binary object files.

4.1 SOURCE FORMAT

Source modules are ASCII text. Lines are separated by the newline character (\n).

All whitespace characters are equivalent; whitespace is ignored except for its use in separating tokens in the language.

The C preprocessor cpp may be used to process PTX source modules. Lines beginning with # are preprocessor directives. The following are common preprocessor directives:

#include, #define, #if, #ifdef, #else, #endif, #line, #file

C: A Reference Manual by Harbison and Steele provides a good description of the C preprocessor.

PTX is case sensitive and uses lowercase for keywords.

Each PTX module must begin with a .version directive specifying the PTX language version, followed by a .target directive specifying the target architecture assumed. See Section 9 for a more information on these directives.

4.2 COMMENTS

Comments in PTX follow C/C++ syntax, using non-nested /* and */ for comments that may span multiple lines, and using // to begin a comment that extends up to the next newline character, which terminates the current line. Comments cannot occur within character constants, string literals, or within other comments.

Comments in PTX are treated as whitespace.

4.3 STATEMENTS

A PTX statement is either a directive or an instruction. Statements begin with an optional label and end with a semicolon.

Examples:

```
        .reg     .b32 r1, r2;
        .global  .f32  array[N];

start:  mov.b32   r1, %tid.x;
        shl.b32   r1, r1, 2;          // shift thread id by 2 bits
        ld.global.b32 r2, array[r1];  // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;        // add 1/2
```

4.3.1 Directive Statements

Directive keywords begin with a dot, so no conflict is possible with user-defined identifiers. The directives in PTX are listed in Table 1 and described in Chapter 5 and Chapter 10.

Table 1. PTX Directives

.address_size	.entry	.local	.pragma	.target
.align	.extern	.maxnctapersm	.reg	.tex
.branchtargets	.file	.maxnreg	.reqntid	.version
.callprototype	.func	.maxntid	.section	.visible
.calltargets	.global	.minnctapersm	.shared	.weak
.const	.loc	.param	.sreg	

4.3.2 Instruction Statements

Instructions are formed from an instruction opcode followed by a comma-separated list of zero or more operands, and terminated with a semicolon. Operands may be register variables, constant expressions, address expressions, or label names. Instructions have an optional guard predicate which controls conditional execution. The guard predicate follows the optional label and precedes the opcode, and is written as @p, where p is a predicate register. The guard predicate may be optionally negated, written as @!p.

The destination operand is first, followed by source operands.

Instruction keywords are listed in Table 2. All instruction keywords are reserved tokens in PTX.

Table 2. Reserved Instruction Keywords

abs	div	or	sin	vavrg2, vavrg4
add	ex2	pmevent	slct	vmad
addc	exit	popc	sqrt	vmax
and	fma	prefetch	st	vmax2, vmax4
atom	isspacep	prefetchu	sub	vmin
bar	ld	prmt	subc	vmin2, vmin4
bfe	ldu	rcp	suld	vote
bfi	lg2	red	suq	vset
bfind	mad	rem	sured	vset2, vset4
bra	mad24	ret	sust	vshl
brev	madc	rsqrt	testp	vshr
brkpt	max	sad	tex	vsub
call	membar	selp	tld4	vsub2, vsub4
clz	min	set	trap	xor
cnot	mov	setp	txq	
copysign	mul	shf	vabsdiff	
cos	mul24	shfl	vabsdiff2, vabsdiff4	
cvt	neg	shl	vadd	
cvta	not	shr	vadd2, vadd4	

4.4 IDENTIFIERS

User-defined identifiers follow extended C++ rules: they either start with a letter followed by zero or more letters, digits, underscore, or dollar characters; or they start with an underscore, dollar, or percentage character followed by one or more letters, digits, underscore, or dollar characters:

```
followsym:   [a-zA-Z0-9_$]
identifier:  [a-zA-Z]{followsym}* | {[_$%]{followsym}+
```

PTX does not specify a maximum length for identifiers and suggests that all implementations support a minimum length of at least 1024 characters.

Many high-level languages such as C and C++ follow similar rules for identifier names, except that the percentage sign is not allowed. PTX allows the percentage sign as the first character of an identifier. The percentage sign can be used to avoid name conflicts, e.g. between user-defined variable names and compiler-generated names.

PTX predefines one constant and a small number of special registers that begin with the percentage sign, listed in Table 3.

Table 3. Predefined Identifiers

%clock	%laneid	%lanemask_gt	%pm0, …, %pm3
%clock64	%lanemask_eq	%nctaid	%smid
%ctaid	%lanemask_le	%ntid	%tid
%envreg<32>	%lanemask_lt	%nsmid	%warpid
%gridid	%lanemask_ge	%nwarpid	WARP_SZ

4.5 CONSTANTS

PTX supports integer and floating-point constants and constant expressions. These constants may be used in data initialization and as operands to instructions. Type checking rules remain the same for integer, floating-point, and bit-size types. For predicate-type data and instructions, integer constants are allowed and are interpreted as in C, i.e., zero values are False and non-zero values are True.

4.5.1 Integer Constants

Integer constants are 64-bits in size and are either signed or unsigned, i.e., every integer constant has type .s64 or .u64. The signed/unsigned nature of an integer constant is needed to correctly evaluate constant expressions containing operations such as division and ordered comparisons, where the behavior of the operation depends on the operand types. When used in an instruction or data initialization, each integer constant is converted to the appropriate size based on the data or instruction type at its use.

Integer literals may be written in decimal, hexadecimal, octal, or binary notation. The syntax follows that of C. Integer literals may be followed immediately by the letter 'U' to indicate that the literal is unsigned.

```
hexadecimal literal:  0[xX]{hexdigit}+U?
octal literal:        0{octal digit}+U?
binary literal:       0[bB]{bit}+U?
decimal literal       {nonzero-digit}{digit}*U?
```

Integer literals are non-negative and have a type determined by their magnitude and optional type suffix as follows: literals are signed (.s64) unless the value cannot be fully represented in .s64 or the unsigned suffix is specified, in which case the literal is unsigned (.u64).

The predefined integer constant WARP_SZ specifies the number of threads per warp for the target platform; to date, all target architectures have a WARP_SZ value of 32.

4.5.2 Floating-Point Constants

Floating-point constants are represented as 64-bit double-precision values, and all floating-point constant expressions are evaluated using 64-bit double precision arithmetic. The only exception is the 32-bit hex notation for expressing an exact single-precision floating-point value; such values retain their exact 32-bit single-precision value and may not be used in constant expressions. Each 64-bit floating-point constant is converted to the appropriate floating-point size based on the data or instruction type at its use.

Floating-point literals may be written with an optional decimal point and an optional signed exponent. Unlike C and C++, there is no suffix letter to specify size; literals are always represented in 64-bit double-precision format.

PTX includes a second representation of floating-point constants for specifying the exact machine representation using a hexadecimal constant. To specify IEEE 754 double-precision floating point values, the constant begins with 0d or 0D followed by 16 hex digits. To specify IEEE 754 single-precision floating point values, the constant begins with 0f or 0F followed by 8 hex digits.

```
0[fF]{hexdigit}{8}     // single-precision floating point
0[dD]{hexdigit}{16}    // double-precision floating point
```

Example:

```
      mov.f32  $f3, 0F3f800000;        //  1.0
```

4.5.3 Predicate Constants

In PTX, integer constants may be used as predicates. For predicate-type data initializers and instruction operands, integer constants are interpreted as in C, i.e., zero values are False and non-zero values are True.

4.5.4 Constant Expressions

In PTX, constant expressions are formed using operators as in C and are evaluated using rules similar to those in C, but simplified by restricting types and sizes, removing most casts, and defining full semantics to eliminate cases where expression evaluation in C is implementation dependent.

Constant expressions are formed from constant literals, unary plus and minus, basic arithmetic operators (addition, subtraction, multiplication, division), comparison operators, the conditional ternary operator (? :), and parentheses. Integer constant expressions also allow unary logical negation (!), bitwise complement (~), remainder (%), shift operators (<< and >>), bit-type operators (&, |, and ^), and logical operators (&&, ||).

Constant expressions in PTX do not support casts between integer and floating-point.

Constant expressions are evaluated using the same operator precedence as in C. Table 4 gives operator precedence and associativity. Operator precedence is highest for unary operators and decreases with each line in the chart. Operators on the same line have the same precedence and are evaluated right-to-left for unary operators and left-to-right for binary operators.

Table 4. Operator Precedence

Kind	Operator Symbols	Operator Names	Associates
Primary	()	parenthesis	n/a
Unary	+ - ! ~ (.s64) (.u64)	plus, minus, negation, complement casts	right right
Binary	* / % + - >> << < > <= >= == != & ^ | && ||	multiplication, division, remainder addition, subtraction shifts ordered comparisons equal, not equal bitwise AND bitwise XOR bitwise OR logical AND logical OR	left
Ternary	? :	conditional	right

4.5.5 Integer Constant Expression Evaluation

Integer constant expressions are evaluated at compile time according to a set of rules that determine the type (signed .s64 versus unsigned .u64) of each sub-expression. These rules are based on the rules in C, but they've been simplified to apply only to 64-bit integers, and behavior is fully defined in all cases (specifically, for remainder and shift operators).

- Literals are signed unless unsigned is needed to prevent overflow, or unless the literal uses a 'U' suffix.

Example: 42, 0x1234, 0123 are signed.

Example: 0xfabc123400000000, 42U, 0x1234U are unsigned.

- Unary plus and minus preserve the type of the input operand.

Example: +123, -1, -(-42) are signed

Example: -1U, -0xfabc123400000000 are unsigned.

- Unary logical negation (!) produces a signed result with value 0 or 1.
- Unary bitwise complement (~) interprets the source operand as unsigned and produces an unsigned result.
- Some binary operators require normalization of source operands. This normalization is known as *the usual arithmetic conversions* and simply converts both operands to unsigned type if either operand is unsigned.

- Addition, subtraction, multiplication, and division perform the usual arithmetic conversions and produce a result with the same type as the converted operands. That is, the operands and result are unsigned if either source operand is unsigned, and is otherwise signed.
- Remainder (%) interprets the operands as unsigned. Note that this differs from C, which allows a negative divisor but defines the behavior to be implementation dependent.
- Left and right shift interpret the second operand as unsigned and produce a result with the same type as the first operand. Note that the behavior of right-shift is determined by the type of the first operand: right shift of a signed value is arithmetic and preserves the sign, and right shift of an unsigned value is logical and shifts in a zero bit.
- AND (&), OR (|), and XOR (^) perform the usual arithmetic conversions and produce a result with the same type as the converted operands.
- AND_OP (&&), OR_OP (||), Equal (==), and Not_Equal (!=) produce a signed result. The result value is 0 or 1.
- Ordered comparisons (<, <=, >, >=) perform the usual arithmetic conversions on source operands and produce a signed result. The result value is 0 or 1.
- Casting of expressions to signed or unsigned is supported using (.s64) and (.u64) casts.
- For the conditional operator (? :) , the first operand must be an integer, and the second and third operands are either both integers or both floating-point. The usual arithmetic conversions are performed on the second and third operands, and the result type is the same as the converted type.

4.5.6 Summary of Constant Expression Evaluation Rules

Table 5 contains a summary of the constant expression evaluation rules.

Table 5. Constant Expression Evaluation Rules

Kind	Operator	Operand Types	Operand Interpretation	Result Type
Primary	() constant literal	any type n/a	same as source n/a	same as source .u64, .s64, or .f64
Unary	+ - ! ~	any type integer integer	same as source zero or non-zero .u64	same as source .s64 .u64
Cast	(.u64) (.s64)	integer integer	.u64 .s64	.u64 .s64
Binary	+ - * / < > <= >= == != % >> << & | ^ && ||	.f64 integer .f64 integer .f64 integer integer integer integer integer	.f64 use usual conversions .f64 use usual conversions .f64 use usual conversions .u64 1st unchanged, 2nd is .u64 .u64 zero or non-zero	.f64 converted type .s64 .s64 .s64 .s64 .u64 same as 1st operand .u64 .s64
Ternary	? :	int ? .f64 : .f64 int ? int : int	same as sources use usual conversions	.f64 converted type

Chapter 5. STATE SPACES, TYPES, AND VARIABLES

While the specific resources available in a given target GPU will vary, the kinds of resources will be common across platforms, and these resources are abstracted in PTX through state spaces and data types.

5.1 STATE SPACES

A state space is a storage area with particular characteristics. All variables reside in some state space. The characteristics of a state space include its size, addressability, access speed, access rights, and level of sharing between threads.

The state spaces defined in PTX are a byproduct of parallel programming and graphics programming. The list of state spaces is shown in Table 6, and properties of state spaces are shown in Table 7.

Table 6. State Spaces

Name	Description
.reg	Registers, fast.
.sreg	Special registers. Read-only; pre-defined; platform-specific.
.const	Shared, read-only memory.
.global	Global memory, shared by all threads.
.local	Local memory, private to each thread.
.param	Kernel parameters, defined per-grid; or Function or local parameters, defined per-thread.
.shared	Addressable memory shared between threads in 1 CTA.
.tex	Global texture memory (deprecated).

Table 7. Properties of State Spaces

Name	Addressable	Initializable	Access	Sharing
.reg	No	No	R/W	per-thread
.sreg	No	No	RO	per-CTA
.const	Yes	Yes[1]	RO	per-grid
.global	Yes	Yes[1]	R/W	Context
.local	Yes	No	R/W	per-thread
.param (as input to kernel)	Yes[2]	No	RO	per-grid
.param (used in functions)	Restricted[3]	No	R/W	per-thread
.shared	Yes	No	R/W	per-CTA
.tex	No[4]	Yes, via driver	RO	Context

[1] Variables in .const and .global state spaces are initialized to zero by default.
[2] Accessible only via the ld.param instruction. Address may be taken via mov instruction.
[3] Accessible via ld.param and st.param instructions. Device function input parameters may have their address taken via mov; the parameter is then located on the stack frame and its address is in the .local state space.
[4] Accessible only via the tex instruction.

5.1.1 Register State Space

Registers (.reg state space) are fast storage locations. The number of registers is limited, and will vary from platform to platform. When the limit is exceeded, register variables will be spilled to memory, causing changes in performance. For each architecture, there is a recommended maximum number of registers to use (see the "CUDA Programming Guide" for details).

Registers may be typed (signed integer, unsigned integer, floating point, predicate) or untyped. Register size is restricted; aside from predicate registers which are 1-bit, scalar registers have a width of 8-, 16-, 32-, or 64-bits, and vector registers have a width of 16-, 32-, 64-, or 128-bits. The most common use of 8-bit registers is with ld, st, and cvt instructions, or as elements of vector tuples.

Registers differ from the other state spaces in that they are not fully addressable, i.e., it is not possible to refer to the address of a register. When compiling to use the Application Binary Interface (ABI), register variables are restricted to function scope and may not be declared at module scope. When compiling legacy PTX code (ISA versions prior to 3.0) containing module-scoped .reg variables, the compiler silently disables use of the ABI.

Registers may have alignment boundaries required by multi-word loads and stores.

5.1.2 Special Register State Space

The special register (.sreg) state space holds predefined, platform-specific registers, such as grid, CTA, and thread parameters, clock counters, and performance monitoring registers. All special registers are predefined.

5.1.3 Constant State Space

The constant (.const) state space is a read-only memory initialized by the host. Constant memory is accessed with a ld.const instruction. Constant memory is restricted in size, currently limited to 64KB which can be used to hold statically-sized constant variables. There is an additional 640KB of constant memory, organized as ten independent 64KB regions. The driver may allocate and initialize constant buffers in these regions and pass pointers to the buffers as kernel function parameters. Since the ten regions are not contiguous, the driver must ensure that constant buffers are allocated so that each buffer fits entirely within a 64KB region and does not span a region boundary.

Statically-sized constant variables have an optional variable initializer; constant variables with no explicit initializer are initialized to zero by default. Constant buffers allocated by the driver are initialized by the host, and pointers to such buffers are passed to the kernel as parameters. See the description of kernel parameter attributes in Section 5.1.6.2 for more details on passing pointers to constant buffers as kernel parameters.

5.1.3.1 Banked Constant State Space (deprecated)

Previous versions of PTX exposed constant memory as a set of eleven 64KB banks, with explicit bank numbers required for variable declaration and during access.

Prior to PTX ISA version 2.2, the constant memory was organized into fixed size banks. There were eleven 64KB banks, and banks were specified using the .const[*bank*] modifier, where *bank* ranged from 0 to 10. If no bank number was given, bank zero was assumed.

By convention, bank zero was used for all statically-sized constant variables. The remaining banks were used to declare "incomplete" constant arrays (as in C, for example), where the size is not known at compile time. For example, the declaration

```
.extern .const[2] .b32 const_buffer[];
```

resulted in const_buffer pointing to the start of constant bank two. This pointer could then be used to access the entire 64KB constant bank. Multiple incomplete array variables declared in the same bank were aliased, with each pointing to the start address of the specified constant bank.

To access data in contant banks 1 through 10, the bank number was required in the state space of the load instruction. For example, an incomplete array in bank 2 was accessed as follows:

```
.extern .const[2] .b32 const_buffer[];
ld.const[2].b32  %r1, [const_buffer+4]; // load second word
```

In PTX ISA version 2.2, we eliminated explicit banks and replaced the incomplete array representation of driver-allocated constant buffers with kernel parameter attributes that allow pointers to constant buffers to be passed as kernel parameters.

5.1.4 Global State Space

The global (.global) state space is memory that is accessible by all threads in a context. It is the mechanism by which different CTAs and different grids can communicate. Use ld.global, st.global, and atom.global to access global variables.

Global memory is not sequentially consistent. Consider the case where one thread executes the following two assignments:

```
    a = a + 1;
    b = b - 1;
```

If another thread sees the variable b change, the store operation updating a may still be in flight. This reiterates the kind of parallelism available in machines that run PTX. Threads must be able to do their work without waiting for other threads to do theirs, as in lock-free and wait-free style programming.

Sequential consistency is provided by the bar.sync instruction. Threads wait at the barrier until all threads in the CTA have arrived. All memory writes prior to the bar.sync instruction are guaranteed to be visible to any reads after the barrier instruction.

Global variables have an optional variable initializer; global variables with no explicit initializer are initialized to zero by default.

5.1.5 Local State Space

The local state space (.local) is private memory for each thread to keep its own data. It is typically standard memory with cache. The size is limited, as it must be allocated on a per-thread basis. Use ld.local and st.local to access local variables.

When compiling to use the Application Binary Interface (ABI), .local state-space variables must be declared within function scope and are allocated on the stack. In implementations that do not support a stack, all local memory variables are stored at fixed addresses, recursive function calls are not supported, and .local variables may be declared at module scope. When compiling legacy PTX code (ISA versions prior to 3.0) containing module-scoped .local variables, the compiler silently disables use of the ABI.

5.1.6 Parameter State Space

The parameter (.param) state space is used (1) to pass input arguments from the host to the kernel, (2a) to declare formal input and return parameters for device functions called from within kernel execution, and (2b) to declare locally-scoped byte array variables that serve as function call arguments, typically for passing large structures by value to a function. Kernel function parameters differ from device function parameters in terms of access and sharing (read-only versus read-write, per-kernel versus per-thread). Note that PTX ISA versions 1.x supports only kernel function parameters in .param space; device function parameters were previously restricted to the register state space. The use of parameter state space for device function parameters was introduced in PTX ISA version 2.0 and requires target architecture sm_20 or higher.

Note: The location of parameter space is implementation specific. For example, in some implementations kernel parameters reside in global memory. No access protection is provided between parameter and global space in this case. Similarly, function parameters are mapped to parameter passing registers and/or stack locations based on the function calling conventions of the Application Binary Interface (ABI). Therefore, PTX code should make no assumptions about the relative locations or ordering of .param space variables.

5.1.6.1 Kernel Function Parameters

Each kernel function definition includes an optional list of parameters. These parameters are addressable, read-only variables declared in the .param state space.

Values passed from the host to the kernel are accessed through these parameter variables using ld.param instructions. The kernel parameter variables are shared across all CTAs within a grid.

The address of a kernel parameter may be moved into a register using the mov instruction. The resulting address is in the .param state space and is accessed using ld.param instructions.

Example:

```
.entry foo ( .param .b32 N, .param .align 8 .b8 buffer[64] )
{
    .reg .u32 %n;
    .reg .f64 %d;

    ld.param.u32 %n, [N];
    ld.param.f64 %d, [buffer];
    …
```

Example:

```
.entry bar ( .param .b32 len )
{
    .reg .u32 %ptr, %n;

    mov.u32      %ptr, len;
    ld.param.u32 %n, [%ptr];
    …
```

Kernel function parameters may represent normal data values, or they may hold addresses to objects in constant, global, local, or shared state spaces. In the case of pointers, the compiler and runtime system need information about which parameters are pointers, and to which state space they point. Kernel parameter attribute directives are used to provide this information at the PTX level. See Section 5.1.6.2 for a description of kernel parameter attribute directives.

Note: The current implementation does not allow creation of generic pointers to constant variables (cvta.const) in programs that have pointers to constant buffers passed as kernel parameters.

5.1.6.2 Kernel Function Parameter Attributes

Kernel function parameters may be declared with an optional **.ptr** attribute to indicate that a parameter is a pointer to memory, and also indicate the state space and alignment of the memory being pointed to. Table 8 describes the **.ptr** kernel parameter attribute.

Table 8. Kernel Parameter Attribute: .ptr

.ptr	Kernel parameter alignment attribute
Syntax	`.param .type .ptr .space .align N varname` `.param .type .ptr .align N varname` `.space = { .const, .global, .local, .shared };`
Description	Used to specify the state space and, optionally, the alignment of memory pointed to by a pointer type kernel parameter. The alignment value *N*, if present, must be a power of two. If no state space is specified, the pointer is assumed to be a generic address pointing to one of const, global, local, or shared memory. If no alignment is specified, the memory pointed to is assumed to be aligned to a 4 byte boundary. Spaces between .ptr, *.space*, and .align may be eliminated to improve readability.
PTX ISA Notes	Introduced in PTX ISA version 2.2. Support for generic addressing of .const space added in PTX ISA version 3.1.
Target ISA Notes	Supported on all target architectures.
Examples	`.entry foo ( .param .u32 param1,` `  .param .u32 .ptr.global.align 16 param2,` `  .param .u32 .pt.const.align 8 param3,` `  .param .u32 .ptr.align 16 param4  // generic address pointer` `) { .. }`

5.1.6.3 Device Function Parameters

PTX ISA version 2.0 extended the use of parameter space to device function parameters. The most common use is for passing objects by value that do not fit within a PTX register, such as C structures larger than 8 bytes. In this case, a byte array in parameter space is used. Typically, the caller will declare a locally-scoped .param byte array variable that represents a flattened C structure or union. This will be passed by value to a callee, which declares a .param formal parameter having the same size and alignment as the passed argument.

Example:

```
// pass object of type struct { double d; int y; };
.func foo ( .reg .b32 N, .param .align 8 .b8 buffer[12] )
{
    .reg .f64 %d;
    .reg .s32 %y;

    ld.param.f64 %d, [buffer];
    ld.param.s32 %y, [buffer+8];
    …
}

// code snippet from the caller
// struct { double d; int y; } mystruct; is flattened, passed to foo
    …
    .reg .f64 dbl;
    .reg .s32 x;
    .param .align 8 .b8 mystruct;
    …
    st.param.f64 [mystruct+0], dbl;
    st.param.s32 [mystruct+8], x;
    call foo, (4, mystruct);
    …
```

See the section on function call syntax for more details.

Function input parameters may be read via ld.param and function return parameters may be written using st.param; it is illegal to write to an input parameter or read from a return parameter.

Aside from passing structures by value, .param space is also required whenever a formal parameter has its address taken within the called function. In PTX, the address of a function input parameter may be moved into a register using the mov instruction. Note that the parameter will be copied to the stack if necessary, and so the address will be in the .local state space and is accessed via ld.local and st.local instructions. It is not possible to use mov to get the address of a return parameter or a locally-scoped .param space variable.

Example:

```
// pass array of up to eight floating-point values in buffer
.func foo ( .param .b32 N, .param .b32 buffer[32] )
{
    .reg .u32  %n, %r;
    .reg .f32  %f;
    .reg .pred %p;

    ld.param.u32 %n, [N];
    mov.u32      %r, buffer;  // forces buffer to .local state space
Loop:
    setp.eq.u32  %p, %n, 0;
@p: bra          Done;
    ld.local.f32 %f, [%r];
    …
    add.u32      %r, %r, 4;
    sub.u32      %n, %n, 1;
    bra          Loop;
Done:
    …
}
```

5.1.7 Shared State Space

The shared (.shared) state space is a per-CTA region of memory for threads in a CTA to share data. An address in shared memory can be read and written by any thread in a CTA. Use ld.shared and st.shared to access shared variables.

Shared memory typically has some optimizations to support the sharing. One example is broadcast; where all threads read from the same address. Another is sequential access from sequential threads.

5.1.8 Texture State Space (deprecated)

The texture (.tex) state space is global memory accessed via the texture instruction. It is shared by all threads in a context. Texture memory is read-only and cached, so accesses to texture memory are not coherent with global memory stores to the texture image.

The GPU hardware has a fixed number of texture bindings that can be accessed within a single kernel (typically 128). The .tex directive will bind the named texture memory variable to a hardware texture identifier, where texture identifiers are allocated sequentially beginning with zero. Multiple names may be bound to the same physical texture identifier. An error is generated if the maximum number of physical resources is exceeded. The texture name must be of type .u32 or .u64.

Physical texture resources are allocated on a per-kernel granularity, and .tex variables are required to be defined in the global scope.

Texture memory is read-only. A texture's base address is assumed to be aligned to a 16 byte boundary.

Example:

```
    .tex .u32 tex_a;          // bound to physical texture 0
    .tex .u32 tex_c, tex_d;   // both bound to physical texture 1
    .tex .u32 tex_d;          // bound to physical texture 2
    .tex .u32 tex_f;          // bound to physical texture 3
```

Note: Explicit declarations of variables in the texture state space is deprecated, and programs should instead reference texture memory through variables of type .texref. The .tex directive is retained for backward compatibility, and variables declared in the .tex state space are equivalent to module-scoped .texref variables in the .global state space.

For example, a legacy PTX definitions such as

```
.tex .u32 tex_a;
```

is equivalent to:

```
.global .texref tex_a;
```

See Section 5.3 for the description of the .texref type and Section 8.7.7 for its use in texture instructions.

5.2 TYPES

5.2.1 Fundamental Types

In PTX, the fundamental types reflect the native data types supported by the target architectures. A fundamental type specifies both a basic type and a size. Register variables are always of a fundamental type, and instructions operate on these types. The same type-size specifiers are used for both variable definitions and for typing instructions, so their names are intentionally short.

Table 9 lists the fundamental type specifiers for each basic type:

Table 9. Fundamental Type Specifiers

Basic Type	Fundamental Type Specifiers
Signed integer	.s8, .s16, .s32, .s64
Unsigned integer	.u8, .u16, .u32, .u64
Floating-point	.f16, .f32, .f64
Bits (untyped)	.b8, .b16, .b32, .b64
Predicate	.pred

Most instructions have one or more type specifiers, needed to fully specify instruction behavior. Operand types and sizes are checked against instruction types for compatibility.

Two fundamental types are compatible if they have the same basic type and are the same size. Signed and unsigned integer types are compatible if they have the same size. The bit-size type is compatible with any fundamental type having the same size.

In principle, all variables (aside from predicates) could be declared using only bit-size types, but typed variables enhance program readability and allow for better operand type checking.

5.2.2 Restricted Use of Sub-Word Sizes

The .u8, .s8, and .b8 instruction types are restricted to ld, st, and cvt instructions. The .f16 floating-point type is allowed only in conversions to and from .f32 and .f64 types. All floating-point instructions operate only on .f32 and .f64 types.

For convenience, ld, st, and cvt instructions permit source and destination data operands to be wider than the instruction-type size, so that narrow values may be loaded, stored, and converted using regular-width registers. For example, 8-bit or 16-bit values may be held directly in 32-bit or 64-bit registers when being loaded, stored, or converted to other types and sizes.

5.3 TEXTURE, SAMPLER, AND SURFACE TYPES

PTX includes built-in "opaque" types for defining texture, sampler, and surface descriptor variables. These types have named fields similar to structures, but all information about layout, field ordering, base address, and overall size is hidden to a PTX program, hence the term "opaque". The use of these opaque types is limited to:

- Variable definition within global (module) scope and in kernel entry parameter lists.
- Static initialization of module-scope variables using comma-delimited static assignment expressions for the named members of the type.
- Referencing textures, samplers, or surfaces via texture and surface load/store instructions (tex, suld, sust, sured).
- Retrieving the value of a named member via query instructions (txq, suq).
- Creating pointers to opaque variables using mov, e.g. mov.u64 reg, opaque_var; The resulting pointer may be stored to and loaded from memory, passed as a parameter to functions, and de-referenced by texture and surface load, store, and query instructions, but the pointer cannot otherwise be treated as an address, i.e., accessing the pointer with ld and st instructions, or performing pointer arithmetic will result in undefined results.
- Opaque variables may not appear in initializers, e.g. to initialize a pointer to an opaque variable.

Note: Indirect access to textures and surfaces using pointers to opaque variables is supported beginning with PTX ISA version 3.1 and requires target sm_20 or later.

Indirect access to textures is supported only in unified texture mode (see below).

The three built-in types are .texref, .samplerref, and .surfref. For working with textures and samplers, PTX has two modes of operation. In the *unified mode,* texture and sampler information is accessed through a single .texref handle. In the *independent mode*, texture and sampler information each have their own handle, allowing them to be defined

separately and combined at the site of usage in the program. In *independent mode* the fields of the .texref type that describe sampler properties are ignored, since these properties are defined by .samplerref variables.

Table 10 and Table 11 list the named members of each type for unified and independent texture modes. These members and their values have precise mappings to methods and values defined in the texture HW class as well as exposed values via the API.

Table 10. Opaque Type Fields in Unified Texture Mode

Member	.texref values	.surfref values
width	in elements	
height	in elements	
depth	in elements	
channel_data_type	enum type corresponding to source language API	
channel_order	enum type corresponding to source language API	
normalized_coords	0, 1	N/A
filter_mode	nearest, linear	N/A
addr_mode_0 addr_mode_1 addr_mode_2	wrap, mirror, clamp_ogl, clamp_to_edge, clamp_to_border	N/A

5.3.1 Texture and Surface Properties

Fields width, height, and depth specify the size of the texture or surface in number of elements in each dimension.

The channel_data_type and channel_order fields specify these properties of the texture or surface using enumeration types corresponding to the source language API. For example, see section 5.3.3 for the OpenCL enumeration types currently supported in PTX.

5.3.2 Sampler Properties

The normalized_coords field indicates whether the texture or surface uses normalized coordinates in the range [0.0, 1.0) instead of unnormalized coordinates in the range [0, N). If no value is specified, the default is set by the runtime system based on the source language.

The filter_mode field specifies how the values returned by texture reads are computed based on the input texture coordinates.

The addr_mode_{0,1,2} fields define the addressing mode in each dimension, which determine how out-of-range coordinates are handled.

See the CUDA C Programming Guide for more details of these properties.

Table 11. Opaque Type Fields in Independent Texture Mode

Member	.samplerref values	.texref values	.surfref values
width	N/A	in elements	
height	N/A	in elements	
depth	N/A	in elements	
channel_data_type	N/A	enum type corresponding to source language API	
channel_order	N/A	enum type corresponding to source language API	
normalized_coords	N/A	0, 1	N/A
force_unnormalized_coords	0, 1	N/A	N/A
filter_mode	nearest, linear	ignored	N/A
addr_mode_0 addr_mode_1 addr_mode_2	wrap, mirror, clamp_ogl, clamp_to_edge, clamp_to_border	ignored	N/A

In independent texture mode, the sampler properties are carried in an independent .samplerref variable, and these fields are disabled in the .texref variables. One additional sampler property, force_unnormalized_coords, is available in independent texture mode.

The force_unnormalized_coords field is a property of .samplerref variables that allows the sampler to override the texture header normalized_coords property. This field is defined only in independent texture mode. When True, the texture header setting is overridden and unnormalized coordinates are used; when False, the texture header setting is used.

The force_unnormalized_coords property is used in compiling OpenCL; in OpenCL, the property of normalized coordinates is carried in sampler headers. To compile OpenCL to PTX, texture headers are always initialized with 'normalized_coords' set to True, and the OpenCL sampler-based 'normalized_coords' flag maps (negated) to the PTX-level 'force_unnormalized_coords' flag.

Variables using these types may be declared at module scope or within kernel entry parameter lists. At module scope, these variables must be in the .global state space. As kernel parameters, these variables are declared in the .param state space.

Example:

```
.global .texref     my_texture_name;
.global .samplerref my_sampler_name;
.global .surfref    my_surface_name;
```

When declared at module scope, the types may be initialized using a list of static expressions assigning values to the named members.

Example:

```
.global .texref tex1;
.global .samplerref tsamp1 = { addr_mode_0 = clamp_to_border,
                               filter_mode = nearest
                             };
```

5.3.3 Channel Data Type and Channel Order Fields

The channel_data_type and channel_order fields have enumeration types corresponding to the source language API. Currently, OpenCL is the only source language that defines these fields. Table 12 and Table 13 show the enumeration values defined in OpenCL version 1.0 for channel data type and channel order.

Table 12. OpenCL 1.0 Channel Data Type Definition

CL_SNORM_INT8	0x10D0
CL_SNORM_INT16	0x10D1
CL_UNORM_INT8	0x10D2
CL_UNORM_INT16	0x10D3
CL_UNORM_SHORT_565	0x10D4
CL_UNORM_SHORT_555	0x10D5
CL_UNORM_INT_101010	0x10D6
CL_SIGNED_INT8	0x10D7
CL_SIGNED_INT16	0x10D8
CL_SIGNED_INT32	0x10D9
CL_UNSIGNED_INT8	0x10DA
CL_UNSIGNED_INT16	0x10DB
CL_UNSIGNED_INT32	0x10DC
CL_HALF_FLOAT	0x10DD
CL_FLOAT	0x10DE

Table 13. OpenCL 1.0 Channel Order Definition

CL_R	0x10B0
CL_A	0x10B1
CL_RG	0x10B2
CL_RA	0x10B3
CL_RGB	0x10B4
CL_RGBA	0x10B5
CL_BGRA	0x10B6
CL_ARGB	0x10B7
CL_INTENSITY	0x10B8
CL_LUMINANCE	0x10B9

5.4 VARIABLES

In PTX, a variable declaration describes both the variable's type and its state space. In addition to fundamental types, PTX supports types for simple aggregate objects such as vectors and arrays.

5.4.1 Variable Declarations

All storage for data is specified with variable declarations. Every variable must reside in one of the state spaces enumerated in the previous section.

A variable declaration names the space in which the variable resides, its type and size, its name, an optional array size, an optional initializer, and an optional fixed address for the variable.

Predicate variables may only be declared in the register state space.

Examples:

```
.global .u32 loc;
.reg    .s32 i;
.const  .f32 bias[] = {-1.0, 1.0};
.global .u8  bg[4] = {0, 0, 0, 0};
.reg    .v4 .f32 accel;
.reg    .pred p, q, r;
```

5.4.2 Vectors

Limited-length vector types are supported. Vectors of length 2 and 4 of any non-predicate fundamental type can be declared by prefixing the type with .v2 or .v4. Vectors must be based on a fundamental type, and they may reside in the register space. Vectors cannot exceed 128-bits in length; for example, .v4 .f64 is not allowed. Three-element vectors may be handled by using a .v4 vector, where the fourth element provides padding. This is a common case for three-dimensional grids, textures, etc.

Examples:

```
.global .v4 .f32 V;        // a length-4 vector of floats
.shared .v2 .u16 uv;       // a length-2 vector of unsigned ints
.global .v4 .b8  v;        // a length-4 vector of bytes
```

By default, vector variables are aligned to a multiple of their overall size (vector length times base-type size), to enable vector load and store instructions which require addresses aligned to a multiple of the access size.

5.4.3 Array Declarations

Array declarations are provided to allow the programmer to reserve space. To declare an array, the variable name is followed with dimensional declarations similar to fixed-size array declarations in C. The size of each dimension is a constant expression.

Examples:

```
.local  .u16 kernel[19][19];
.shared .u8  mailbox[128];
```

The size of the array specifies how many elements should be reserved. For the declaration of array ‘kernel’ above, 19*19 = 361 halfwords are reserved, for a total of 722 bytes.

When declared with an initializer, the first dimension of the array may be omitted. The size of the first array dimension is determined by the number of elements in the array initializer.

Examples:

```
.global .u32 index[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
.global .s32 offset[][2] = { {-1, 0}, {0, -1}, {1, 0}, {0, 1} };
```

Array ‘index’ has eight elements, and array ‘offset’ is a 4x2 array.

5.4.4 Initializers

Declared variables may specify an initial value using a syntax similar to C/C++, where the variable name is followed by an equals sign and the initial value or values for the variable. A scalar takes a single value, while vectors and arrays take nested lists of values inside of curly braces (the nesting matches the dimensionality of the declaration).

As in C, array initializers may be incomplete, i.e., the number of initializer elements may be less than the extent of the corresponding array dimension, with remaining array locations initialized to the default value for the specified array type.

Examples:

```
.const  .f32 vals[8] = { 0.33, 0.25, 0.125 };
.global .s32 x[3][2] = { {1,2}, {3} };
```

is equivalent to

```
.const  .f32 vals[4] = { 0.33, 0.25, 0.125, 0.0, 0.0 };
.global .s32 x[3][2] = { {1,2}, {3,0}, {0,0} };
```

Currently, variable initialization is supported only for constant and global state spaces. Variables in constant and global state spaces with no explicit initializer are initialized to zero by default. Initializers are not allowed in external variable declarations.

Variable names appearing in initializers represent the address of the variable; this can be used to statically initialize a pointer to a variable. Initializers may also contain *var+offset* expressions, where *offset* is a byte offset added to the address of *var*. Only variables in .global or .const state spaces may be used in initializers. By default, the resulting address is the offset in the variable's state space (as is the case when taking the address ov a variable with a mov instruction). An operator, generic(), is provided to create a generic address for variables used in initializers.

Examples:

```
.const  .u32 foo = 42;
.global .u32 bar[] = { 2, 3, 5 };
.global .u32 p1 = foo;          // offset of foo in .const space
.global .u32 p2 = generic(foo); // generic address of foo

// array of generic-address pointers to elements of bar
.global .u32 parr[] = { generic(bar), generic(bar)+4, generic(bar)+8 };
```

Note: PTX 3.1 redefines the default addressing for global variables in initializers, from generic addresses to offsets in the global state space. Legacy PTX code is treated as having an implicit generic() operator for each global variable used in an initializer. PTX 3.1 code should either include explicit generic() operators in initializers, use cvta.global to form generic addresses at runtime, or load from the non-generic address using ld.global.

Label names appearing in initializers represent the address of the next instruction following the label; this can be used to initialize a jump table to be used with indirect branches. Device function names appearing in initializers represent the address of the first instruction in the function; this can be used to initialize a table of function pointers to be used with indirect calls. Beginning in PTX ISA version 3.1, kernel function names can be used as initializers e.g. to initialize a table of kernel function pointers, to be used with CUDA Dynamic Parallelism to launch kernels from GPU. See the *CUDA Dynamic Parallelism Programming Guide* for details.

Note: Indirect branch is currently unimplemented.

Variables that hold addresses of variables or instructions should be of type .u32 or .u64.

Initializers are allowed for all types except .f16 and .pred.

Examples:

```
.global .s32 n = 10;
.global .f32 blur_kernel[][3]
               = {{.05,.1,.05},{.1,.4,.1},{.05,.1,.05}};

.global .u32 foo[] = { 2, 3, 5, 7, 9, 11 };
.global .u64 ptr = generic(foo);   // generic address of foo[0]
.global .u64 ptr = generic(foo)+8; // generic address of foo[2]
```

5.4.5 Alignment

Byte alignment of storage for all addressable variables can be specified in the variable declaration. Alignment is specified using an optional .align *byte-count* specifier immediately following the state-space specifier. The variable will be aligned to an address which is an integer multiple of *byte-count*. The alignment value *byte-count* must be a power of two. For arrays, alignment specifies the address alignment for the starting address of the entire array, not for individual elements.

The default alignment for scalar and array variables is to a multiple of the base-type size. The default alignment for vector variables is to a multiple of the overall vector size.

Examples:

```
// allocate array at 4-byte aligned address.  Elements are bytes.
      .const .align 4 .b8 bar[8] = {0,0,0,0,2,0,0,0};
```

Note that all PTX instructions that access memory require that the address be aligned to a multiple of the transfer size.

5.4.6 Parameterized Variable Names

Since PTX supports virtual registers, it is quite common for a compiler frontend to generate a large number of register names. Rather than require explicit declaration of every name, PTX supports a syntax for creating a set of variables having a common prefix string appended with integer suffixes.

For example, suppose a program uses a large number, say one hundred, of .b32 variables, named %r0, %r1, ..., %r99. These 100 register variables can be declared as follows:

```
      .reg .b32 %r<100>;    // declare %r0, %r1, …, %r99
```

This shorthand syntax may be used with any of the fundamental types and with any state space, and may be preceded by an alignment specifier. Array variables cannot be declared this way, nor are initializers permitted.

Chapter 6. INSTRUCTION OPERANDS

6.1 OPERAND TYPE INFORMATION

All operands in instructions have a known type from their declarations. Each operand type must be compatible with the type determined by the instruction template and instruction type. There is no automatic conversion between types.

The bit-size type is compatible with every type having the same size. Integer types of a common size are compatible with each other. Operands having type different from but compatible with the instruction type are silently cast to the instruction type.

6.2 SOURCE OPERANDS

The source operands are denoted in the instruction descriptions by the names a, b, and c. PTX describes a load-store machine, so operands for ALU instructions must all be in variables declared in the .reg register state space. For most operations, the sizes of the operands must be consistent.

The cvt (convert) instruction takes a variety of operand types and sizes, as its job is to convert from nearly any data type to any other data type (and size).

The ld, st, mov, and cvt instructions copy data from one location to another. Instructions ld and st move data from/to addressable state spaces to/from registers. The mov instruction copies data between registers.

Most instructions have an optional predicate guard that controls conditional execution, and a few instructions have additional predicate source operands. Predicate operands are denoted by the names p, q, r, s.

6.3 DESTINATION OPERANDS

PTX instructions that produce a single result store the result in the field denoted by d (for destination) in the instruction descriptions. The result operand is a scalar or vector variable in the register state space.

6.4 USING ADDRESSES, ARRAYS, AND VECTORS

Using scalar variables as operands is straightforward. The interesting capabilities begin with addresses, arrays, and vectors.

6.4.1 Addresses as Operands

Address arithmetic is performed using integer arithmetic and logical instructions. Examples include pointer arithmetic and pointer comparisons. All addresses and address computations are byte-based; there is no support for C-style pointer arithmetic.

The mov instruction can be used to move the address of a variable into a pointer. The address is an offset in the state space in which the variable is declared. Load and store operations move data between registers and locations in addressable state spaces. The syntax is similar to that used in many assembly languages, where scalar variables are simply named and addresses are de-referenced by enclosing the address expression in square brackets. Address expressions include variable names, address registers, address register plus byte offset, and immediate address expressions which evaluate at compile-time to a constant address.

Here are a few examples:

```
    .shared .u16 x;
    .reg    .u16 r0;
    .global .v4 .f32 V;
    .reg    .v4 .f32 W;
    .const  .s32 tbl[256];
    .reg    .b32 p;
    .reg    .s32 q;

    ld.shared.u16   r0,[x];
    ld.gloal.v4.f32 W, [V];
    ld.const.s32    q, [tbl+12];
    mov.u32         p, tbl;
```

6.4.2 Arrays as Operands

Arrays of all types can be declared, and the identifier becomes an address constant in the space where the array is declared. The size of the array is a constant in the program.

Array elements can be accessed using an explicitly calculated byte address, or by indexing into the array using square-bracket notation. The expression within square brackets is either a constant integer, a register variable, or a simple "register with constant offset" expression, where the offset is a constant expression that is either added or subtracted from a register variable. If more complicated indexing is desired, it must be written as an address calculation prior to use. Examples are

```
ld.global.u32  s, a[0];
ld.global.u32  s, a[N-1];
mov.u32        s, a[1];          // move address of a[1] into s
```

6.4.3 Vectors as Operands

Vector operands are supported by a limited subset of instructions, which include mov, ld, st, and tex. Vectors may also be passed as arguments to called functions.

Vector elements can be extracted from the vector with the suffixes .x, .y, .z and .w, as well as the typical color fields .r, .g, .b and .a.

A brace-enclosed list is used for pattern matching to pull apart vectors.

```
.reg .v4 .f32 V;
.reg .f32     a, b, c, d;

mov.v4.f32 {a,b,c,d}, V;
```

Vector loads and stores can be used to implement wide loads and stores, which may improve memory performance. The registers in the load/store operations can be a vector, or a brace-enclosed list of similarly typed scalars. Here are examples:

```
ld.global.v4.f32  {a,b,c,d}, [addr+offset];
ld.global.v2.u32  V2, [addr+offset2];
```

Elements in a brace-enclosed vector, say {Ra, Rb, Rc, Rd}, correspond to extracted elements as follows:

```
Ra = V.x = V.r
Rb = V.y = V.g
Rc = V.z = V.b
Rd = V.w = V.a
```

6.4.4 Labels and Function Names as Operands

Labels and function names can be used only in branch and call instructions, and in move instructions to get the address of the label or function into a register, for use in an indirect branch or call.

Beginning in PTX ISA version 3.1, the mov instruction may be used to take the address of kernel functions, to be passed to a system call that initiates a kernel launch from the GPU. This feature is part of the support for CUDA Dynamic Parallelism. See the *CUDA Dynamic Parallelism Programming Guide* for details.

6.5 TYPE CONVERSION

All operands to all arithmetic, logic, and data movement instruction must be of the same type and size, except for operations where changing the size and/or type is part of the definition of the instruction. Operands of different sizes or types must be converted prior to the operation.

6.5.1 Scalar Conversions

Table 14 shows what precision and format the cvt instruction uses given operands of differing types. For example, if a cvt.s32.u16 instruction is given a u16 source operand and s32 as a destination operand, the u16 is zero-extended to s32.

Conversions to floating-point that are beyond the range of floating-point numbers are represented with the maximum floating-point value (IEEE 754 Inf for f32 and f64, and ~131,000 for f16).

Table 14. Convert Instruction Precision and Format

		Destination Format										
		s8	**s16**	**s32**	**s64**	**u8**	**u16**	**u32**	**u64**	**f16**	**f32**	**f64**
Source Format	**s8**	-	sext	sext	sext	-	sext	sext	sext	s2f	s2f	s2f
	s16	chop[1]	-	sext	sext	chop[1]	-	sext	sext	s2f	s2f	s2f
	s32	chop[1]	chop[1]	-	sext	chop[1]	chop[1]	-	sext	s2f	s2f	s2f
	s64	chop[1]	chop[1]	chop	-	chop[1]	chop[1]	chop	-	s2f	s2f	s2f
	u8	-	zext	zext	zext	-	zext	zext	zext	u2f	u2f	u2f
	u16	chop[1]	-	zext	zext	chop[1]	-	zext	zext	u2f	u2f	u2f
	u32	chop[1]	chop[1]	-	zext	chop[1]	chop[1]	-	zext	u2f	u2f	u2f
	u64	chop[1]	chop[1]	chop	-	chop[1]	chop[1]	chop	-	u2f	u2f	u2f
	f16	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	-	f2f	f2f
	f32	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	f2f	-	f2f
	f64	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	f2f	f2f	-
Notes		sext = sign extend; zext = zero-extend; chop = keep only low bits that fit; s2f = signed-to-float; f2s = float-to-signed; u2f = unsigned-to-float; f2u = float-to-unsigned; f2f = float-to-float; [1] If the destination register is wider than the destination format, the result is extended to the destination register width after chopping. The type of extension (sign or zero) is based on the destination format. For example, cvt.s16.u32 targeting a 32-bit register will first chop to 16-bit then sign-extend to 32-bits.										

6.5.2 Rounding Modifiers

Conversion instructions may specify a rounding modifier. In PTX, there are four integer rounding modifiers and four floating-point rounding modifiers. Table 15 and Table 16 summarize the rounding modifiers.

Table 15. Floating-Point Rounding Modifiers

Modifier	Description
.rn	mantissa LSB rounds to nearest even
.rz	mantissa LSB rounds towards zero
.rm	mantissa LSB rounds towards negative infinity
.rp	mantissa LSB rounds towards positive infinity

Table 16. Integer Rounding Modifiers

Modifier	Description
.rni	round to nearest integer, choosing even integer if source is equidistant between two integers.
.rzi	round to nearest integer in the direction of zero
.rmi	round to nearest integer in direction of negative infinity
.rpi	round to nearest integer in direction of positive infinity

6.6 OPERAND COSTS

Operands from different state spaces affect the speed of an operation. Registers are fastest, while global memory is slowest. Much of the delay to memory can be hidden in a number of ways. The first is to have multiple threads of execution so that the hardware can issue a memory operation and then switch to other execution. Another way to hide latency is to issue the load instructions as early as possible, as execution is not blocked until the desired result is used in a subsequent (in time) instruction. The register in a store operation is available much more quickly. Table 17 gives estimates of the costs of using different kinds of memory.

Table 17. Cost Estimates for Accessing State-Spaces

Space	Time	Notes
Register	0	
Shared	0	
Constant	0	Amortized cost is low, first access is high
Local	> 100 clocks	
Parameter	0	
Immediate	0	
Global	> 100 clocks	
Texture	> 100 clocks	
Surface	> 100 clocks	

Chapter 7. ABSTRACTING THE ABI

Rather than expose details of a particular calling convention, stack layout, and Application Binary Interface (ABI), PTX provides a slightly higher-level abstraction and supports multiple ABI implementations. In this section, we describe the features of PTX needed to achieve this hiding of the ABI. These include syntax for function definitions, function calls, parameter passing, support for variadic functions ("varargs"), and memory allocated on the stack ("alloca").

7.1 FUNCTION DECLARATIONS AND DEFINITIONS

In PTX, functions are declared and defined using the .func directive. A function *declaration* specifies an optional list of return parameters, the function name, and an optional list of input parameters; together these specify the function's interface, or prototype. A function *definition* specifies both the interface and the body of the function. A function must be declared or defined prior to being called.

The simplest function has no parameters or return values, and is represented in PTX as follows:

```
.func foo
{
   …
   ret;
}

   …
   call foo;
   …
```

Here, execution of the call instruction transfers control to foo, implicitly saving the return address. Execution of the ret instruction within foo transfers control to the instruction following the call.

Scalar and vector base-type input and return parameters may be represented simply as register variables. At the call, arguments may be register variables or constants, and return values may be placed directly into register variables. The arguments and return variables at the call must have type and size that match the callee's corresponding formal parameters.

Example:

```
.func (.reg .u32 %res) inc_ptr ( .reg .u32 %ptr, .reg .u32 %inc )
{
   add.u32 %res, %ptr, %inc;
   ret;
}

   …
   call (%r1), inc_ptr, (%r1,4);
   …
```

When using the ABI, .reg state space parameters must be at least 32-bits in size. Subword scalar objects in the source language should be promoted to 32-bit registers in PTX, or use .param state space byte arrays described next.

Objects such as C structures and unions are flattened into registers or byte arrays in PTX and are represented using .param space memory. For example, consider the following C structure, passed by value to a function:

```
struct {
   double dbl;
   char   c[4];
};
```

In PTX, this structure will be flattened into a byte array. Since memory accesses are required to be aligned to a multiple of the access size, the structure in this example will be a 12 byte array with 8 byte alignment so that accesses to the .f64 field are aligned. The .param state space is used to pass the structure by value:

Example:

```
.func (.reg .s32 out) bar (.reg .s32 x, .param .b8 .align 8 y[12])
{
    .reg .f64 f1;
    .reg .b32 c1, c2, c3, c4;
    …
    ld.param.f64 f1, [y+0];
    ld.param.b8  c1, [y+8];
    ld.param.b8  c2, [y+9];
    ld.param.b8  c3, [y+10];
    ld.param.b8  c4, [y+11];
    …
    … // computation using x,f1,c1,c2,c3,c4;
}

{
     .param .b8 .align 8 py[12];
```

```
    …
    st.param.b64 [py+ 0], %rd;
    st.param.b8  [py+ 8], %rc1;
    st.param.b8  [py+ 9], %rc2;
    st.param.b8  [py+10], %rc1;
    st.param.b8  [py+11], %rc2;
    // scalar args in .reg space, byte array in .param space
    call (%out), bar, (%x, py);
    …
```

In this example, note that .param space variables are used in two ways. First, a .param variable y is used in function definition bar to represent a formal parameter. Second, a .param variable py is declared in the body of the calling function and used to set up the structure being passed to bar.

The following is a conceptual way to think about the .param state space use in device functions.

For a caller,

- The .param state space is used to set values that will passed to a called function and/or to receive return values from a called function. Typically, a .param byte array is used to collect together fields of a structure being passed by value.

For a callee,

- The .param state space is used to receive parameter values and/or pass return values back to the caller.

The following restrictions apply to parameter passing.

For a caller,

- Arguments may be .param variables, .reg variables, or constants.
- In the case of .param space formal parameters that are byte arrays, the argument must also be a .param space byte array with matching type, size, and alignment. A .param argument must be declared within the local scope of the caller.
- In the case of .param space formal parameters that are base-type scalar or vector variables, the corresponding argument may be either a .param or .reg space variable with matching type and size, or a constant that can be represented in the type of the formal parameter.
- In the case of .reg space formal parameters, the corresponding argument may be either a .param or .reg space variable of matching type and size, or a constant that can be represented in the type of the formal parameter.
- In the case of .reg space formal parameters, the register must be at least 32-bits in size.
- For .param arguments, all st.param and ld.param instructions used for argument passing must be contained in the basic block with the call instruction. This enables backend optimization and ensures that the .param variable does not consume extra space in the caller's frame beyond that needed by the ABI. The .param variable

simply allows a mapping to be made at the call site between data that may be in multiple locations (e.g., structure being manipulated by caller is located in registers and memory) to something that can be passed as a parameter or return value to the callee.

For a callee,

- Input and return parameters may be .param variables or .reg variables.
- Parameters in .param memory must be aligned to a multiple of 1, 2, 4, 8, or 16 bytes.
- Parameters in the .reg state space must be at least 32-bits in size.
- The .reg state space can be used to receive and return base-type scalar and vector values, including sub-word size objects when compiling in non-ABI mode. Supporting the .reg state space provides legacy support.

Note that the choice of .reg or .param state space for parameter passing has no impact on whether the parameter is ultimately passed in physical registers or on the stack. The mapping of parameters to physical registers and stack locations depends on the ABI definition and the order, size, and alignment of parameters.

7.1.1 Changes from PTX ISA Version 1.x

In PTX ISA version 1.x, formal parameters were restricted to .reg state space, and there was no support for array parameters. Objects such as C structures were flattened and passed or returned using multiple registers. PTX ISA version 1.x supports multiple return values for this purpose.

Beginning with PTX ISA version 2.0, formal parameters may be in either .reg or .param state space, and .param space parameters support arrays. For targets sm_20 or higher, PTX restricts functions to a single return value, and a .param byte array should be used to return objects that do not fit into a register. PTX continues to support multiple return registers for sm_1x targets.

Note: PTX implements a stack-based ABI only for targets sm_20 or higher.

PTX ISA versions prior to 3.0 permitted variables in .reg and .local state spaces to be defined at module scope. When compiling to use the ABI, PTX ISA version 3.0 and later disallows module-scoped .reg and .local variables and restricts their use to within function scope. When compiling without use of the ABI, module-scoped .reg and .local variables are supported as before. When compiling legacy PTX code (ISA versions prior to 3.0) containing module-scoped .reg or .local variables, the compiler silently disables use of the ABI.

7.2 VARIADIC FUNCTIONS

Note: The current version of PTX does not support variadic functions.

To support functions with a variable number of arguments, PTX provides a high-level mechanism similar to the one provided by the stdarg.h and varargs.h headers in C.

In PTX, variadic functions are declared with an ellipsis at the end of the input parameter list, following zero or more fixed parameters:

```
.func baz ( .reg .u32 a, .reg .u32 b, … )
.func okay ( … )
```

Built-in functions are provided to initialize, iteratively access, and end access to a list of variable arguments. The function prototypes are defined as follows:

```
.func (.reg .u32 ptr) %va_start;

.func (.reg .b32 val) %va_arg (.reg .u32 ptr,
                               .reg .u32 sz,
                               .reg .u32 align);

.func (.reg .b64 val) %va_arg64 (.reg .u32 ptr,
                                 .reg .u32 sz,
                                 .reg .u32 align);

.func                 %va_end (.reg .u32 ptr);
```

%va_start returns a handle to whatever structure is used by the ABI to support variable argument lists. This handle is then passed to the %va_arg and %va_arg64 built-in functions, along with the size and alignment of the next data value to be accessed. For %va_arg, the size may be 1, 2, or 4 bytes; for %va_arg64, the size may be 1, 2, 4, or 8 bytes. In both cases, the alignment may be 1, 2, 4, 8, or 16 bytes. Once all arguments have been processed, %va_end is called to free the variable argument list handle.

Here's an example PTX program using the built-in functions to support a variable number of arguments:

```
        // compute max over N signed integers
        .func ( .reg .s32 result ) maxN ( .reg .u32 N, ... )
        {
        .reg .u32  ap, ctr;
        .reg .s32  val;
        .reg .pred p;

        call (ap), %va_start;
        mov.b32 result, 0x8000000;  // default to MININT
        mov.b32 ctr, 0;
Loop:   setp.ge.u32  p, ctr, N;
@p      bra Done;
        call (val), %va_arg, (ap, 4, 4);
        max.s32 result, result, val;
        add.u32 ctr, ctr, 1;
        bra Loop;
Done:
        call %va_end, (ap);
        ret;
        }
       …
       call (%max), maxN, (3, %r1, %r2, %r3);
       …
       call (%max), maxN, (2, %s1, %s2);
       …
```

7.3 ALLOCA

Note: The current version of PTX does not support alloca.

PTX provides another built-in function for allocating storage at runtime on the per-thread local memory stack. To allocate memory, a function simply calls the built-in function %alloca, defined as follows:

```
.func ( .reg .u32 ptr ) %alloca ( .reg .u32 size );
```

The resulting pointer is to the base address in local memory of the allocated memory. The array is then accessed with ld.local and st.local instructions.

If a particular alignment is required, it is the responsibility of the user program to allocate additional space and adjust the base pointer to achieve the desired alignment. The built-in %alloca function is guaranteed only to return a 4-byte aligned pointer.

Chapter 8. INSTRUCTION SET

8.1 FORMAT AND SEMANTICS OF INSTRUCTION DESCRIPTIONS

This section describes each PTX instruction. In addition to the name and the format of the instruction, the semantics are described, followed by some examples that attempt to show several possible instantiations of the instruction.

8.2 PTX INSTRUCTIONS

PTX instructions generally have from zero to four operands, plus an optional guard predicate appearing after an '@' symbol to the left of the opcode:

- @p opcode;
- @p opcode a;
- @p opcode d, a;
- @p opcode d, a, b;
- @p opcode d, a, b, c;

For instructions that create a result value, the d operand is the destination operand, while a, b, and c are source operands.

The setp instruction writes two destination registers. We use a '|' symbol to separate multiple destination registers.

```
setp.lt.s32  p|q, a, b;  // p = (a < b); q = !(a < b);
```

For some instructions the destination operand is optional. A "bit bucket" operand denoted with an underscore (_) may be used in place of a destination register.

8.3 PREDICATED EXECUTION

In PTX, predicate registers are virtual and have .pred as the type specifier. So, predicate registers can be declared as

```
      .reg .pred p, q, r;
```

All instructions have an optional "guard predicate" which controls conditional execution of the instruction. The syntax to specify conditional execution is to prefix an instruction with "@{!}p", where p is a predicate variable, optionally negated. Instructions without a guard predicate are executed unconditionally.

Predicates are most commonly set as the result of a comparison performed by the setp instruction.

As an example, consider the high-level code

```
      if (i < n)
          j = j + 1;
```

This can be written in PTX as

```
      setp.lt.s32  p, i, n;    // p = (i < n)
@p    add.s32      j, j, 1;    // if i < n, add 1 to j
```

To get a conditional branch or conditional function call, use a predicate to control the execution of the branch or call instructions. To implement the above example as a true conditional branch, the following PTX instruction sequence might be used:

```
      setp.lt.s32  p, i, n;    // compare i to n
@!p   bra  L1;                 // if False, branch over
      add.s32      j, j, 1;
L1:   …
```

8.3.1 Comparisons

8.3.1.1 Integer and Bit-Size Comparisons

The signed integer comparisons are the traditional eq (equal), ne (not-equal), lt (less-than), le (less-than-or-equal), gt (greater-than), and ge (greater-than-or-equal). The unsigned comparisons are eq, ne, lo (lower), ls (lower-or-same), hi (higher), and hs (higher-or-same). The bit-size comparisons are eq and ne; ordering comparisons are not defined for bit-size types.

Table 18 shows the operators for signed integer, unsigned integer, and bit-size types.

Table 18. Operators for Signed Integer, Unsigned Integer, and Bit-Size Types

Meaning	Signed Operator	Unsigned Operator	Bit-Size Operator
a == b	eq	eq	eq
a != b	ne	ne	ne
a < b	lt	lo	
a <= b	le	ls	
a > b	gt	hi	
a >= b	ge	hs	

8.3.1.2 Floating-Point Comparisons

The ordered floating-point comparisons are eq, ne, lt, le, gt, and ge. If either operand is NaN, the result is False. Table 19 lists the floating-point comparison operators.

Table 19. Floating-Point Comparison Operators

Meaning	Floating-Point Operator
a == b && !isNaN(a) && !isNaN(b)	eq
a != b && !isNaN(a) && !isNaN(b)	ne
a < b && !isNaN(a) && !isNaN(b)	lt
a <= b && !isNaN(a) && !isNaN(b)	le
a > b && !isNaN(a) && !isNaN(b)	gt
a >= b && !isNaN(a) && !isNaN(b)	ge

To aid comparison operations in the presence of NaN values, unordered floating-point comparisons are provided: equ, neu, ltu, leu, gtu, and geu. If both operands are numeric values (not NaN), then the comparison has the same result as its ordered counterpart. If either operand is NaN, then the result of the comparison is True.

Table 20 lists the floating-point comparison operators accepting NaN values.

Table 20. Floating-Point Comparison Operators Accepting NaN

Meaning	Floating-Point Operator
a == b || isNaN(a) || isNaN(b)	equ
a != b || isNaN(a) || isNaN(b)	neu
a < b || isNaN(a) || isNaN(b)	ltu
a <= b || isNaN(a) || isNaN(b)	leu
a > b || isNaN(a) || isNaN(b)	gtu
a >= b || isNaN(a) || isNaN(b)	geu

To test for NaN values, two operators num (numeric) and nan (isNaN) are provided. num returns True if both operands are numeric values (not NaN), and nan returns True if either operand is NaN. Table 21 lists the floating-point comparison operators testing for NaN values.

Table 21. Floating-Point Comparison Operators Testing for NaN

Meaning	Floating-Point Operator
!isNaN(a) && !isNaN(b)	num
isNaN(a) || isNaN(b)	nan

8.3.2 Manipulating Predicates

Predicate values may be computed and manipulated using the following instructions: and, or, xor, not, and mov.

There is no direct conversion between predicates and integer values, and no direct way to load or store predicate register values. However, setp can be used to generate a predicate from an integer, and the predicate-based select (selp) instruction can be used to generate an integer value based on the value of a predicate; for example:

```
      selp.u32 %r1,1,0,%p;       // convert predicate to 32-bit value
```

8.4 TYPE INFORMATION FOR INSTRUCTIONS AND OPERANDS

Typed instructions must have a type-size modifier. For example, the add instruction requires type and size information to properly perform the addition operation (signed, unsigned, float, different sizes), and this information must be specified as a suffix to the opcode.

Example:

```
.reg .u16 d, a, b;

add.u16 d, a, b;    // perform a 16-bit unsigned add
```

Some instructions require multiple type-size modifiers, most notably the data conversion instruction cvt. It requires separate type-size modifiers for the result and source, and these are placed in the same order as the operands. For example:

```
.reg .u16 a;
.reg .f32 d;

cvt.f32.u16 d, a;   // convert 16-bit unsigned to 32-bit float
```

In general, an operand's type must agree with the corresponding instruction-type modifier. The rules for operand and instruction type conformance are as follows:

- Bit-size types agree with any type of the same size.
- Signed and unsigned integer types agree provided they have the same size, and integer operands are silently cast to the instruction type if needed. For example, an unsigned integer operand used in a signed integer instruction will be treated as a signed integer by the instruction.
- Floating-point types agree only if they have the same size; i.e., they must match exactly.

Table 22 summarizes these type checking rules.

Table 22. Type Checking Rules

		Operand Type			
		.bX	.sX	.uX	.fX
Instruction Type	.bX	okay	okay	okay	okay
	.sX	okay	okay	okay	invalid
	.uX	okay	okay	okay	invalid
	.fX	okay	invalid	invalid	okay

Note: some operands have their type and size defined independently from the instruction type-size. For example, the shift amount operand for left and right shift instructions always has type .u32, while the remaining operands have their type and size determined by the instruction type.

Example:

```
// 64-bit arithmetic right shift; shift amount 'b' is .u32
    shr.s64 d,a,b;
```

8.4.1 Operand Size Exceeding Instruction-Type Size

For convenience, ld, st, and cvt instructions permit source and destination data operands to be wider than the instruction-type size, so that narrow values may be loaded, stored, and converted using regular-width registers. For example, 8-bit or 16-bit values may be held directly in 32-bit or 64-bit registers when being loaded, stored, or converted to other types and sizes. The operand type checking rules are relaxed for bit-size and integer (signed and unsigned) instruction types; floating-point instruction types still require that the operand type-size matches exactly, unless the operand is of bit-size type.

When a source operand has a size that exceeds the instruction-type size, the source data is truncated ("chopped") to the appropriate number of bits specified by the instruction type-size.

Table 23 summarizes the relaxed type-checking rules for source operands. Note that some combinations may still be invalid for a particular instruction; for example, the cvt instruction does not support .bX instruction types, so those rows are invalid for cvt.

Table 23. Relaxed Type-checking Rules for Source Operands

		Source Operand Type													
		b8	**b16**	**b32**	**b64**	**s8**	**s16**	**s32**	**s64**	**u8**	**u16**	**u32**	**u64**	**f16**	**f32**	**f64**
Instruction Type	**b8**	-	chop	chop	chop	-	chop	chop	chop	-	chop	chop	chop	chop	chop	chop
	b16	inv	-	chop	chop	inv	-	chop	chop	inv	-	chop	chop	-	chop	chop
	b32	inv	inv	-	chop	inv	inv	-	chop	inv	inv	-	chop	inv	-	chop
	b64	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv	-	inv	inv	-
	s8	-	chop	chop	chop	-	chop	chop	chop	-	chop	chop	chop	inv	inv	inv
	s16	inv	-	chop	chop	inv	-	chop	chop	inv	-	chop	chop	inv	inv	inv
	s32	inv	inv	-	chop	inv	inv	-	chop	inv	inv	-	chop	inv	inv	inv
	s64	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv
	u8	-	chop	chop	chop	-	chop	chop	chop	-	chop	chop	chop	inv	inv	inv
	u16	inv	-	chop	chop	inv	-	chop	chop	inv	-	chop	chop	inv	inv	inv
	u32	inv	inv	-	chop	inv	inv	-	chop	inv	inv	-	chop	inv	inv	inv
	u64	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv
	f16	inv	-	chop	chop	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv
	f32	inv	inv	-	chop	inv	inv	inv	inv	inv	inv	inv	inv	inv	-	inv
	f64	inv	inv	inv	-	inv	inv	inv	inv	inv	inv	inv	inv	inv	inv	-
Notes		chop = keep only low bits that fit; "-" = allowed, no conversion needed; inv = invalid, parse error. 1. Source register size must be of equal or greater size than the instruction-type size. 2. Bit-size source registers may be used with any appropriately-sized instruction type. The data is truncated ("chopped") to the instruction-type size and interpreted according to the instruction type. 3. Integer source registers may be used with any appropriately-sized bit-size or integer instruction type. The data is truncated to the instruction-type size and interpreted according to the instruction type. 4. Floating-point source registers can only be used with bit-size or floating-point instruction types. When used with a narrower bit-size type, the data will be truncated. When used with a floating-point instruction type, the size must match exactly.														

When a destination operand has a size that exceeds the instruction-type size, the destination data is zero- or sign-extended to the size of the destination register. If the corresponding instruction type is signed integer, the data is sign-extended; otherwise, the data is zero-extended.

Table 24 summarizes the relaxed type-checking rules for destination operands.

Table 24. Relaxed Type-checking Rules for Destination Operands

		Destination Operand Type														
		b8	**b16**	**b32**	**b64**	**s8**	**s16**	**s32**	**s64**	**u8**	**u16**	**u32**	**u64**	**f16**	**f32**	**f64**
Instruction Type	**b8**	-	zext	zext	zext	-	zext	zext	zext	-	zext	zext	zext	zext	zext	zext
	b16	inv	-	zext	zext	inv	-	zext	zext	inv	-	zext	zext	-	zext	zext
	b32	inv	inv	-	zext	inv	inv	-	zext	inv	inv	-	zext	inv	-	zext
	b64	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv	-	inv	inv	-
	s8	-	sext	sext	sext	-	sext	sext	sext	-	sext	sext	sext	inv	inv	inv
	s16	inv	-	sext	sext	inv	-	sext	sext	inv	-	sext	sext	inv	inv	inv
	s32	inv	inv	-	sext	inv	inv	-	sext	inv	inv	-	sext	inv	inv	inv
	s64	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv
	u8	-	zext	zext	zext	-	zext	zext	zext	-	zext	zext	zext	inv	inv	inv
	u16	inv	-	zext	zext	inv	-	zext	zext	inv	-	zext	zext	inv	inv	inv
	u32	inv	inv	-	zext	inv	inv	-	zext	inv	inv	-	zext	inv	inv	inv
	u64	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv	-	inv	inv	inv
	f16	inv	-	zext	zext	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv
	f32	inv	inv	-	zext	inv	inv	inv	inv	inv	inv	inv	inv	inv	-	inv
	f64	inv	inv	inv	-	inv	inv	inv	inv	inv	inv	inv	inv	inv	inv	-
Notes		sext = sign extend; zext = zero-extend; "-" = Allowed but no conversion needed; inv = Invalid, parse error. 1. Destination register size must be of equal or greater size than the instruction-type size. 2. Bit-size destination registers may be used with any appropriately-sized instruction type. The data is sign-extended to the destination register width for signed integer instruction types, and is zero-extended to the destination register width otherwise. 3. Integer destination registers may be used with any appropriately-sized bit-size or integer instruction type. The data is sign-extended to the destination register width for signed integer instruction types, and is zero-extended to the destination register width for bit-size and unsigned integer instruction types. 4. Floating-point destination registers can only be used with bit-size or floating-point instruction types. When used with a narrower bit-size instruction type, the data will be zero-extended. When used with a floating-point instruction type, the size must match exactly.														

8.5 DIVERGENCE OF THREADS IN CONTROL CONSTRUCTS

Threads in a CTA execute together, at least in appearance, until they come to a conditional control construct such as a conditional branch, conditional function call, or conditional return. If threads execute down different control flow paths, the threads are called *divergent*. If all of the threads act in unison and follow a single control flow path, the threads are called *uniform*. Both situations occur often in programs.

A CTA with divergent threads may have lower performance than a CTA with uniformly executing threads, so it is important to have divergent threads re-converge as soon as possible. All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the .uni suffix. For divergent control flow, the optimizing code generator automatically determines points of re-convergence. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent.

8.6 SEMANTICS

The goal of the semantic description of an instruction is to describe the results in all cases in as simple language as possible. The semantics are described using C, until C is not expressive enough.

8.6.1 Machine-Specific Semantics of 16-bit Code

A PTX program may execute on a GPU with either a 16-bit or a 32-bit data path. When executing on a 32-bit data path, 16-bit registers in PTX are mapped to 32-bit physical registers, and 16-bit computations are "promoted" to 32-bit computations. This can lead to computational differences between code run on a 16-bit machine versus the same code run on a 32-bit machine, since the "promoted" computation may have bits in the high-order half-word of registers that are not present in 16-bit physical registers. These extra precision bits can become visible at the application level, for example, by a right-shift instruction.

At the PTX language level, one solution would be to define semantics for 16-bit code that is consistent with execution on a 16-bit data path. This approach introduces a performance penalty for 16-bit code executing on a 32-bit data path, since the translated code would require many additional masking instructions to suppress extra precision bits in the high-order half-word of 32-bit registers.

Rather than introduce a performance penalty for 16-bit code running on 32-bit GPUs, the semantics of 16-bit instructions in PTX is machine-specific. A compiler or programmer may chose to enforce portable, machine-independent 16-bit semantics by adding explicit conversions to 16-bit values at appropriate points in the program to guarantee portability of the code. However, for many performance-critical applications, this is not desirable, and for many applications the difference in execution is preferable to limiting performance.

8.7 INSTRUCTIONS

All PTX instructions may be predicated. In the following descriptions, the optional guard predicate is omitted from the syntax.

8.7.1 Integer Arithmetic Instructions

Integer arithmetic instructions operate on the integer types in register and constant immediate forms. The integer arithmetic instructions are:

- add
- sub
- mul
- mad
- mul24
- mad24
- sad
- div
- rem
- abs
- neg
- min
- max
- popc
- clz
- bfind
- brev
- bfe
- bfi

Table 25. Integer Arithmetic Instructions: add

add	Add two values.
Syntax	`add.`*type* `d, a, b;` `add{.sat}.s32 d, a, b; // .sat applies only to .s32` `.`*type* `= { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Performs addition and writes the resulting value into a destination register.
Semantics	d = a + b;
Notes	Saturation modifier: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`@p add.u32 x,y,z;` `add.sat.s32 c,c,1;`

Table 26. Integer Arithmetic Instructions: sub

sub	Subtract one value from another.
Syntax	`sub.`*type* `d, a, b;` `sub{.sat}.s32 d, a, b; // .sat applies only to .s32` `.`*type* `= { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Performs subtraction and writes the resulting value into a destination register.
Semantics	d = a – b;
Notes	Saturation modifier: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`sub.s32 c,a,b;`

Table 27. Integer Arithmetic Instructions: mul

mul	Multiply two values.
Syntax	`mul{.hi,.lo,.wide}.`*`type`*`  d, a, b;` `.`*`type`*` = { .u16, .u32, .u64,` `  .s16, .s32, .s64 };`
Description	Compute the product of two values.
Semantics	t = a * b; n = bitwidth of type; d = t; // for .wide d = t<2n-1..n>; // for .hi variant d = t<n-1..0>; // for .lo variant
Notes	The type of the operation represents the types of the **a** and **b** operands. If **.hi** or **.lo** is specified, then **d** is the same size as **a** and **b**, and either the upper or lower half of the result is written to the destination register. If .wide is specified, then **d** is twice as wide as **a** and **b** to receive the full result of the multiplication. The .wide suffix is supported only for 16- and 32-bit integer types.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`mul.wide.s16 fa,fxs,fys;  // 16*16 bits yields 32 bits` `mul.lo.s16 fa,fxs,fys;  // 16*16 bits, save only the low 16 bits` `mul.wide.s32 z,x,y;  // 32*32 bits, creates 64 bit result`

Table 28. Integer Arithmetic Instructions: mad

mad	Multiply two values, optionally extract the high or low half of the intermediate result, and add a third value.
Syntax	`mad{.hi,.lo,.wide}.type d, a, b, c;` `mad.hi.sat.s32 d, a, b, c;` `.type = { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Multiplies two values, optionally extracts the high or low half of the intermediate result, and adds a third value. Writes the result into a destination register.
Semantics	t = a * b; n = bitwidth of type; d = t + c; // for .wide d = t<2n-1..n> + c; // for .hi variant d = t<n-1..0> + c; // for .lo variant
Notes	The type of the operation represents the types of the **a** and **b** operands. If .hi or .lo is specified, then **d** and **c** are the same size as **a** and **b**, and either the upper or lower half of the result is written to the destination register. If .wide is specified, then **d** and **c** are twice as wide as **a** and **b** to receive the result of the multiplication. The .wide suffix is supported only for 16- and 32-bit integer types. Saturation modifier: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type in .hi mode.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`@p mad.lo.s32 d,a,b,c;` `mad.lo.s32 r,p,q,r;`

Table 29. Integer Arithmetic Instructions: mul24

mul24	Multiply two 24-bit integer values.
Syntax	`mul24{.hi,.lo}.`*`type`*`  d, a, b;` `.`*`type`*` = { .u32, .s32 };`
Description	Compute the product of two 24-bit integer values held in 32-bit source registers, and return either the high or low 32-bits of the 48-bit result.
Semantics	t = a * b; d = t<47..16>; // for .hi variant d = t<31..0>; // for .lo variant
Notes	Integer multiplication yields a result that is twice the size of the input operands, i.e. 48-bits. mul24.hi performs a 24x24-bit multiply and returns the high 32 bits of the 48-bit result. mul24.lo performs a 24x24-bit multiply and returns the low 32 bits of the 48-bit result. All operands are of the same type and size. mul24.hi may be less efficient on machines without hardware support for 24-bit multiply.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`mul24.lo.s32 d,a,b;  // low 32-bits of 24x24-bit signed multiply.`

Table 30. Integer Arithmetic Instructions: mad24

mad24	Multiply two 24-bit integer values and add a third value.
Syntax	`mad24{.hi,.lo}.`*`type`*`  d, a, b, c;` `mad24.hi.sat.s32  d, a, b, c;` *`.type`* `= { .u32, .s32 };`
Description	Compute the product of two 24-bit integer values held in 32-bit source registers, and add a third, 32-bit value to either the high or low 32-bits of the 48-bit result. Return either the high or low 32-bits of the 48-bit result.
Semantics	t = a * b; d = t<47..16> + c; // for .hi variant d = t<31..0> + c; // for .lo variant
Notes	Integer multiplication yields a result that is twice the size of the input operands, i.e. 48-bits. mad24.hi performs a 24x24-bit multiply and adds the high 32 bits of the 48-bit result to a third value. mad24.lo performs a 24x24-bit multiply and adds the low 32 bits of the 48-bit result to a third value. All operands are of the same type and size. Saturation modifier: .sat limits result of 32-bit signed addition to MININT..MAXINT (no overflow). Applies only to .s32 type in .hi mode. mad24.hi may be less efficient on machines without hardware support for 24-bit multiply.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`mad24.lo.s32 d,a,b,c;  // low 32-bits of 24x24-bit signed multiply.`

Table 31. Integer Arithmetic Instructions: sad

sad	Sum of absolute differences.
Syntax	`sad.type d, a, b, c;` `.type = { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Adds the absolute value of **a**-**b** to **c** and writes the resulting value into **d**.
Semantics	d = c + ((a<b) ? b-a : a-b);
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`sad.s32 d,a,b,c;` `sad.u32 d,a,b,d; // running sum`

Table 32. Integer Arithmetic Instructions: div

div	Divide one value by another.
Syntax	div.*type* d, a, b; *.type* = { .u16, .u32, .u64, .s16, .s32, .s64 };
Description	Divides **a** by **b**, stores result in **d**.
Semantics	d = a / b;
Notes	Division by zero yields an unspecified, machine-specific value.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	div.s32 b,n,i;

Table 33. Integer Arithmetic Instructions: rem

rem	The remainder of integer division.
Syntax	`rem.type d, a, b;` `.type = { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Divides **a** by **b**, store the remainder in **d**.
Semantics	d = a % b;
Notes	The behavior for negative numbers is machine-dependent and depends on whether divide rounds towards zero or negative infinity.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`rem.s32 x,x,8; // x = x%8;`

Table 34. Integer Arithmetic Instructions: abs

abs	Absolute value.
Syntax	`abs.`*type* `d, a;` *type* `= { .s16, .s32, .s64 };`
Description	Take the absolute value of **a** and store it in **d**.
Semantics	`d = |a|;`
Notes	Only for signed integers.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`abs.s32 r0,a;`

Table 35. Integer Arithmetic Instructions: neg

neg	Arithmetic negate.
Syntax	`neg.`*type* `d, a;` *type* `= { .s16, .s32, .s64 };`
Description	Negate the sign of **a** and store the result in **d**.
Semantics	`d = -a;`
Notes	Only for signed integers.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`neg.s32 r0,a;`

Table 36. Integer Arithmetic Instructions: min

min	Find the minimum of two values.
Syntax	`min.`*type*` d, a, b;` `.`*type*` = { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Store the minimum of **a** and **b** in **d**.
Semantics	d = (a < b) ? a : b; // Integer (signed and unsigned)
Notes	Signed and unsigned differ.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`min.s32 r0,a,b;` `@p min.u16 h,i,j;`

Table 37. Integer Arithmetic Instructions: max

max	Find the maximum of two values.
Syntax	`max.`*type*` d, a, b;` `.`*type*` = { .u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Store the maximum of **a** and **b** in **d**.
Semantics	d = (a > b) ? a : b; // Integer (signed and unsigned)
Notes	Signed and unsigned differ.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`max.u32 d,a,b;` `max.s32 q,q,0;`

Table 38. Integer Arithmetic Instructions: popc

popc	Population count.
Syntax	`popc.type d, a;` `.type = { .b32, .b64 };`
Description	Count the number of one bits in **a** and place the resulting 'population count' in 32-bit destination register **d**. Operand **a** has the instruction type and destination **d** has type .u32.
Semantics	.u32 d = 0; while (a != 0) { if (a & 0x1) d++; a = a >> 1; }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	popc requires sm_20 or higher.
Examples	`popc.b32 d, a;` `popc.b64 cnt, X; // cnt is .u32`

Table 39. Integer Arithmetic Instructions: clz

clz	Count leading zeros.
Syntax	`clz.type d, a;` `.type = { .b32, .b64 };`
Description	Count the number of leading zeros in **a** starting with the most-significant bit and place the result in 32-bit destination register **d.** Operand **a** has the instruction type, and destination **d** has type .u32. For .b32 type, the number of leading zeros is between 0 and 32, inclusively. For .b64 type, the number of leading zeros is between 0 and 64, inclusively.
Semantics	.u32 d = 0; if (.*type* == .b32) { max = 32; mask = 0x80000000; } else { max = 64; mask = 0x8000000000000000; } while (d < max && (a&mask == 0)) { d++; a = a << 1; }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	clz requires sm_20 or higher.
Examples	`clz.b32 d, a;` `clz.b64 cnt, X; // cnt is .u32`

Table 40. Integer Arithmetic Instructions: bfind

bfind	Find most significant non-sign bit.
Syntax	`bfind.`*type*` d, a;` `bfind.shiftamt.`*type*` d, a;` `.`*type*` = { .u32, .u64,` `.s32, .s64 };`
Description	Find the bit position of the most significant non-sign bit in **a** and place the result in **d**. Operand **a** has the instruction type, and destination **d** has type .u32. For unsigned integers, bfind returns the bit position of the most significant "1". For signed integers, bfind returns the bit position of the most significant "0" for negative inputs and the most significant "1" for non-negative inputs. If .shiftamt is specified, bfind returns the shift amount needed to left-shift the found bit into the most-significant bit position. bfind returns 0xffffffff if no non-sign bit is found.
Semantics	msb = (.*type*==.u32 || .*type*==.s32) ? 31 : 63; // negate negative signed inputs if ((.*type*==.s32 || .*type*==.s64) && (a & (1<<msb))) { a = ~a; } .u32 d = 0xffffffff; for (.s32 i=msb; i>=0; i--) { if (a & (1<<i)) { d = i; break; } } if (.shiftamt && d != 0xffffffff) { d = msb - d; }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	bfind requires sm_20 or higher.
Examples	`bfind.u32 d, a;` `bfind.shiftamt.s64 cnt, X; // cnt is .u32`

Table 41. Integer Arithmetic Instructions: brev

brev	Bit reverse.
Syntax	`brev.`*type*` d, a;` `.`*type*` = { .b32, .b64 };`
Description	Perform bitwise reversal of input.
Semantics	msb = (.*type*==.b32) ? 31 : 63; for (i=0; i<=msb; i++) { d[i] = a[msb-i]; }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	brev requires sm_20 or higher.
Examples	`brev.b32 d, a;`

Table 42. Integer Arithmetic Instructions: bfe

bfe	Bit Field Extract.
Syntax	`bfe.`*type*` d, a, b, c;` `.`*type*` = { .u32, .u64,` `.s32, .s64 };`
Description	Extract bit field from **a** and place the zero or sign-extended result in **d**. Source **b** gives the bit field starting bit position, and source **c** gives the bit field length in bits. Operands **a** and **d** have the same type as the instruction type. Operands **b** and **c** are type .u32, but are restricted to the 8-bit value range 0..255. The sign bit of the extracted field is defined as: .u32, .u64: zero .s32, .s64: msb of input a if the extracted field extends beyond the msb of **a** msb of extracted field, otherwise If the bit field length is zero, the result is zero. The destination **d** is padded with the sign bit of the extracted field. If the start position is beyond the msb of the input, the destination **d** is filled with the replicated sign bit of the extracted field.
Semantics	msb = (.*type*==.u32 || .*type*==.s32) ? 31 : 63; pos = b & 0xff; // pos restricted to 0..255 range len = c & 0xff; // len restricted to 0..255 range if (.*type*==.u32 || .*type*==.u64 || len==0) sbit = 0; else sbit = a[min(pos+len-1,msb)]; d = 0; for (i=0; i<=msb; i++) { d[i] = (i<len && pos+i<=msb) ? a[pos+i] : sbit; }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	bfe requires sm_20 or higher.
Examples	`bfe.b32 d,a,start,len;`

Table 43. Integer Arithmetic Instructions: bfi

bfi	Bit Field Insert.
Syntax	`bfi.`*type* `f, a, b, c, d;` `.`*type* `= { .b32, .b64 };`
Description	Align and insert a bit field from **a** into **b**, and place the result in **f**. Source **c** gives the starting bit position for the insertion, and source **d** gives the bit field length in bits. Operands **a**, **b**, and **f** have the same type as the instruction type. Operands **c** and **d** are type .u32, but are restricted to the 8-bit value range 0..255. If the bit field length is zero, the result is **b**. If the start position is beyond the msb of the input, the result is **b**.
Semantics	msb = (.*type*==.b32) ? 31 : 63; pos = c & 0xff; // pos restricted to 0..255 range len = d & 0xff; // len restricted to 0..255 range f = b; for (i=0; i<len && pos+i<=msb; i++) { f[pos+i] = a[i]; }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	bfi requires sm_20 or higher.
Examples	`bfi.b32 d,a,b,start,len;`

8.7.2 Extended-Precision Integer Arithmetic Instructions

Instructions add.cc, addc, sub.cc, subc, mad.cc and madc reference an implicitly specified condition code register (CC) having a single carry flag bit (CC.CF) holding carry-in/carry-out or borrow-in/borrow-out. These instructions support extended-precision integer addition, subtraction, and multiplication. No other instructions access the condition code, and there is no support for setting, clearing, or testing the condition code. The condition code register is not preserved across calls and is mainly intended for use in straight-line code sequences for computing extended-precision integer addition, subtraction, and multiplication.

The extended-precision arithmetic instructions are:

- add.cc, addc
- sub.cc, subc
- mad.cc, madc

Table 44. Extended-Precision Arithmetic Instructions: add.cc

add.cc	Add two values with carry-out.
Syntax	`add.cc.`*`type`*`  d, a, b;` `.`*`type`*` = { .u32, .s32 };`
Description	Performs 32-bit integer addition and writes the carry-out value into the condition code register.
Semantics	d = a + b; carry-out written to CC.CF
Notes	No integer rounding modifiers. No saturation. Behavior is the same for unsigned and signed integers.
PTX ISA Notes	Introduced in PTX ISA version 1.2.
Target ISA Notes	Supported on all target architectures.
Examples	`@p  add.cc.u32   x1,y1,z1;   // extended-precision addition of` `@p  addc.cc.u32  x2,y2,z2;   // two 128-bit values` `@p  addc.cc.u32  x3,y3,z3;` `@p  addc.u32     x4,y4,z4;`

Table 45. Extended-Precision Arithmetic Instructions: addc

addc	Add two values with carry-in and optional carry-out.
Syntax	`addc{.cc}.`*`type`*`  d, a, b;` `.`*`type`*` = {.u32, .s32 };`
Description	Performs 32-bit integer addition with carry-in and optionally writes the carry-out value into the condition code register.
Semantics	d = a + b + CC.CF; if .cc specified, carry-out written to CC.CF
Notes	No integer rounding modifiers. No saturation. Behavior is the same for unsigned and signed integers.
PTX ISA Notes	Introduced in PTX ISA version 1.2.
Target ISA Notes	Supported on all target architectures.
Examples	`@p  add.cc.u32   x1,y1,z1;   // extended-precision addition of` `@p  addc.cc.u32  x2,y2,z2;   // two 128-bit values` `@p  addc.cc.u32  x3,y3,z3;` `@p  addc.u32     x4,y4,z4;`

Table 46. Extended-Precision Arithmetic Instructions: sub.cc

sub.cc	Subract one value from another, with borrow-out.
Syntax	`sub.cc.type d, a, b;` `.type = { .u32, .s32 };`
Description	Performs 32-bit integer subtraction and writes the borrow-out value into the condition code register.
Semantics	d = a – b; borrow-out written to CC.CF
Notes	No integer rounding modifiers. No saturation. Behavior is the same for unsigned and signed integers.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`@p sub.cc.u32 x1,y1,z1; // extended-precision subtraction` `@p subc.cc.u32 x2,y2,z2; // of two 128-bit values` `@p subc.cc.u32 x3,y3,z3;` `@p subc.u32 x4,y4,z4;`

Table 47. Extended-Precision Arithmetic Instructions: subc

subc	Subtract one value from another, withborrow-in and optional borrow-out.
Syntax	`subc{.cc}.type d, a, b;` `.type = {.u32, .s32 };`
Description	Performs 32-bit integer subtraction with borrow-in and optionally writes the borrow-out value into the condition code register.
Semantics	d = a - (b + CC.CF); if .cc specified, borrow-out written to CC.CF
Notes	No integer rounding modifiers. No saturation. Behavior is the same for unsigned and signed integers.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`@p sub.cc.u32 x1,y1,z1; // extended-precision subtraction` `@p subc.cc.u32 x2,y2,z2; // of two 128-bit values` `@p subc.cc.u32 x3,y3,z3;` `@p subc.u32 x4,y4,z4;`

Table 48. Extended-Precision Arithmetic Instructions: mad.cc

mad.cc	Multiply two values, extract high or low half of result, and add a third value with carry-out.
Syntax	`mad{.hi,.lo}.cc.type d, a, b, c;` `.type = { .u32, .s32 };`
Description	Multiplies two values, extracts either the high or low word of the 64-bit result, and adds a third value. Writes the 32-bit result to the destination register and the carry-out from the addition into the condition code register.
Semantics	t = a * b; d = t<63..32> + c; // for .hi variant d = t<31..0> + c; // for .lo variant carry-out from addition is written to CC.CF
Notes	Generally used in combination with madc and addc to implement extended-precision multi-word multiplication. See madc for an example.
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	Requires target sm_20 or higher.
Examples	`@p mad.lo.cc.u32 d,a,b,c;` `mad.lo.cc.u32 r,p,q,r;`

Table 49. Extended-Precision Arithmetic Instructions: madc

madc	Multiply two values, extract high or low half of result, and add a third value with carry-in and optional carry-out.
Syntax	`madc{.hi,.lo}{.cc}.type d, a, b, c;` `.type = {.u32, .s32 };`
Description	Multiplies two values, extracts either the high or low word of the 64-bit result, and adds a third value along with carry-in. Writes the 32-bit result to the destination register and optionally writes the carry-out from the addition into the condition code register.
Semantics	t = a * b; d = t<63..32> + c + CC.CF; // for .hi variant d = t<31..0> + c + CC.CF; // for .lo variant if .cc specified, carry-out from addition is written to CC.CF
Notes	Generally used in combination with mad.cc and addc to implement extended-precision multi-word multiplication. See example below.
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	Requires target sm_20 or higher.
Examples	`// extended-precision multiply: [r3,r2,r1,r0] = [r5,r4] * [r7,r6]` `mul.lo.u32 r0,r4,r6; // r0=(r4*r6).[31:0], no carry-out` `mul.hi.u32 r1,r4,r6; // r1=(r4*r6).[63:32], no carry-out` `mad.lo.cc.u32 r1,r5,r6,r1; // r1+=(r5*r6).[31:0], may carry-out` `madc.hi.u32 r2,r5,r6,0; // r2 =(r5*r6).[63:32]+carry-in, no carry-out` `mad.lo.cc.u32 r1,r4,r7,r1; // r1+=(r4*r7).[31:0], may carry-out` `madc.hi.cc.u32 r2,r4,r7,r2; // r2+=(r4*r7).[63:32]+carry-in, may carry-out` `addc.u32 r3,0,0; // r3 = carry-in, no carry-out` `mad.lo.cc.u32 r2,r5,r7,r2; // r2+=(r5*r7).[31:0], may carry-out` `madc.hi.u32 r3,r5,r7,r3; // r3+=(r5*r7).[63:32]+carry-in`

8.7.3 Floating-Point Instructions

Floating-point instructions operate on .f32 and .f64 register operands and constant immediate values. The floating-point instructions are:

- testp
- copysign
- add
- sub
- mul
- fma
- mad
- div
- abs
- neg
- min
- max
- rcp
- sqrt
- rsqrt
- sin
- cos
- lg2
- ex2

Instructions that support rounding modifiers are IEEE-754 compliant. Double-precision instructions support subnormal inputs and results. Single-precision instructions support subnormal inputs and results by default for sm_20 and subsequent targets, and flush subnormal inputs and results to sign-preserving zero for sm_1x targets. The optional .ftz modifier on single-precision instructions provides backward compatibility with sm_1x targets by flushing subnormal inputs and results to sign-preserving zero regardless of the target architecture.

Single-precision add, sub, mul, and mad support saturation of results to the range [0.0, 1.0], with NaNs being flushed to positive zero. NaN payloads are supported for double-precision instructions (except for rcp.approx.ftz.f64, which maps input NaNs to a canonical NaN). Single-precision instructions return an unspecified NaN. Note that future implementations may support NaN payloads for single-precision instructions, so PTX programs should not rely on the specific single-precision NaNs being generated.

Table 50 summarizes floating-point instructions in PTX.

Table 50. Summary of Floating-Point Instructions

Instruction	.rn	.rz	.rm	.rp	.ftz	.sat	Notes
{add,sub,mul}.*rnd*.f32	✓	✓	✓	✓	✓	✓	If no rounding modifier is specified, default is .rn and instructions may be folded into a multiply-add.
{add,sub,mul}.*rnd*.f64	✓	✓	✓	✓			If no rounding modifier is specified, default is .rn and instructions may be folded into a multiply-add.
mad.f32					✓	✓	.target sm_1x No rounding modifier.
{mad,fma}.*rnd*.f32	✓	✓	✓	✓	✓	✓	.target sm_20 or higher mad.f32 and fma.f32 are the same.
{mad,fma}.*rnd*.f64	✓	✓	✓	✓			mad.f64 and fma.f64 are the same.
div.full.f32					✓		No rounding modifier.
{div,rcp,sqrt}.approx.f32					✓		
rcp.approx.ftz.f64					✓		.target sm_20 or higher
{div,rcp,sqrt}.*rnd*.f32	✓	✓	✓	✓	✓		.target sm_20 or higher
{div,rcp,sqrt}.*rnd*.f64	✓	✓	✓	✓			.target sm_20 or higher
{abs,neg,min,max}.f32	n/a	n/a	n/a	n/a	✓		
{abs,neg,min,max}.f64	n/a	n/a	n/a	n/a			
rsqrt.approx.f32					✓		
rsqrt.approx.f64							
{sin,cos,lg2,ex2}.approx.f32					✓		

Table 51. Floating-Point Instructions: testp

testp	Test floating-point property.
Syntax	`testp.op.type p, a; // result is .pred` `.op = { .finite, .infinite,` `.number, .notanumber,` `.normal, .subnormal };` `.type = { .f32, .f64 };`
Description	testp tests common properties of floating-point numbers and returns a predicate value of 1 if True and 0 if False. testp.finite — True if the input is not infinite or NaN testp.infinite — True if the input is positive or negative infinity testp.number — True if the input is not NaN testp.notanumber — True if the input is NaN testp.normal — True if the input is a normal number (not NaN, not infinity) testp.subnormal — True if the input is a subnormal number (not NaN, not infinity) As a special case, positive and negative zero are considered normal numbers.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	testp requires sm_20 or higher.
Examples	`testp.notanumber.f32 isnan, f0;` `testp.infinite.f64 p, X;`

Table 52. Floating-Point Instructions: copysign

copysign	Copy sign of one input to another.
Syntax	`copysign.type d, a, b;` `.type = { .f32, .f64 };`
Description	Copy sign bit of **a** into value of **b**, and return the result as **d**.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	copysign requires sm_20 or higher.
Examples	`copysign.f32 x, y, z;` `copysign.f64 A, B, C;`

Table 53. Floating-Point Instructions: add

add	Add two values.
Syntax	`add{.rnd}{.ftz}{.sat}.f32 d, a, b;` `add{.rnd}.f64 d, a, b;` `.rnd = { .rn, .rz, .rm, .rp };`
Description	Performs addition and writes the resulting value into a destination register.
Semantics	d = a + b;
Notes	Rounding modifiers (default is .rn): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. add.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: add.f64 supports subnormal numbers. add.f32 flushes subnormal inputs and results to sign-preserving zero. Saturation modifier: add.sat.f32 clamps the result to [0.0, 1.0]. NaN results are flushed to +0.0f. An add instruction with an explicit rounding modifier treated conservatively by the code optimizer. An add instruction with no rounding modifier defaults to round-to-nearest-even and may be optimized aggressively by the code optimizer. In particular, mul/add sequences with no rounding modifiers may be optimized to use fused-multiply-add instructions on the target device.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	add.f32 supported on all target architectures. add.f64 requires sm_13 or higher. Rounding modifiers have the following target requirements: .rn, .rz available for all targets .rm, .rp for add.f64, requires sm_13 or higher. for add.f32, requires sm_20 or higher.
Examples	`@p add.rz.ftz.f32 f1,f2,f3;`

Table 54. Floating-Point Instructions: sub

sub	Subtract one value from another.
Syntax	`sub{.rnd}{.ftz}{.sat}.f32 d, a, b;` `sub{.rnd}.f64 d, a, b;` `.rnd = { .rn, .rz, .rm, .rp };`
Description	Performs subtraction and writes the resulting value into a destination register.
Semantics	d = a - b;
Notes	Rounding modifiers (default is .rn): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. sub.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: sub.f64 supports subnormal numbers. sub.f32 flushes subnormal inputs and results to sign-preserving zero. Saturation modifier: sub.sat.f32 clamps the result to [0.0, 1.0]. NaN results are flushed to +0.0f. A sub instruction with an explicit rounding modifier treated conservatively by the code optimizer. A sub instruction with no rounding modifier defaults to round-to-nearest-even and may be optimized aggressively by the code optimizer. In particular, mul/sub sequences with no rounding modifiers may be optimized to use fused-multiply-add instructions on the target device.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	sub.f32 supported on all target architectures. sub.f64 requires sm_13 or higher. Rounding modifiers have the following target requirements: .rn, .rz available for all targets .rm, .rp for sub.f64, requires sm_13 or higher. for sub.f32, requires sm_20 or higher.
Examples	`sub.f32 c,a,b;` `sub.rn.ftz.f32 f1,f2,f3;`

Table 55. Floating-Point Instructions: mul

mul	Multiply two values.
Syntax	`mul{.rnd}{.ftz}{.sat}.f32 d, a, b;` `mul{.rnd}.f64 d, a, b;` `.rnd = { .rn, .rz, .rm, .rp };`
Description	Compute the product of two values.
Semantics	d = a * b;
Notes	For floating-point multiplication, all operands must be the same size. Rounding modifiers (default is .rn): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. mul.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: mul.f64 supports subnormal numbers. mul.f32 flushes subnormal inputs and results to sign-preserving zero. Saturation modifier: mul.sat.f32 clamps the result to [0.0, 1.0]. NaN results are flushed to +0.0f. A mul instruction with an explicit rounding modifier treated conservatively by the code optimizer. A mul instruction with no rounding modifier defaults to round-to-nearest-even and may be optimized aggressively by the code optimizer. In particular, mul/add and mul/sub sequences with no rounding modifiers may be optimized to use fused-multiply-add instructions on the target device.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	mul.f32 supported on all target architectures. mul.f64 requires sm_13 or higher. Rounding modifiers have the following target requirements: .rn, .rz available for all targets .rm, .rp for mul.f64, requires sm_13 or higher. for mul.f32, requires sm_20 or higher.
Examples	`mul.ftz.f32 circumf,radius,pi // a single-precision multiply`

Table 56. Floating-Point Instructions: fma

fma	Fused multiply-add.
Syntax	`fma.rnd{.ftz}{.sat}.f32 d, a, b, c;` `fma.rnd.f64 d, a, b, c;` `.rnd = { .rn, .rz, .rm, .rp };`
Description	Performs a fused multiply-add with no loss of precision in the intermediate product and addition.
Semantics	d = a*b + c;
Notes	fma.f32 computes the product of **a** and **b** to infinite precision and then adds **c** to this product, again in infinite precision. The resulting value is then rounded to single precision using the rounding mode specified by *.rnd*. fma.f64 computes the product of **a** and **b** to infinite precision and then adds **c** to this product, again in infinite precision. The resulting value is then rounded to double precision using the rounding mode specified by *.rnd*. fma.f64 is the same as mad.f64. Rounding modifiers (no default): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. fma.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: fma.f64 supports subnormal numbers. fma.f32 is unimplemented for sm_1x targets. Saturation: fma.sat.f32 clamps the result to [0.0, 1.0]. NaN results are flushed to +0.0f.
PTX ISA Notes	fma.f64 introduced in PTX ISA version 1.4. fma.f32 introduced in PTX ISA version 2.0.
Target ISA Notes	fma.f32 requires sm_20 or higher. fma.f64 requires sm_13 or higher.
Examples	`fma.rn.ftz.f32 w,x,y,z;` `@p fma.rn.f64 d,a,b,c;`

Table 57. Floating-Point Instructions: mad

mad	Multiply two values and add a third value.
Syntax	`mad{.ftz}{.sat}.f32 d, a, b, c; // .target sm_1x` `mad.rnd{.ftz}{.sat}.f32 d, a, b, c; // .target sm_20` `mad.rnd.f64 d, a, b, c; // .target sm_13 and higher` `.rnd = { .rn, .rz, .rm, .rp };`
Description	Multiplies two values and adds a third, and then writes the resulting value into a destination register.
Semantics	d = a*b + c;
Notes	**For .target sm_20 and higher:** mad.f32 computes the product of **a** and **b** to infinite precision and then adds **c** to this product, again in infinite precision. The resulting value is then rounded to single precision using the rounding mode specified by *.rnd*. mad.f64 computes the product of **a** and **b** to infinite precision and then adds **c** to this product, again in infinite precision. The resulting value is then rounded to double precision using the rounding mode specified by *.rnd*. mad.{f32,f64} is the same as fma.{f32,f64}. **For .target sm_1x:** mad.f32 computes the product of **a** and **b** at double precision, and then the mantissa is truncated to 23 bits, but the exponent is preserved. Note that this is different from computing the product with mul, where the mantissa can be rounded and the exponent will be clamped. The exception for mad.f32 is when **c** = +/-0.0, mad.f32 is identical to the result computed using separate mul and add instructions. When JIT-compiled for SM 2.0 devices, mad.f32 is implemented as a fused multiply-add (i.e., fma.rn.ftz.f32). In this case, mad.f32 can produce slightly different numeric results and backward compatibility is not guaranteed in this case. mad.f64 computes the product of **a** and **b** to infinite precision and then adds **c** to this product, again in infinite precision. The resulting value is then rounded to double precision using the rounding mode specified by *.rnd*. Unlike mad.f32, the treatment of subnormal inputs and output follows IEEE 754 standard. mad.f64 is the same as fma.f64. Rounding modifiers (no default): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. mad.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: mad.f64 supports subnormal numbers. mad.f32 flushes subnormal inputs and results to sign-preserving zero. Saturation modifier: mad.sat.f32 clamps the result to [0.0, 1.0]. NaN results are flushed to +0.0f.
PTX ISA Notes	Introduced in PTX ISA version 1.0. In PTX ISA versions 1.4 and later, a rounding modifier is required for mad.f64. Legacy mad.f64 instructions having no rounding modifier will map to mad.rn.f64. In PTX ISA versions 2.0 and later, a rounding modifier is required for mad.f32 for sm_20 and higher targets.

Errata	mad.f32 requires a rounding modifier for sm_20 and higher targets. However for PTX ISA version 3.0 and earlier, ptxas does not enforce this requirement and mad.f32 silently defaults to mad.rn.f32. For PTX ISA version 3.1, ptxas generates a warning and defaults to mad.rn.f32, and in subsequent releases ptxas will enforce the requirement for PTX ISA version 3.2 and later.
Target ISA Notes	mad.f32 supported on all target architectures. mad.f64 requires sm_13 or higher. Rounding modifiers have the following target requirements: .rn,.rz,.rm,.rp for mad.f64, requires sm_13 or higher. .rn,.rz,.rm,.rp for mad.f32, requires sm_20 or higher.
Examples	`@p mad.f32 d,a,b,c;`

Table 58. Floating-Point Instructions: div

div	Divide one value by another.
Syntax	`div.approx{.ftz}.f32  d, a, b;  // fast, approximate divide` `div.full{.ftz}.f32  d, a, b;  // full-range approximate divide` `div.rnd{.ftz}.f32  d, a, b;  // IEEE 754 compliant rounding` `div.rnd.f64  d, a, b;  // IEEE 754 compliant rounding` `.rnd = { .rn, .rz, .rm, .rp };`
Description	Divides a by b, stores result in d.
Semantics	d = a / b;
Notes	**Fast, approximate single-precision divides:** div.approx.f32 implements a fast approximation to divide, computed as **d** = **a** * (1/**b**). For **b** in [2^{-126}, 2^{126}], the maximum ulp error is 2. div.full.f32 implements a relatively fast, full-range approximation that scales operands to achieve better accuracy, but is not fully IEEE 754 compliant and does not support rounding modifiers. The maximum ulp error is 2 across the full range of inputs. Subnormal inputs and results are flushed to sign-preserving zero. Fast, approximate division by zero creates a value of infinity (with same sign as a). **Divide with IEEE 754 compliant rounding:** Rounding modifiers (no default): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. div.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: div.f64 supports subnormal numbers. div.f32 flushes subnormal inputs and results to sign-preserving zero.
PTX ISA Notes	div.f32 and div.f64 introduced in PTX ISA version 1.0. Explicit modifiers .approx, .full, .ftz, and rounding introduced in PTX ISA version 1.4. For PTX ISA version 1.4 and later, one of .approx, .full, or *.rnd* is required. For PTX ISA versions 1.0 through 1.3, div.f32 defaults to div.approx.ftz.f32, and div.f64 defaults to div.rn.f64.
Target ISA Notes	div.approx.f32 and div.full.f32 supported on all target architectures. div.*rnd*.f32 requires sm_20 or higher. div.rn.f64 requires sm_13 or higher, or .target map_f64_to_f32. div.{rz,rm,rp}.f64 requires sm_20 or higher.
Examples	`div.approx.ftz.f32  diam,circum,3.14159;` `div.full.ftz.f32  x, y, z;` `div.rn.f64  xd, yd, zd;`

Table 59. Floating-Point Instructions: abs

abs	Absolute value.
Syntax	`abs{.ftz}.f32 d, a;` `abs.f64 d, a;`
Description	Take the absolute value of **a** and store the result in **d**.
Semantics	`d = |a|;`
Notes	Subnormal numbers: sm_20+: By default, subnormal numbers are supported. abs.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: abs.f64 supports subnormal numbers. abs.f32 flushes subnormal inputs and results to sign-preserving zero. NaN inputs yield an unspecified NaN. Future implementations may comply with the IEEE 754 standard by preserving payload and modifying only the sign bit.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	abs.f32 supported on all target architectures. abs.f64 requires sm_13 or higher.
Examples	`abs.ftz.f32 x,f0;`

Table 60. Floating-Point Instructions: neg

neg	Arithmetic negate.
Syntax	`neg{.ftz}.f32 d, a;` `neg.f64 d, a;`
Description	Negate the sign of **a** and store the result in **d**.
Semantics	`d = -a;`
Notes	Subnormal numbers: sm_20+: By default, subnormal numbers are supported. neg.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: neg.f64 supports subnormal numbers. neg.f32 flushes subnormal inputs and results to sign-preserving zero. NaN inputs yield an unspecified NaN. Future implementations may comply with the IEEE 754 standard by preserving payload and modifying only the sign bit.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	neg.f32 supported on all target architectures. neg.f64 requires sm_13 or higher.
Examples	`neg.ftz.f32 x,f0;`

Table 61. Floating-Point Instructions: min

min	Find the minimum of two values.
Syntax	`min{.ftz}.f32 d, a, b;` `min.f64 d, a, b;`
Description	Store the minimum of **a** and **b** in **d**.
Semantics	if (isNaN(a) && isNaN(b)) d = NaN; else if (isNaN(a)) d = b; else if (isNaN(b)) d = a; else d = (a < b) ? a : b;
Notes	Subnormal numbers: sm_20+: By default, subnormal numbers are supported. min.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: min.f64 supports subnormal numbers. min.f32 flushes subnormal inputs and results to sign-preserving zero.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	min.f32 supported on all target architectures. min.f64 requires sm_13 or higher.
Examples	`@p min.ftz.f32 z,z,x;` `min.f64 a,b,c;`

Table 62. Floating-Point Instructions: max

max	Find the maximum of two values.
Syntax	`max{.ftz}.f32 d, a, b;` `max.f64 d, a, b;`
Description	Store the maximum of **a** and **b** in **d**.
Semantics	if (isNaN(a) && isNaN(b)) d = NaN; else if (isNaN(a)) d = b; else if (isNaN(b)) d = a; else d = (a > b) ? a : b;
Notes	Subnormal numbers: sm_20+: By default, subnormal numbers are supported. max.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: max.f64 supports subnormal numbers. max.f32 flushes subnormal inputs and results to sign-preserving zero.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	max.f32 supported on all target architectures. max.f64 requires sm_13 or higher.
Examples	`max.ftz.f32 f0,f1,f2;` `max.f64 a,b,c;`

Table 63. Floating-Point Instructions: rcp

<table>
<tr><td>rcp</td><td>Take the reciprocal of a value.</td></tr>
<tr><td>Syntax</td><td>
<pre>rcp.approx{.ftz}.f32 d, a; // fast, approximate reciprocal
rcp.rnd{.ftz}.f32   d, a; // IEEE 754 compliant rounding
rcp.rnd.f64   d, a; // IEEE 754 compliant rounding</pre>
.rnd = { .rn, .rz, .rm, .rp };
</td></tr>
<tr><td>Description</td><td>Compute 1/a, store result in d.</td></tr>
<tr><td>Semantics</td><td>d = 1 / a;</td></tr>
<tr><td>Notes</td><td>
Fast, approximate single-precision reciprocal:
rcp.approx.f32 implements a fast approximation to reciprocal. The maximum absolute error is $2^{-23.0}$ over the range 1.0-2.0.
<table>
<tr><th>Input</th><th>Result</th></tr>
<tr><td>-Inf</td><td>-0.0</td></tr>
<tr><td>-subnormal</td><td>-Inf</td></tr>
<tr><td>-0.0</td><td>-Inf</td></tr>
<tr><td>+0.0</td><td>+Inf</td></tr>
<tr><td>+subnormal</td><td>+Inf</td></tr>
<tr><td>+Inf</td><td>+0.0</td></tr>
<tr><td>NaN</td><td>NaN</td></tr>
</table>
Reciprocal with IEEE 754 compliant rounding:
Rounding modifiers (no default):
.rn mantissa LSB rounds to nearest even
.rz mantissa LSB rounds towards zero
.rm mantissa LSB rounds towards negative infinity
.rp mantissa LSB rounds towards positive infinity

Subnormal numbers:
sm_20+: By default, subnormal numbers are supported.
rcp.ftz.f32 flushes subnormal inputs and results to sign-preserving zero.
sm_1x: rcp.f64 supports subnormal numbers.
rcp.f32 flushes subnormal inputs and results to sign-preserving zero.
</td></tr>
<tr><td>PTX ISA Notes</td><td>
rcp.f32 and rcp.f64 introduced in PTX ISA version 1.0. rcp.rn.f64 and explicit modifiers .approx and .ftz were introduced in PTX ISA version 1.4. General rounding modifiers were added in PTX ISA version 2.0.
For PTX ISA version 1.4 and later, one of .approx or .rnd is required.
For PTX ISA versions 1.0 through 1.3, rcp.f32 defaults to rcp.approx.ftz.f32, and rcp.f64 defaults to rcp.rn.f64.
</td></tr>
<tr><td>Target ISA Notes</td><td>
rcp.approx.f32 supported on all target architectures.
rcp.rnd.f32 requires sm_20 or higher.
rcp.rn.f64 requires sm_13 or higher, or .target map_f64_to_f32.
rcp.{rz,rm,rp}.f64 requires sm_20 or higher.
</td></tr>
<tr><td>Examples</td><td>
<pre>rcp.approx.ftz.f32 ri,r;
rcp.rn.ftz.f32   xi,x;
rcp.rn.f64   xi,x;</pre>
</td></tr>
</table>

Table 64. Floating-Point Instructions: rcp.approx.ftz.f64

<table>
<tr><td>rcp.approx.ftz.f64</td><td>Compute a fast, gross approximation to the reciprocal of a value.</td></tr>
<tr><td>Syntax</td><td><code>rcp.approx.ftz.f64 d, a;</code></td></tr>
<tr><td>Description</td><td>Compute a fast, gross approximation to the reciprocal as follows:
1. extract the most-significant 32 bits of .f64 operand a in 1.11.20 IEEE floating-point format (i.e., ignore the least-significant 32 bits of a),
2. compute an approximate .f64 reciprocal of this value using the most-significant 20 bits of the mantissa of operand a,
3. place the resulting 32-bits in 1.11.20 IEEE floating-point format in the most-significant 32-bits of destination d,and
4. zero the least significant 32 mantissa bits of .f64 destination d.</td></tr>
<tr><td>Semantics</td><td>tmp = a[63:32]; // upper word of a, 1.11.20 format
d[63:32] = 1.0 / tmp;
d[31:0] = 0x00000000;</td></tr>
<tr><td>Notes</td><td>rcp.approx.ftz.f64 implements a fast, gross approximation to reciprocal.

<table><tr><th>Input a[63:32]</th><th>Result d[63:32]</th></tr><tr><td>-Inf</td><td>-0.0</td></tr><tr><td>-subnormal</td><td>-Inf</td></tr><tr><td>-0.0</td><td>-Inf</td></tr><tr><td>+0.0</td><td>+Inf</td></tr><tr><td>+subnormal</td><td>+Inf</td></tr><tr><td>+Inf</td><td>+0.0</td></tr><tr><td>NaN</td><td>NaN</td></tr></table>
Input NaNs map to a canonical NaN with encoding 0x7fffffff00000000.
Subnormal inputs and results are flushed to sign-preserving zero.</td></tr>
<tr><td>PTX ISA Notes</td><td>rcp.approx.ftz.f64 introduced in PTX ISA version 2.1.</td></tr>
<tr><td>Target ISA Notes</td><td>rcp.approx.ftz.f64 requires sm_20 or higher.</td></tr>
<tr><td>Examples</td><td><code>rcp.ftz.f64 xi,x;</code></td></tr>
</table>

Table 65. Floating-Point Instructions: sqrt

<table>
<tr><td>sqrt</td><td>Take the square root of a value.</td></tr>
<tr><td>Syntax</td><td>
<pre>sqrt.approx{.ftz}.f32 d, a; // fast, approximate square root
sqrt.rnd{.ftz}.f32   d, a; // IEEE 754 compliant rounding
sqrt.rnd.f64   d, a; // IEEE 754 compliant rounding</pre>
.rnd = { .rn, .rz, .rm, .rp };
</td></tr>
<tr><td>Description</td><td>Compute sqrt(a) and store the result in d.</td></tr>
<tr><td>Semantics</td><td>d = sqrt(a);</td></tr>
<tr><td>Notes</td><td>
sqrt.approx.f32 implements a fast approximation to square root. The maximum absolute error for sqrt.approx.f32 is TBD.
<table>
<tr><th>Input</th><th>Result</th></tr>
<tr><td>-Inf</td><td>NaN</td></tr>
<tr><td>-normal</td><td>NaN</td></tr>
<tr><td>-subnormal</td><td>-0.0</td></tr>
<tr><td>-0.0</td><td>-0.0</td></tr>
<tr><td>+0.0</td><td>+0.0</td></tr>
<tr><td>+subnormal</td><td>+0.0</td></tr>
<tr><td>+Inf</td><td>+Inf</td></tr>
<tr><td>NaN</td><td>NaN</td></tr>
</table>
Square root with IEEE 754 compliant rounding:
Rounding modifiers (no default):
.rn mantissa LSB rounds to nearest even
.rz mantissa LSB rounds towards zero
.rm mantissa LSB rounds towards negative infinity
.rp mantissa LSB rounds towards positive infinity

Subnormal numbers:
sm_20+: By default, subnormal numbers are supported.
sqrt.ftz.f32 flushes subnormal inputs and results to sign-preserving zero.
sm_1x: sqrt.f64 supports subnormal numbers.
sqrt.f32 flushes subnormal inputs and results to sign-preserving zero.
</td></tr>
<tr><td>PTX ISA Notes</td><td>
sqrt.f32 and sqrt.f64 introduced in PTX ISA version 1.0. sqrt.rn.f64 and explicit modifiers .approx and .ftz were introduced in PTX ISA version 1.4. General rounding modifiers were added in PTX ISA version 2.0.
For PTX ISA version 1.4 and later, one of .approx or .rnd is required.
For PTX ISA versions 1.0 through 1.3, sqrt.f32 defaults to sqrt.approx.ftz.f32, and sqrt.f64 defaults to sqrt.rn.f64.
</td></tr>
<tr><td>Target ISA Notes</td><td>
sqrt.approx.f32 supported on all target architectures.
sqrt.rnd.f32 requires sm_20 or higher.
sqrt.rn.f64 requires sm_13 or higher, or .target map_f64_to_f32.
sqrt.{rz,rm,rp}.f64 requires sm_20 or higher.
</td></tr>
<tr><td>Examples</td><td>
<pre>sqrt.approx.ftz.f32 r,x;
sqrt.rn.ftz.f32   r,x;
sqrt.rn.f64   r,x;</pre>
</td></tr>
</table>

Table 66. Floating-Point Instructions: rsqrt

<table>
<tr><td>rsqrt</td><td>Take the reciprocal of the square root of a value.</td></tr>
<tr><td>Syntax</td><td>
<pre>rsqrt.approx{.ftz}.f32 d, a;
rsqrt.approx.f64   d, a;</pre>
</td></tr>
<tr><td>Description</td><td>Compute 1/sqrt(a) and store the result in d.</td></tr>
<tr><td>Semantics</td><td>d = 1/sqrt(a);</td></tr>
<tr><td>Notes</td><td>
rsqrt.approx implements an approximation to the reciprocal square root.
<table>
<tr><th>Input</th><th>Result</th></tr>
<tr><td>-Inf</td><td>NaN</td></tr>
<tr><td>-normal</td><td>NaN</td></tr>
<tr><td>-subnormal</td><td>-Inf</td></tr>
<tr><td>-0.0</td><td>-Inf</td></tr>
<tr><td>+0.0</td><td>+Inf</td></tr>
<tr><td>+subnormal</td><td>+Inf</td></tr>
<tr><td>+Inf</td><td>+0.0</td></tr>
<tr><td>NaN</td><td>NaN</td></tr>
</table>
The maximum absolute error for rsqrt.f32 is $2^{-22.4}$ over the range 1.0-4.0.
The maximum absolute error for rsqrt.f64 is TBD.

Subnormal numbers:
sm_20+: By default, subnormal numbers are supported.
rsqrt.ftz.f32 flushes subnormal inputs and results to sign-preserving zero.
sm_1x: rsqrt.f64 supports subnormal numbers.
rsqrt.f32 flushes subnormal inputs and results to sign-preserving zero.

Note that rsqrt.approx.f64 is emulated in software and are relatively slow.
</td></tr>
<tr><td>PTX ISA Notes</td><td>
rsqrt.f32 and rsqrt.f64 were introduced in PTX ISA version 1.0. Explicit modifiers .approx and .ftz were introduced in PTX ISA version 1.4.
For PTX ISA version 1.4 and later, the .approx modifier is required.
For PTX ISA versions 1.0 through 1.3, rsqrt.f32 defaults to rsqrt.approx.ftz.f32, and rsqrt.f64 defaults to rsqrt.approx.f64.
</td></tr>
<tr><td>Target ISA Notes</td><td>
rsqrt.f32 supported on all target architectures.
rsqrt.f64 requires sm_13 or higher.
</td></tr>
<tr><td>Examples</td><td>
<pre>rsqrt.approx.ftz.f32 isr, x;
rsqrt.approx.f64   ISR, X;</pre>
</td></tr>
</table>

Table 67. Floating-Point Instructions: sin

sin	Find the sine of a value.
Syntax	`sin.approx{.ftz}.f32 d, a;`
Description	Find the sine of the angle **a** (in radians).
Semantics	d = sin(a);
Floating-Point Notes	sin.approx.f32 implements a fast approximation to sine. **Input** / **Result** -Inf / NaN -subnormal / -0.0 -0.0 / -0.0 +0.0 / +0.0 +subnormal / +0.0 +Inf / NaN NaN / NaN The maximum absolute error is $2^{-20.9}$ in quadrant 00. Subnormal numbers: sm_20+: By default, subnormal numbers are supported. sin.ftz.f32 flushes subnormal inputs and results to sign-preserving zero. sm_1x: Subnormal inputs and results to sign-preserving zero.
PTX ISA Notes	sin.f32 introduced in PTX ISA version 1.0. Explicit modifiers .approx and .ftz introduced in PTX ISA version 1.4. For PTX ISA version 1.4 and later, the .approx modifier is required. For PTX ISA versions 1.0 through 1.3, sin.f32 defaults to sin.approx.ftz.f32.
Target ISA Notes	Supported on all target architectures.
Examples	`sin.approx.ftz.f32 sa, a;`

Table 68. Floating-Point Instructions: cos

<table>
<tr><td>cos</td><td>Find the cosine of a value.</td></tr>
<tr><td>Syntax</td><td><code>cos.approx{.ftz}.f32 d, a;</code></td></tr>
<tr><td>Description</td><td>Find the cosine of the angle a (in radians).</td></tr>
<tr><td>Semantics</td><td>d = cos(a);</td></tr>
<tr><td>Notes</td><td>cos.approx.f32 implements a fast approximation to cosine.

<table>
<tr><th>Input</th><th>Result</th></tr>
<tr><td>-Inf</td><td>NaN</td></tr>
<tr><td>-subnormal</td><td>+1.0</td></tr>
<tr><td>-0.0</td><td>+1.0</td></tr>
<tr><td>+0.0</td><td>+1.0</td></tr>
<tr><td>+subnormal</td><td>+1.0</td></tr>
<tr><td>+Inf</td><td>NaN</td></tr>
<tr><td>NaN</td><td>NaN</td></tr>
</table>

The maximum absolute error is $2^{-20.9}$ in quadrant 00.

Subnormal numbers:
sm_20+: By default, subnormal numbers are supported.
cos.ftz.f32 flushes subnormal inputs and results to sign-preserving zero.
sm_1x: Subnormal inputs and results to sign-preserving zero.</td></tr>
<tr><td>PTX ISA Notes</td><td>cos.f32 introduced in PTX ISA version 1.0. Explicit modifiers .approx and .ftz introduced in PTX ISA version 1.4.
For PTX ISA version 1.4 and later, the .approx modifier is required.
For PTX ISA versions 1.0 through 1.3, cos.f32 defaults to cos.approx.ftz.f32.</td></tr>
<tr><td>Target ISA Notes</td><td>Supported on all target architectures.</td></tr>
<tr><td>Examples</td><td><pre>cos.approx.ftz.f32 ca, a;</pre></td></tr>
</table>

Table 69. Floating-Point Instructions: lg2

<table>
<tr><td>lg2</td><td>Find the base-2 logarithm of a value.</td></tr>
<tr><td>Syntax</td><td><code>lg2.approx{.ftz}.f32 d, a;</code></td></tr>
<tr><td>Description</td><td>Determine the $\log_2$ of a.</td></tr>
<tr><td>Semantics</td><td>d = log(a) / log(2);</td></tr>
<tr><td>Notes</td><td>lg2.approx.f32 implements a fast approximation to $\log_2$(a).

<table>
<tr><th>Input</th><th>Result</th></tr>
<tr><td>-Inf</td><td>NaN</td></tr>
<tr><td>-subnormal</td><td>-Inf</td></tr>
<tr><td>-0.0</td><td>-Inf</td></tr>
<tr><td>+0.0</td><td>-Inf</td></tr>
<tr><td>+subnormal</td><td>-Inf</td></tr>
<tr><td>+Inf</td><td>+Inf</td></tr>
<tr><td>NaN</td><td>NaN</td></tr>
</table>

The maximum absolute error is $2^{-22.6}$ for mantissa.

Subnormal numbers:
sm_20+: By default, subnormal numbers are supported.
lg2.ftz.f32 flushes subnormal inputs and results to sign-preserving zero.
sm_1x: Subnormal inputs and results to sign-preserving zero.</td></tr>
<tr><td>PTX ISA Notes</td><td>lg2.f32 introduced in PTX ISA version 1.0. Explicit modifiers .approx and .ftz introduced in PTX ISA version 1.4.
For PTX ISA version 1.4 and later, the .approx modifier is required.
For PTX ISA versions 1.0 through 1.3, lg2.f32 defaults to lg2.approx.ftz.f32.</td></tr>
<tr><td>Target ISA Notes</td><td>Supported on all target architectures.</td></tr>
<tr><td>Examples</td><td><pre>lg2.approx.ftz.f32 la, a;</pre></td></tr>
</table>

Table 70. Floating-Point Instructions: ex2

<table>
<tr><td>ex2</td><td>Find the base-2 exponential of a value.</td></tr>
<tr><td>Syntax</td><td><code>ex2.approx{.ftz}.f32 d, a;</code></td></tr>
<tr><td>Description</td><td>Raise 2 to the power a.</td></tr>
<tr><td>Semantics</td><td>d = 2 ^ a;</td></tr>
<tr><td>Notes</td><td>ex2.approx.f32 implements a fast approximation to 2^a.

<table>
<tr><th>Input</th><th>Result</th></tr>
<tr><td>-Inf</td><td>+0.0</td></tr>
<tr><td>-subnormal</td><td>+1.0</td></tr>
<tr><td>-0.0</td><td>+1.0</td></tr>
<tr><td>+0.0</td><td>+1.0</td></tr>
<tr><td>+subnormal</td><td>+1.0</td></tr>
<tr><td>+Inf</td><td>+Inf</td></tr>
<tr><td>NaN</td><td>NaN</td></tr>
</table>

The maximum absolute error is $2^{-22.5}$ for fraction in the primary range.

Subnormal numbers:
sm_20+: By default, subnormal numbers are supported.
ex2.ftz.f32 flushes subnormal inputs and results to sign-preserving zero.
sm_1x: Subnormal inputs and results to sign-preserving zero.</td></tr>
<tr><td>PTX ISA Notes</td><td>ex2.f32 introduced in PTX ISA version 1.0. Explicit modifiers .approx and .ftz introduced in PTX ISA version 1.4.
For PTX ISA version 1.4 and later, the .approx modifier is required.
For PTX ISA versions 1.0 through 1.3, ex2.f32 defaults to ex2.approx.ftz.f32.</td></tr>
<tr><td>Target ISA Notes</td><td>Supported on all target architectures.</td></tr>
<tr><td>Examples</td><td><pre>ex2.approx.ftz.f32 xa, a;</pre></td></tr>
</table>

8.7.4 Comparison and Selection Instructions

The comparison select instructions are:

- set
- setp
- selp
- slct

As with single-precision floating-point instructions, the set, setp, and slct instructions support subnormal numbers for sm_20 and higher targets and flush single-precision subnormal inputs to sign-preserving zero for sm_1x targets. The optional .ftz modifier provides backward compatibility with sm_1x targets by flushing subnormal inputs and results to sign-preserving zero regardless of the target architecture.

Table 71. Comparison and Selection Instructions: set

set	Compare two numeric values with a relational operator, and optionally combine this result with a predicate value by applying a Boolean operator.
Syntax	`set.CmpOp{.ftz}.dtype.stype  d, a, b;` `set.CmpOp.BoolOp{.ftz}.dtype.stype  d, a, b, {!}c;` `.CmpOp = { eq, ne, lt, gt, ge, lo, ls, hi, hs,` `equ, neu, ltu, leu, gtu, geu, num, nan };` `.BoolOp = { and, or, xor };` `.dtype = { .u32, .s32, .f32 };` `.stype = { .b16, .b32, .b64,` `.u16, .u32, .u64,` `.s16, .s32, .s64,` `.f32, .f64 };`
Description	Compares two numeric values and optionally combines the result with another predicate value by applying a Boolean operator. If this result is True, 1.0f is written for floating-point destination types, and 0xffffffff is written for integer destination types. Otherwise, 0x00000000 is written. Operand **d** has type *.dtype*; operands **a** and **b** have type *.stype*; operand **c** has type .pred.
Semantics	t = (a CmpOp b) ? 1 : 0; if (isFloat(dtype)) d = BoolOp(t, c) ? 1.0f : 0x00000000; else d = BoolOp(t, c) ? 0xffffffff : 0x00000000;
Integer Notes	The signed and unsigned comparison operators are eq, ne, lt, le, gt, ge. For unsigned values, the comparison operators lo, ls, hi, and hs for lower, lower-or-same, higher, and higher-or-same may be used instead of lt, le, gt, ge, respectively. The untyped, bit-size comparisons are eq and ne.
Floating Point Notes	The ordered comparisons are eq, ne, lt, le, gt, ge. If either operand is NaN, the result is False. To aid comparison operations in the presence of NaN values, unordered versions are included: equ, neu, ltu, leu, gtu, geu. If both operands are numeric values (not NaN), then these comparisons have the same result as their ordered counterparts. If either operand is NaN, then the result of these comparisons is True. num returns True if both operands are numeric values (not NaN), and nan returns True if either operand is NaN. Subnormal numbers: sm_20+: By default, subnormal numbers are supported. set.ftz.*dtype*.f32 flushes subnormal inputs to sign-preserving zero. sm_1x: set.*dtype*.f64 supports subnormal numbers. set.*dtype*.f32 flushes subnormal inputs to sign-preserving zero. Modifier .ftz applies only to .f32 comparisons.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	set with .f64 source type requires sm_13 or higher.
Examples	`@p set.lt.and.f32.s32 d,a,b,r;` `set.eq.u32.u32 d,i,n;`

Table 72. Comparison and Selection Instructions: setp

setp	Compare two numeric values with a relational operator, and (optionally) combine this result with a predicate value by applying a Boolean operator.
Syntax	`setp.CmpOp{.ftz}.type  p[|q], a, b;` `setp.CmpOp.BoolOp{.ftz}.type  p[|q], a, b, {!}c;` `.CmpOp  = { eq, ne, lt, gt, ge, lo, ls, hi, hs,` `equ, neu, ltu, leu, gtu, geu, num, nan };` `.BoolOp = { and, or, xor };` `.type  = { .b16, .b32, .b64,` `.u16, .u32, .u64,` `.s16, .s32, .s64,` `.f32, .f64 };`
Description	Compares two values and combines the result with another predicate value by applying a Boolean operator. This result is written to the first destination operand. A related value computed using the complement of the compare result is written to the second destination operand. Applies to all numeric types. Operands **a** and **b** have type *.type*; operands **p**, **q**, and **c** have type .pred.
Semantics	t = (a CmpOp b) ? 1 : 0; p = BoolOp(t, c); q = BoolOp(!t, c);
Integer Notes	The signed and unsigned comparison operators are eq, ne, lt, le, gt, ge. For unsigned values, the comparison operators lo, ls, hi, and hs for lower, lower-or-same, higher, and higher-or-same may be used instead of lt, le, gt, ge, respectively. The untyped, bit-size comparisons are eq and ne.
Floating Point Notes	The ordered comparisons are eq, ne, lt, le, gt, ge. If either operand is NaN, the result is False. To aid comparison operations in the presence of NaN values, unordered versions are included: equ, neu, ltu, leu, gtu, geu. If both operands are numeric values (not NaN), then these comparisons have the same result as their ordered counterparts. If either operand is NaN, then the result of these comparisons is True. num returns True if both operands are numeric values (not NaN), and nan returns True if either operand is NaN. Subnormal numbers: sm_20+: By default, subnormal numbers are supported. setp.ftz.*dtype*.f32 flushes subnormal inputs to sign-preserving zero. sm_1x: setp.*dtype*.f64 supports subnormal numbers. setp.*dtype*.f32 flushes subnormal inputs to sign-preserving zero. Modifier .ftz applies only to .f32 comparisons.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	setp with .f64 source type requires sm_13 or higher.
Examples	`setp.lt.and.s32  p|q,a,b,r;` `@q  setp.eq.u32  p,i,n;`

Table 73. Comparison and Selection Instructions: selp

selp	Select between source operands, based on the value of the predicate source operand.
Syntax	`selp.`*type* `d, a, b, c;` `.`*type* `= { .b16, .b32, .b64,` `.u16, .u32, .u64,` `.s16, .s32, .s64,` `.f32, .f64 };`
Description	Conditional selection. If **c** is True, **a** is stored in **d**, **b** otherwise. Operands **d**, **a**, and **b** must be of the same type. Operand **c** is a predicate.
Semantics	d = (c == 1) ? a : b;
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	selp.f64 requires sm_13 or higher.
Examples	`selp.s32 r0,r,g,p;` `@q selp.f32 f0,t,x,xp;`

Table 74. Comparison and Selection Instructions: slct

slct	Select one source operand, based on the sign of the third operand.
Syntax	`slct.`*dtype*`.s32 d, a, b, c;` `slct{.ftz}.`*dtype*`.f32 d, a, b, c;` `.`*dtype* `= { .b16, .b32, .b64,` `.u16, .u32, .u64,` `.s16, .s32, .s64,` `.f32, .f64 };`
Description	Conditional selection. If **c** ≥ 0, **a** is stored in **d**, otherwise **b** is stored in **d**. Operands **d**, **a**, and **b** are treated as a bitsize type of the same width as the first instruction type; operand **c** must match the second instruction type (.s32 or .f32). The selected input is copied to the output without modification.
Semantics	d = (c >= 0) ? a : b;
Floating Point Notes	For .f32 comparisons, negative zero equals zero. Subnormal numbers: sm_20+: By default, subnormal numbers are supported. slct.ftz.*dtype*.f32 flushes subnormal values of operand **c** to sign-preserving zero, and operand **a** is selected. sm_1x: slct.*dtype*.f32 flushes subnormal values of operand **c** to sign-preserving zero, and operand **a** is selected. Modifier .ftz applies only to .f32 comparisons. If operand **c** is NaN, the comparison is unordered and operand **b** is selected.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	slct.f64 requires sm_13 or higher.
Examples	`slct.u32.s32 x, y, z, val;` `slct.ftz.u64.f32 A, B, C, fval;`

8.7.5 Logic and Shift Instructions

The logic and shift instructions are fundamentally untyped, performing bit-wise operations on operands of any type, provided the operands are of the same size. This permits bit-wise operations on floating point values without having to define a union to access the bits. Instructions and, or, xor, and not also operate on predicates.

The logical shift instructions are:

- and
- or
- xor
- not
- cnot
- shf
- shl
- shr

Table 75. Logic and Shift Instructions: and

and	Bitwise AND.
Syntax	`and.`*type* `d, a, b;` `.`*type* `= { .pred, .b16, .b32, .b64 };`
Description	Compute the bit-wise and operation for the bits in **a** and **b**.
Semantics	d = a & b;
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`and.b32 x,q,r;` `and.b32 sign,fpvalue,0x80000000;`

Table 76. Logic and Shift Instructions: or

or	Bitwise OR.
Syntax	`or.`*type* `d, a, b;` `.`*type* `= { .pred, .b16, .b32, .b64 };`
Description	Compute the bit-wise or operation for the bits in **a** and **b**.
Semantics	d = a | b;
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`or.b32 mask mask,0x00010001` `or.pred p,q,r;`

Table 77. Logic and Shift Instructions: xor

xor	Bitwise exclusive-OR (inequality).
Syntax	`xor.type d, a, b;` `.type = { .pred, .b16, .b32, .b64 };`
Description	Compute the bit-wise exclusive-or operation for the bits in **a** and **b**.
Semantics	d = a ^ b;
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`xor.b32 d,q,r;` `xor.b16 d,x,0x0001;`

Table 78. Logic and Shift Instructions: not

not	Bitwise negation; one's complement.
Syntax	`not.type d, a;` `.type = { .pred, .b16, .b32, .b64 };`
Description	Invert the bits in **a**.
Semantics	d = ~a;
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicates.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`not.b32 mask,mask;` `not.pred p,q;`

Table 79. Logic and Shift Instructions: cnot

cnot	C/C++ style logical negation.
Syntax	`cnot.type d, a;` `.type = { .b16, .b32, .b64 };`
Description	Compute the logical negation using C/C++ semantics.
Semantics	d = (a==0) ? 1 : 0;
Notes	The size of the operands must match, but not necessarily the type.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`cnot.b32 d,a;`

Table 80. Logic and Shift Instructions: shf

shf	Funnel shift.
Syntax	`shf.l.mode.b32 d, a, b, c; // left shift` `shf.r.mode.b32 d, a, b, c; // right shift` `.mode = { .clamp, .wrap };`
Description	Shift the 64-bit value formed by concatenating operands **a** and **b** left or right by the amount specified by the unsigned 32-bit value in **c**. Operand **b** holds bits 63:32 and operand **a** holds bits 31:0 of the 64-bit source value. The source is shifted left or right by the clamped or wrapped value in **c**. For shf.l, the *most-significant* 32-bits of the result are written into **d**; for shf.r, the *least-significant* 32-bits of the result are written into **d**.
Semantics	u32 n = (*.mode* == .clamp) ? min(c, 32) : c & 0x1f; switch (shf.*dir*) { // shift concatenation of [b, a] case shf.l: // extract 32 msbs u32 d = (b << n) | (a >> (32-n)); case shf.r: // extract 32 lsbs u32 d = (b << (32-n)) | (a >> n); }
Notes	Use funnel shift for multi-word shift operations and for rotate operations. The shift amount is limited to the range 0..32 in clamp mode and 0..31 in wrap mode, so shifting multi-word values by distances greater than 32 requires first moving 32-bit words, then using shf to shift the remaining 0..31 distance. To shift data sizes greater than 64 bits to the right, use repeated shf.r instructions applied to adjacent words, operating from least-significant word towards most-significant word. At each step, a single word of the shifted result is computed. The most-significant word of the result is computed using a shr.{u32,s32} instruction, which zero or sign fills based on the instruction type. To shift data sizes greater than 64 bits to the left, use repeated shf.l instructions applied to adjacent words, operating from most-significant word towards least-significant word. At each step, a single word of the shifted result is computed. The least-significant word of the result is computed using a shl instruction. Use funnel shift to perform 32-bit left or right rotate by supplying the same value for source arguments **a** and **b**.
PTX ISA Notes	Introduced in PTX ISA version 3.1.
Target ISA Notes	Requires sm_35 or higher.

Examples

```
shf.l.clamp.b32  r3,r1,r0,16;

// 128-bit left shift; n < 32
// [r7,r6,r5,r4] = [r3,r2,r1,r0] << n
shf.l.clamp.b32  r7,r2,r3,n;
shf.l.clamp.b32  r6,r1,r2,n;
shf.l.clamp.b32  r5,r0,r1,n;
shl.b32          r4,r0,n;

// 128-bit right shift, arithmetic; n < 32
// [r7,r6,r5,r4] = [r3,r2,r1,r0] >> n
shf.r.clamp.b32  r4,r0,r1,n;
shf.r.clamp.b32  r5,r1,r2,n;
shf.r.clamp.b32  r6,r2,r3,n;
shr.s32          r7,r3,n;     // result is sign-extended

shf.r.clamp.b32  r1,r0,r0,n;  // rotate right by n; n < 32
shf.l.clamp.b32  r1,r0,r0,n;  // rotate left by n; n < 32

// extract 32-bits from [r1,r0] starting at position n < 32
shf.r.clamp.b32  r0,r0,r1,n;
```

Table 81. Logic and Shift Instructions: shl

shl	Shift bits left, zero-fill on right.
Syntax	`shl.type d, a, b;` `.type = { .b16, .b32, .b64 };`
Description	Shift **a** left by the amount specified by unsigned 32-bit value in **b**.
Semantics	d = a << b;
Notes	Shift amounts greater than the register width N are clamped to N. The sizes of the destination and first source operand must match, but not necessarily the type. The **b** operand must be a 32-bit value, regardless of the instruction type.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`shl.b32 q,a,2;`

Table 82. Logic and Shift Instructions: shr

shr	Shift bits right, sign or zero fill on left.
Syntax	`shr.type d, a, b;` `.type = { .b16, .b32, .b64,` `.u16, .u32, .u64,` `.s16, .s32, .s64 };`
Description	Shift **a** right by the amount specified by unsigned 32-bit value in **b**. Signed shifts fill with the sign bit, unsigned and untyped shifts fill with 0.
Semantics	d = a >> b;
Notes	Shift amounts greater than the register width N are clamped to N. The sizes of the destination and first source operand must match, but not necessarily the type. The **b** operand must be a 32-bit value, regardless of the instruction type. Bit-size types are included for symmetry with shl.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`shr.u16 c,a,2;` `shr.s32 i,i,1;` `shr.b16 k,i,j;`

8.7.6 Data Movement and Conversion Instructions

These instructions copy data from place to place, and from state space to state space, possibly converting it from one format to another. mov, ld, ldu, and st operate on both scalar and vector types. The isspacep instruction is provided to query whether a generic address falls within a particular state space window. The cvta instruction converts addresses between generic and const, global, local, or shared state spaces.

Instructions ld, st, suld, and sust support optional cache operations.

The Data Movement and Conversion Instructions are:

- mov
- shfl
- prmt
- ld
- ldu
- st
- prefetch, prefetchu
- isspacep
- cvta
- cvt

8.7.6.1 Cache Operators

PTX ISA version 2.0 introduced optional cache operators on load and store instructions. The cache operators require a target architecture of sm_20 or higher. For sm_20 and higher, the cache operators have the following definitions and behavior.

Table 83. Cache Operators for Memory Load Instructions

Operator	Meaning
.ca	Cache at all levels, likely to be accessed again. The default load instruction cache operation is ld.ca, which allocates cache lines in all levels (L1 and L2) with normal eviction policy. Global data is coherent at the L2 level, but multiple L1 caches are not coherent for global data. If one thread stores to global memory via one L1 cache, and a second thread loads that address via a second L1 cache with ld.ca, the second thread may get stale L1 cache data, rather than the data stored by the first thread. The driver must invalidate global L1 cache lines between dependent grids of parallel threads. Stores by the first grid program are then correctly fetched by the second grid program issuing default ld.ca loads cached in L1.
.cg	Cache at global level (cache in L2 and below, not L1). Use ld.cg to cache loads only globally, bypassing the L1 cache, and cache only in the L2 cache. As a result of this request, any existing cache lines that match the requested address in L1 will be evicted.
.cs	Cache streaming, likely to be accessed once. The ld.cs load cached streaming operation allocates global lines with evict-first policy in L1 and L2 to limit cache pollution by temporary streaming data that may be accessed once or twice. When ld.cs is applied to a Local window address, it performs the ld.lu operation.
.lu	Last use. The ld.lu load last use operation, when applied to a local address, invalidates (discards) the local L1 line following the load, if the line is fully covered. The compiler / programmer may use ld.lu when restoring spilled registers and popping function stack frames to avoid needless write-backs of lines that will not be used again. The ld.lu instruction performs a load cached streaming operation (ld.cs) on global addresses.
.cv	Cache as volatile (consider cached system memory lines stale, fetch again). The ld.cv load cached volatile operation applied to a global System Memory address invalidates (discards) a matching L2 line and re-fetches the line on each new load, to allow the thread program to poll a SysMem location written by the CPU. A ld.cv to a frame buffer DRAM address is the same as ld.cs, evict-first.

Table 84. Cache Operators for Memory Store Instructions

Operator	Meaning
.wb	Cache write-back all coherent levels. The default store instruction cache operation is st.wb, which writes back cache lines of coherent cache levels with normal eviction policy. Data stored to local per-thread memory is cached in L1 and L2 with with write-back. However, sm_20 does NOT cache global store data in L1 because multiple L1 caches are not coherent for global data. Global stores bypass L1, and discard any L1 lines that match, regardless of the cache operation. Future GPUs may have globally-coherent L1 caches, in which case st.wb could write-back global store data from L1. If one thread stores to global memory, bypassing its L1 cache, and a second thread in a different SM later loads from that address via a different L1 cache with ld.ca, the second thread may get a hit on stale L1 cache data, rather than get the data from L2 or memory stored by the first thread. The driver must invalidate global L1 cache lines between dependent grids of thread arrays. Stores by the first grid program are then correctly missed in L1 and fetched by the second grid program issuing default ld.ca loads.
.cg	Cache at global level (cache in L2 and below, not L1). Use st.cg to cache global store data only globally, bypassing the L1 cache, and cache only in the L2 cache. In sm_20, st.cg is the same as st.wb for global data, but st.cg to local memory uses the L1 cache, and marks local L1 lines evict-first.
.cs	Cache streaming, likely to be accessed once. The st.cs store cached-streaming operation allocates cache lines with evict-first policy in L2 (and L1 if Local) to limit cache pollution by streaming output data.
.wt	Cache write-through (to system memory). The st.wt store write-through operation applied to a global System Memory address writes through the L2 cache, to allow a CPU program to poll a SysMem location written by the GPU with st.wt. Addresses not in System Memory use normal write-back.

Table 85. Data Movement and Conversion Instructions: mov

mov	Set a register variable with the value of a register variable or an immediate value. Take the non-generic address of a variable in global, local, or shared state space.
Syntax	`mov.type d, a;` `mov.type d, sreg;` `mov.type d, avar; // get address of variable` `mov.type d, avar+imm; // get address of variable with offset` `mov.type d, label; // get address of label` `mov.type d, fname; // get address of device function` `mov.u64 d, kernel; // get address of entry function` `.type = { .pred,` `.b16, .b32, .b64,` `.u16, .u32, .u64,` `.s16, .s32, .s64,` `.f32, .f64 };`
Description	Write register **d** with the value of **a**. Operand **a** may be a register, special register, variable with optional offset in an addressable memory space, label, or function name. For variables declared in .const, .global, .local, and .shared state spaces, mov places the non-generic address of the variable (i.e., the address of the variable in its state space) into the destination register. The generic address of a variable in const, global, local, or shared state space may be generated by first taking the address within the state space with mov and then converting it to a generic address using the cvta instruction; alternately, the generic address of a variable declared in const, global, local, or shared state space may be taken directly using the cvta instruction. Note that if the address of a device function parameter is moved to a register, the parameter will be copied onto the stack and the address will be in the local state space.
Semantics	d = a; d = *sreg*; d = &*avar*; // address is non-generic; i.e., within the variable's declared state space d = &*avar*+*imm*; d = &*label*;
Notes	Although only predicate and bit-size types are required, we include the arithmetic types for the programmer's convenience: their use enhances program readability and allows additional type checking.
PTX ISA Notes	Introduced in PTX ISA version 1.0. Taking the address of kernel entry functions requires PTX ISA version 3.1 or later. Kernel function addresses should only be used in the context of CUDA Dynamic Parallelism system calls. See the *CUDA Dynamic Parallelism Programming Guide* for details.
Target ISA Notes	mov.f64 requires sm_13 or higher. Taking the address of kernel entry functions requires sm_35 or higher.
Examples	`mov.f32 d,a;` `mov.u16 u,v;` `mov.f32 k,0.1;` `mov.u32 ptr, A; // move address of A into ptr` `mov.u32 ptr, A[5]; // move address of A[5] into ptr` `mov.u32 ptr, A+20; // move address with offset into ptr` `mov.u32 addr, myFunc; // get address of device function 'myFunc'` `mov.u64 kptr, main; // get address of entry function 'main'`

Table 86. Data Movement and Conversion Instructions: mov

mov	Move vector-to-scalar (pack) or scalar-to-vector (unpack).
Syntax	`mov.`*`type`*`  d, a;` *`.type`* `= { .b16, .b32, .b64 };`
Description	Write scalar register **d** with the packed value of vector register **a**, or write vector register **d** with the unpacked values from scalar register **a**. For bit-size types, mov may be used to pack vector elements into a scalar register or unpack sub-fields of a scalar register into a vector. Both the overall size of the vector and the size of the scalar must match the size of the instruction type.
Semantics	d = a.x | (a.y << 8) // pack two 8-bit elements into .b16 d = a.x | (a.y << 8) | (a.z << 16) | (a.w << 24) // pack four 8-bit elements into .b32 d = a.x | (a.y << 16) // pack two 16-bit elements into .b32 d = a.x | (a.y << 16) | (a.z << 32) | (a.w << 48) // pack four 16-bit elements into .b64 d = a.x | (a.y << 32) // pack two 32-bit elements into .b64 { d.x, d.y } = { a[0..7], a[8..15] } // unpack 8-bit elements from .b16 { d.x, d.y, d.z, d.w } = { a[0..7], a[8..15], a[16..23], a[24..31] } // unpack 8-bit elements from .b32 { d.x, d.y } = { a[0..15], a[16..31] } // unpack 16-bit elements from .b32 { d.x, d.y, d.z, d.w } = { a[0..15], a[16..31], a[32..47], a[48..63] } // unpack 16-bit elements from .b64 { d.x, d.y } = { a[0..31], a[32..63] } // unpack 32-bit elements from .b64
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.b32 %r1,{a,b};  // a,b have type .u16` `mov.b64 {lo,hi}, %x;  // %x is a double; lo,hi are .u32` `mov.b32 %r1,{x,y,z,w};  // x,y,z,w have type .b8` `mov.b32 {r,g,b,a},%r1;  // r,g,b,a have type .u8`

Table 87. Data Movement and Conversion Instructions: shfl

shfl	Register data shuffle within threads of a warp.
Syntax	`shfl.mode.b32 d[|p], a, b, c;` `.mode = { .up, .down, .bfly, .idx };`
Description	Exchange register data between threads of a warp. Each thread in the currently executing warp will compute a source lane index *j* based on input operands **b** and **c** and the *mode*. Operand **b** specifies a source lane or source lane offset, depending on the mode. Operand **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. If the computed source lane index *j* is in range, the thread will copy the input operand **a** from lane *j* into its own destination register **d**; otherwise, the thread will simply copy its own input **a** to destination **d**. The optional destination predicate **p** is set to True if the computed source lane *j* is in range, and otherwise set to False. Note that an out of range value of **b** may still result in a valid computed source lane index *j*. In this case, a data transfer occurs and the destination predicate **p** is True. Note that results are undefined in divergent control flow within a warp, if an active thread sources a register from an inactive thread.
Semantics	lane[4:0] = [Thread].laneid; // 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]; // get value of source register a if thread is active and guard predicate true, else zero if (isActive(Thread) && isGuardPredicateTrue(Thread)) { SourceA[lane] = a; } else { // Value of SourceA[lane] is unpredictable for inactive/predicated-off threads in warp } 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 d = SourceA[j]; // copy input a from lane j if (*dest predicate selected*) p = pval;
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	shfl requires sm_30 or higher.

Examples

```
    // Warp-level INCLUSIVE PLUS SCAN:
    //
    // Assumes input in following registers:
    //     - Rx  = sequence value for this thread
    //
    shfl.up.b32  Ry|p, Rx, 0x1,  0x0;
@p  add.f32      Rx, Ry, Rx;
    shfl.up.b32  Ry|p, Rx, 0x2,  0x0;
@p  add.f32      Rx, Ry, Rx;
    shfl.up.b32  Ry|p, Rx, 0x4,  0x0;
@p  add.f32      Rx, Ry, Rx;
    shfl.up.b32  Ry|p, Rx, 0x8,  0x0;
@p  add.f32      Rx, Ry, Rx;
    shfl.up.b32  Ry|p, Rx, 0x10, 0x0;
@p  add.f32      Rx, Ry, Rx;


    // Warp-level INCLUSIVE PLUS REVERSE-SCAN:
    //
    // Assumes input in following registers:
    //     - Rx  = sequence value for this thread
    //
    shfl.down.b32  Ry|p, Rx, 0x1,  0x1f;
@p  add.f32        Rx, Ry, Rx;
    shfl.down.b32  Ry|p, Rx, 0x2,  0x1f;
@p  add.f32        Rx, Ry, Rx;
    shfl.down.b32  Ry|p, Rx, 0x4,  0x1f;
@p  add.f32        Rx, Ry, Rx;
    shfl.down.b32  Ry|p, Rx, 0x8,  0x1f;
@p  add.f32        Rx, Ry, Rx;
    shfl.down.b32  Ry|p, Rx, 0x10, 0x1f;
@p  add.f32        Rx, Ry, Rx;


    // BUTTERFLY REDUCTION:
    //
    // Assumes input in following registers:
    //     - Rx  = sequence value for this thread
    //
    shfl.bfly.b32  Ry, Rx, 0x10, 0x1f;   // no predicate dest
    add.f32        Rx, Ry, Rx;
    shfl.bfly.b32  Ry, Rx, 0x8,  0x1f;
    add.f32        Rx, Ry, Rx;
    shfl.bfly.b32  Ry, Rx, 0x4,  0x1f;
    add.f32        Rx, Ry, Rx;
    shfl.bfly.b32  Ry, Rx, 0x2,  0x1f;
    add.f32        Rx, Ry, Rx;
    shfl.bfly.b32  Ry, Rx, 0x1,  0x1f;
    add.f32        Rx, Ry, Rx;
    //
    // All threads now hold sum in Rx
```

Table 88. Data Movement and Conversion Instructions: prmt

prmt — Permute bytes from register pair.

Syntax

```
prmt.b32{.mode}  d, a, b, c;

.mode = { .f4e, .b4e, .rc8, .ecl, .ecr, .rc16 };
```

Description

Pick four arbitrary bytes from two 32-bit registers, and reassemble them into a 32-bit destination register.

In the generic form (no mode specified), the permute control consists of four 4-bit selection values. The bytes in the two source registers are numbered from 0 to 7: {b, a} = {{b7, b6, b5, b4}, {b3, b2, b1, b0}}. For each byte in the target register, a 4-bit selection value is defined.

The 3 lsbs of the selection value specify which of the 8 source bytes should be moved into the target position. The msb defines if the byte value should be copied, or if the sign (msb of the byte) should be replicated over all 8 bits of the target position (sign extend of the byte value); msb=0 means copy the literal value; msb=1 means replicate the sign. Note that the sign extension is only performed as part of generic form.

Thus, the four 4-bit values fully specify an arbitrary byte permute, as a 16b permute code.

default mode	d.b3 source select	d.b2 source select	d.b1 source select	d.b0 source select
index	c[15:12]	c[11:8]	c[7:4]	c[3:0]

The more specialized form of the permute control uses the two lsb's of operand c (which is typically an address pointer) to control the byte extraction.

mode	selector c[1:0]	d.b3 source	d.b2 source	d.b1 source	d.b0 source
f4e (forward 4 extract)	0	3	2	1	0
	1	4	3	2	1
	2	5	4	3	2
	3	6	5	4	3
b4e (backward 4 extract)	0	5	6	7	0
	1	6	7	0	1
	2	7	0	1	2
	3	0	1	2	3
rc8 (replicate 8)	0	0	0	0	0
	1	1	1	1	1
	2	2	2	2	2
	3	3	3	3	3
ecl (edge clamp left)	0	3	2	1	0
	1	3	2	1	0
	2	3	2	1	0
	3	3	2	1	0
ecr (edge clamp right)	0	0	0	0	0
	1	1	1	1	0
	2	2	2	1	0
	3	3	2	1	0
rc16 (replicate 16)	0	1	0	1	0
	1	3	2	3	2
	2	1	0	1	0
	3	3	2	3	2

Semantics	tmp64 = (b<<32) | a; // create 8 byte source if (! *mode*) { ctl[0] = (c >> 0) & 0xf; ctl[1] = (c >> 4) & 0xf; ctl[2] = (c >> 8) & 0xf; ctl[3] = (c >> 12) & 0xf; } else { ctl[0] = ctl[1] = ctl[2] = ctl[3] = (c >> 0) & 0x3; } tmp[07:00] = ReadByte(mode, ctl[0], tmp64); tmp[15:08] = ReadByte(mode, ctl[1], tmp64); tmp[23:16] = ReadByte(mode, ctl[2], tmp64); tmp[31:24] = ReadByte(mode, ctl[3], tmp64);
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	prmt requires sm_20 or higher.
Examples	`prmt.b32      r1, r2, r3, r4;` `prmt.b32.f4e  r1, r2, r3, r4;`

Table 89. Data Movement and Conversion Instructions: ld

ld	Load a register variable from an addressable state space variable.
Syntax	`ld{.ss}{.cop}.type d, [a]; // load from address` `ld{.ss}{.cop}.vec.type d, [a]; // vector load from addr` `ld.volatile{.ss}.type d, [a]; // load from address` `ld.volatile{.ss}.vec.type d, [a]; // vector load from addr` `.ss = { .const, .global, .local, // state space` `.param, .shared };` `.cop = { .ca, .cg, .cs, .lu, .cv }; // cache operation` `.vec = { .v2, .v4 };` `.type = { .b8, .b16, .b32, .b64,` `.u8, .u16, .u32, .u64,` `.s8, .s16, .s32, .s64,` `.f32, .f64 };`
Description	Load register variable **d** from the location specified by the source address operand **a** in specified state space. If no state space is given, perform the load using generic addressing. In generic addressing, an address maps to global memory unless it falls within a window for const, local, or shared memory. Within these windows, an address maps to the corresponding location in cosnt, local, or shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. See Sections 5.1.6 and 7.1 for descriptions of the proper use of ld.param. The addressable operand **a** is one of: [var] the name of an addressable variable **var**, [reg] an integer or bit-size type register **reg** containing a byte address, [reg+immOff] a sum of register **reg** containing a byte address plus a constant integer byte offset (signed, 32-bit), or [immAddr] an immediate absolute byte address (unsigned, 32-bit). The address must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. ld.volatile may be used with .global and .shared spaces to inhibit optimization of references to volatile memory. This may be used, for example, to enforce sequential consistency between threads accessing shared memory. Generic addressing may be used with ld.volatile. Cache operations are not permitted with ld.volatile.
Semantics	d = a; // named variable a d = *a; // register d = *(a+immOff); // register-plus-offset d = *(immAddr); // immediate address
Notes	Destination **d** must be in the .reg state space. A destination register wider than the specified type may be used. The value loaded is sign-extended to the destination register width for signed integers, and is zero-extended to the destination register width for unsigned and bit-size types. See Table 24 for a description of these relaxed type-checking rules. .f16 data may be loaded using ld.b16, and then converted to .f32 or .f64 using cvt.
PTX ISA Notes	ld introduced in PTX ISA version 1.0. ld.volatile introduced in PTX ISA version 1.1. Generic addressing and cache operations introduced in PTX ISA version 2.0. Support for generic addressing of .const space added in PTX ISA version 3.1.

Target ISA Notes	ld.f64 requires sm_13 or higher. Generic addressing requires sm_20 or higher. Cache operations require sm_20 or higher.
Examples	`ld.global.f32 d,[a];` `ld.shared.v4.b32 Q,[p];` `ld.const.s32 d,[p+4];` `ld.local.b32 x,[p+-8]; // negative offset` `ld.local.b64 x,[240]; // immediate address` `ld.global.b16 %r,[fs]; // load .f16 data into 32-bit reg` `cvt.f32.f16 %r,%r; // up-convert f16 data to f32`

Table 90. Data Movement and Conversion Instructions: ld.global.nc

ld.global.nc	Load a register variable from global state space via non-coherent cache.
Syntax	`ld.global{.cop}.nc.type d, [a];` `ld.global{.cop}.nc.vec.type d, [a];` `.cop = { .ca, .cg, .cs }; // cache operation` `.vec = { .v2, .v4 };` `.type = { .b8, .b16, .b32, .b64,` `.u8, .u16, .u32, .u64,` `.s8, .s16, .s32, .s64,` `.f32, .f64 };`
Description	Load register variable **d** from the location specified by the source address operand **a** in the global state space, and optionally cache in non-coherent texture cache. Since the cache is non-coherent, the data should be read-only within the kernel's process. The texture cache is larger, has higher bandwidth, and longer latency than the global memory cache. For applications with sufficient parallelism to cover the longer latency, ld.global.nc should offer better performance than ld.global. The addressable operand **a** is one of: [var] the name of an addressable variable **var**, [reg] an integer or bit-size type register **reg** containing a byte address, [reg+immOff] a sum of register **reg** containing a byte address plus a constant integer byte offset (signed, 32-bit), or [immAddr] an immediate absolute byte address (unsigned, 32-bit). The address must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture.
Semantics	d = a; // named variable a d = *a; // register d = *(a+immOff); // register-plus-offset d = *(immAddr); // immediate address
Notes	Destination **d** must be in the .reg state space. A destination register wider than the specified type may be used. The value loaded is sign-extended to the destination register width for signed integers, and is zero-extended to the destination register width for unsigned and bit-size types. .f16 data may be loaded using ld.b16, and then converted to .f32 or .f64 using cvt.
PTX ISA Notes	Introduced in PTX ISA version 3.1.
Target ISA Notes	Requires sm_35 or higher.
Examples	`ld.global.nc.f32 d,[a];`

Table 91. Data Movement and Conversion Instructions: ldu

ldu	Load read-only data from an address that is common across threads in the warp.
Syntax	`ldu{.ss}.type d, [a]; // load from address` `ldu{.ss}.vec.type d, [a]; // vec load from address` `.ss = { .global }; // state space` `.vec = { .v2, .v4 };` `.type = { .b8, .b16, .b32, .b64,` `.u8, .u16, .u32, .u64,` `.s8, .s16, .s32, .s64,` `.f32, .f64 };`
Description	Load *read-only* data into register variable **d** from the location specified by the source address operand **a** in the global state space, where the address is guaranteed to be the same across all threads in the warp. If no state space is given, perform the load using generic addressing. In generic addressing, an address maps to global memory unless it falls within a window for const, local, or shared memory. Within these windows, an address maps to the corresponding location in const, local, or shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. For ldu, only generic addresses that map to global memory are legal. The addressable operand **a** is one of: [var] the name of an addressable variable **var**, [reg] a register **reg** containing a byte address, [reg+immOff] a sum of register **reg** containing a byte address plus a constant integer byte offset (signed, 32-bit), or [immAddr] an immediate absolute byte address (unsigned, 32-bit). The address must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The data at the specified address must be read-only. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. A register containing an address may be declared as a bit-size type or integer type.
Semantics	d = a; // named variable a d = *a; // register d = *(a+immOff); // register-plus-offset d = *(immAddr); // immediate address
Notes	Destination **d** must be in the .reg state space. A destination register wider than the specified type may be used. The value loaded is sign-extended to the destination register width for signed integers, and is zero-extended to the destination register width for unsigned and bit-size types. See Table 24 for a description of these relaxed type-checking rules. .f16 data may be loaded using ldu.b16, and then converted to .f32 or .f64 using cvt.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	ldu.f64 requires sm_13 or higher.
Examples	`ldu.global.f32 d,[a];` `ldu.global.b32 d,[p+4];` `ldu.global.v4.f32 Q,[p];`

Table 92. Data Movement and Conversion Instructions: st

st	Store a register variable to an addressable state space variable.
Syntax	st{.*ss*}{.*cop*}.*type* [a], b; // store to address st{.*ss*}{.*cop*}.*vec*.*type* [a], b; // vector store to addr st.volatile{.*ss*}.*type* [a], b; // store to address st.volatile{.*ss*}.*vec*.*type* [a], b; // vector store to addr *.ss* = {.global, .local, .param, .shared }; // state space *.cop* = { .wb, .cg, .cs, .wt }; // cache operation *.vec* = { .v2, .v4 }; *.type* = { .b8, .b16, .b32, .b64, .u8, .u16, .u32, .u64, .s8, .s16, .s32, .s64, .f32, .f64 };
Description	Store the value of register variable **b** in the location specified by the destination address operand **a** in specified state space. If no state space is given, perform the store using generic addressing. In generic addressing, an address maps to global memory unless it falls within a window for const, local, or shared memory. Within these windows, an address maps to the corresponding location in const, local, or shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. Stores to const memory are illegal. See Sections 5.1.6 and 7.1 for descriptions of the proper use of st.param. The addressable operand **a** is one of: [var] the name of an addressable variable var, [reg] an integer or bit-size type register reg containing a byte address, [reg+immOff] a sum of register reg containing a byte address plus a constant integer byte offset (signed, 32-bit), or [immAddr] an immediate absolute byte address (unsigned, 32-bit). The address must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. st.volatile may be used with .global and .shared spaces to inhibit optimization of references to volatile memory. This may be used, for example, to enforce sequential consistency between threads accessing shared memory. Generic addressing may be used with st.volatile. Cache operations are not permitted with st.volatile.
Semantics	d = a; // named variable d *d = a; // register *(d+immOffset) = a; // register-plus-offset *(immAddr) = a; // immediate address
Notes	Operand **b** must be in the .reg state space. A source register wider than the specified type may be used. The lower n bits corresponding to the instruction-type width are stored to memory. See Table 23 for a description of these relaxed type-checking rules. .f16 data resulting from a cvt instruction may be stored using st.b16.
PTX ISA Notes	st introduced in PTX ISA version 1.0. st.volatile introduced in PTX ISA version 1.1. Generic addressing and cache operations introduced in PTX ISA version 2.0.
Target ISA Notes	st.f64 requires sm_13 or higher. Generic addressing requires sm_20 or higher. Cache operations require sm_20 or higher.

Examples	

```
st.global.f32    [a],b;
st.local.b32     [q+4],a;
st.global.v4.s32 [p],Q;
st.local.b32     [q+-8],a; // negative offset
st.local.s32     [100],r7; // immediate address

cvt.f16.f32      %r,%r;    // %r is 32-bit register
st.b16           [fs],%r;  // store lower 16 bits
```

Table 93. Data Movement and Conversion Instructions: prefetch, prefetchu

prefetch **prefetchu**	Prefetch line containing generic address at specified level of memory hierarchy, in specified state space.
Syntax	`prefetch{.space}.level [a]; // prefetch to data cache` `prefetchu.L1 [a]; // prefetch to uniform cache` `.space = { .global, .local };` `.level = { .L1, .L2 };`
Description	The prefetch instruction brings the cache line containing the specified address in global or local memory state space into the specified cache level. If no state space is given, the prefetch uses generic addressing. In generic addressing, an address maps to global memory unless it falls within a window for const, local, or shared memory. Within these windows, an address maps to the corresponding location in const, local, or shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. The prefetchu instruction brings the cache line containing the specified generic address into the specified uniform cache level. The addressable operand **a** is one of: [var] the name of an addressable variable var, [reg] a register reg containing a byte address, [reg+immOff] a sum of register reg containing a byte address plus a constant integer byte offset (signed, 32-bit), or [immAddr] an immediate absolute byte address (unsigned, 32-bit). The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. A prefetch to a shared memory location performs no operation. A prefetch into the uniform cache requires a generic address, and no operation occurs if the address maps to a const, local, or shared memory location.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	prefetch and prefetchu require sm_20 or higher.
Examples	`prefetch.global.L1 [ptr];` `prefetchu.L1 [addr];`

Table 94. Data Movement and Conversion Instructions: isspacep

isspacep	Query whether a generic address falls within a specified state space window.
Syntax	`isspacep.`*`space`*`  p, a;    // result is .pred` `.`*`space`*` = { const, .global, .local, .shared };`
Description	Write predicate register **p** with 1 if generic address a falls within the specified state space window and with 0 otherwise. Destination **p** has type .pred; the source address operand must be of type .u32 or .u64.
PTX ISA Notes	Introduced in PTX ISA version 2.0. isspacep.const introduced in PTX ISA version 3.1.
Target ISA Notes	isspacep requires sm_20 or higher. Support for generic addressing of .const space added in PTX ISA version 3.1.
Examples	`isspacep.const  iscnst, cptr;` `isspacep.global  isglbl, gptr;` `isspacep.local  islcl,  lptr;` `isspacep.shared  isshrd, sptr;`

Table 95. Data Movement and Conversion Instructions: cvta

cvta	Convert address from const, global, local, or shared state space to generic, or vice-versa. Take the generic address of a variable declared in const, global, local, or shared state space.
Syntax	**`// convert const, global, local, or shared address to generic address`** `cvta.`*`space.size`*`  p, a;  // source address in register a` `cvta.`*`space.size`*`  p, `*`var`*`;  // get generic address of `*`var`* `cvta.`*`space.size`*`  p, `*`var+imm`*`;  // generic address of `*`var+offset`* **`// convert generic address to const, global, local, or shared address`** `cvta.to.`*`space.size`*`  p, a;` `.`*`space`*` = { .const, .global, .local, .shared };` `.`*`size`*`  = { .u32, .u64 };`
Description	Convert a const, global, local, or shared address to a generic address, or vice-versa. The source and destination addresses must be the same size. Use cvt.u32.u64 or cvt.u64.u32 to truncate or zero-extend addresses. For variables declared in const, global, local, or shared state space, the generic address of the variable may be taken using cvta. The source is either a register or a variable defined in const, global, local, or shared memory with an optional offset. When converting a generic address into a const, global, local, or shared address, the resulting address is undefined in cases where the generic address does not fall within the address window of the specified state space. A program may use isspacep to guard against such incorrect behavior.
PTX ISA Notes	Introduced in PTX ISA version 2.0. cvta.const and cvta.to.const introduced in PTX ISA version 3.1. **Note: the current implementation does not allow generic pointers to const space variables in programs that contain pointers to constant buffers passed as kernel parameters.**
Target ISA Notes	cvta requires sm_20 or higher.
Examples	`cvta.const.u32  ptr,cvar;` `cvta.local.u32  ptr,lptr;` `cvta.shared.u32  p,As+4;` `cvta.to.global.u32  p,gptr;`

Table 96. Data Movement and Conversion Instructions: cvt

cvt	Convert a value from one type to another.
Syntax	`cvt{.`*`irnd`*`}{.ftz}{.sat}.`*`dtype`*`.`*`atype`*`  d, a;  // integer rounding` `cvt{.`*`frnd`*`}{.ftz}{.sat}.`*`dtype`*`.`*`atype`*`  d, a;  // fp rounding` `.`*`irnd`*`  = { .rni, .rzi, .rmi, .rpi };` `.`*`frnd`*`  = { .rn,  .rz,  .rm,  .rp  };` `.`*`dtype`*` = .`*`atype`*` = { .u8, .u16, .u32, .u64,` `.s8, .s16, .s32, .s64,` `.f16, .f32, .f64 };`
Description	Convert between different types and sizes.
Semantics	d = convert(a);
Integer Notes	Integer rounding is required for float-to-integer conversions, and for same-size float-to-float conversions where the value is rounded to an integer. Integer rounding is illegal in all other instances. Integer rounding modifiers: .rni round to nearest integer, choosing even integer if source is equidistant between two integers. .rzi round to nearest integer in the direction of zero .rmi round to nearest integer in direction of negative infinity .rpi round to nearest integer in direction of positive infinity Subnormal numbers: sm_20+: By default, subnormal numbers are supported. For cvt.ftz.*dtype*.f32 float-to-integer conversions and cvt.ftz.f32.f32 float-to-float conversions with integer rounding, subnormal inputs are flushed to sign-preserving zero. sm_1x: For cvt.ftz.*dtype*.f32 float-to-integer conversions and cvt.ftz.f32.f32 float-to-float conversions with integer rounding, subnormal inputs are flushed to sign-preserving zero. The optional .ftz modifier may be specified in these cases for clarity. **Note**: In PTX ISA versions 1.4 and earlier, the cvt instruction did not flush single-precision subnormal inputs or results to zero if the destination type size was 64-bits. The compiler will preserve this behavior for legacy PTX code. Saturation modifier: .sat For integer destination types, .sat limits the result to MININT..MAXINT for the size of the operation. Note that saturation applies to both signed and unsigned integer types. The saturation modifier is allowed only in cases where the destination type's value range is not a superset of the source type's value range; i.e., the .sat modifier is illegal in cases where saturation is not possible based on the source and destination types. For float-to-integer conversions, the result is clamped to the destination range by default; i.e, .sat is redundant.

Floating Point Notes	Floating-point rounding is required for float-to-float conversions that result in loss of precision, and for integer-to-float conversions. Floating-point rounding is illegal in all other instances. Floating-point rounding modifiers: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity A floating-point value may be rounded to an integral value using the integer rounding modifiers (see Integer Notes). The operands must be of the same size. The result is an integral value, stored in floating-point format. Subnormal numbers: sm_20+: By default, subnormal numbers are supported. Modifier .ftz may be specified to flush single-precision subnormal inputs and results to sign-preserving zero. sm_1x: Single-precision subnormal inputs and results are flushed to sign-preserving zero. The optional .ftz modifier may be specified in these cases for clarity. **Note**: In PTX ISA versions 1.4 and earlier, the cvt instruction did not flush single-precision subnormal inputs or results to zero if either source or destination type was .f64. The compiler will preserve this behavior for legacy PTX code. Specifically, if the PTX ISA version is 1.4 or earlier, single-precision subnormal inputs and results are flushed to sign-preserving zero only for cvt.f32.f16, cvt.f16.f32, and cvt.f32.f32 instructions. Saturation modifier: .sat For floating-point destination types, .sat limits the result to the range [0.0, 1.0]. NaN results are flushed to positive zero. Applies to .f16, .f32, and .f64 types.
Notes	A source register wider than the specified type may be used. The lower n bits corresponding to the instruction-type width are used in the conversion. See Table 23 for a description of these relaxed type-checking rules. A destination register wider than the specified type may be used. The result of conversion is sign-extended to the destination register width for signed integers, and is zero-extended to the destination register width for unsigned, bit-size, and floating-point types. See Table 24 for a description of these relaxed type-checking rules.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	cvt to or from .f64 requires sm_13 or higher.
Examples	`cvt.f32.s32 f,i;` `cvt.s32.f64 j,r;  // float-to-int saturates by default` `cvt.rni.f32.f32 x,y; // round to nearest int, result is fp` `cvt.f32.f32 x,y;  // note .ftz behavior for sm_1x targets`

8.7.7 Texture Instructions

This section describes PTX instructions for accessing textures and samplers. PTX supports the following operations on texture and sampler descriptors:

- Static initialization of texture and sampler descriptors.
- Module-scope and per-entry scope definitions of texture and sampler descriptors.
- Ability to query fields within texture and sampler descriptors.

8.7.7.1 Texturing modes

For working with textures and samplers, PTX has two modes of operation. In the *unified mode,* texture and sampler information is accessed through a single .texref handle. In the *independent mode*, texture and sampler information each have their own handle, allowing them to be defined separately and combined at the site of usage in the program. The advantage of unified mode is that it allows 128 samplers per kernel, with the restriction that they correspond 1-to-1 with the 128 possible textures per kernel. The advantage of independent mode is that textures and samplers can be mixed and matched, but the number of samplers is greatly restricted to 16 per kernel.

The texturing mode is selected using .target options 'texmode_unified' and 'texmode_independent'. A PTX module may declare only one texturing mode. If no texturing mode is declared, the module is assumed to use unified mode.

Example: calculate an element's power contribution as element's power/total number of elements.

```
.target texmode_independent
.global .samplerref tsamp1 = { addr_mode_0 = clamp_to_border,
                               filter_mode = nearest
                             };
...
.entry compute_power
  ( .param .texref tex1 )
{
  txq.width.b32  r6, [tex1]; // get tex1's width
  txq.height.b32 r5, [tex1]; // get tex1's height
  tex.2d.v4.f32.f32  {r1,r2,r3,r4}, [tex1, tsamp1, {f1,f2}];
  mul.u32 r5, r5, r6;
  add.f32 r1, r1, r2;
  add.f32 r3, r3, r4;
  add.f32 r1, r1, r3;
  cvt.f32.u32 r5, r5;
  div.f32 r1, r1, r5;
}
```

8.7.7.2 Mipmaps

A *mipmap* is a sequence of textures, each of which is a progressively lower resolution representation of the same image. The height and width of each image, or level of detail (LOD), in the mipmap is a power of two smaller than the previous level. Mipmaps are used in graphics applications to improve rendering speed and reduce aliasing artifacts. For example, a high-resolution mipmap image is used for objects that are close to the user; lower-resolution images are used as the object appears farther away. Mipmap filtering modes are provided when switching between two levels of detail (LOD's) in order to avoid abrupt changes in visual fidelity.

Example: If the texture has a basic size of 256 by 256 pixels, then the associated mipmap set may contain a series of eight images, each one-fourth the total area of the previous one: 128×128 pixels, 64×64, 32×32, 16×16, 8×8, 4×4, 2×2, 1×1 (a single pixel). If, for example, a scene is rendering this texture in a space of 40×40 pixels, then either a scaled up version of the 32×32 (without trilinear interpolation) or an interpolation of the 64×64 and the 32×32 mipmaps (with trilinear interpolation) would be used.

The total number of LODs in a complete mipmap pyramid is calculated through the following equation:

```
numLODs = 1 + floor(log2(max(w, h, d)))
```

The finest LOD is called the base level and is the 0th level. The next (coarser) level is the 1st level, and so on. The coarsest level is the level of size (1 x 1 x 1). Each successively smaller mipmap level has half the {width, height, depth} of the previous level, but if this half value is a fractional value, it's rounded down to the next largest integer. Essentially, the size of a mipmap level can be specified as:

```
max(1, floor(w_b / 2^i)) x
  max(1, floor(h_b / 2^i)) x
    max(1, floor(d_b / 2^i))
```

where i is the ith level beyond the 0th level (the base level). And w_b, h_b and d_b are the width, height and depth of the base level respectively.

PTX support for mipmaps

The PTX tex instruction supports three modes for specifying the LOD: *base*, *level*, and *grad*ient. In base mode, the instruction always picks level 0. In level mode, an additional argument is provided to specify the LOD to fetch from. In grad mode, two floating-point vector arguments provide 'partials' (e.g. {ds/dx, dt/dx} and {ds/dy, dt/dy} for a 2d texture), which the tex instruction uses to compute the LOD.

These instructions provide access to texture memory.

- tex
- tld4
- txq

Table 97. Texture Instructions: tex

tex	Perform a texture memory lookup.
Syntax	`tex.geom.v4.dtype.ctype  d, [a, c];` `tex.geom.v4.dtype.ctype  d, [a, b, c];  // explicit sampler` `// mipmaps` `tex.base.geom.v4.dtype.ctype  d, [a, {b,} c];` `tex.level.geom.v4.dtype.ctype  d, [a, {b,} c], lod;` `tex.grad.geom.v4.dtype.ctype  d, [a, {b,} c], dPdx, dPdy;` `.geom  = { .1d, .2d, .3d, .a1d, .a2d, .cube, .acube };` `.dtype = { .u32, .s32, .f32 };` `.ctype = {  .s32, .f32 };  // .cube, .acube require .f32`
Description	**tex.{1d,2d,3d}** Texture lookup using a texture coordinate vector. The instruction loads data from the texture named by operand **a** at coordinates given by operand **c** into destination **d**. Operand **c** is a scalar or singleton tuple for 1d textures; is a two-element vector for 2d textures; and is a four-element vector for 3d textures, where the fourth element is ignored. An optional texture sampler **b** may be specified. If no sampler is specified, the sampler behavior is a property of the named texture. The instruction always returns a four-element vector of 32-bit values. Coordinates may be given in either signed 32-bit integer or 32-bit floating point form. A texture base address is assumed to be aligned to a 16 byte boundary, and the address given by the coordinate vector must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. **tex.{a1d,a2d}** Texture array selection, followed by texture lookup. The instruction first selects a texture from the texture array named by operand **a** using the index given by the first element of the array coordinate vector **c**. The instruction then loads data from the selected texture at coordinates given by the remaining elements of operand **c** into destination **d**. Operand **c** is a bit-size type vector or tuple containing an index into the array of textures followed by coordinates within the selected texture, as follows: • For 1d texture arrays, operand **c** has type .v2.b32. The first element is interpreted as an unsigned integer index (.u32) into the texture array, and the second element is interpreted as a 1d texture coordinate of type *.ctype*. • For 2d texture arrays, operand **c** has type .v4.b32. The first element is interpreted as an unsigned integer index (.u32) into the texture array, and the next two elements are interpreted as 2d texture coordinates of type *.ctype*. The fourth element is ignored. An optional texture sampler **b** may be specified. If no sampler is specified, the sampler behavior is a property of the named texture. The instruction always returns a four-element vector of 32-bit values. The texture array index is a 32-bit unsigned integer, and texture coordinate elements are 32-bit signed integer or floating point values. **tex.cube** *Cubemap* texture lookup. The instruction loads data from the cubemap texture named by operand **a** at coordinates given by operand **c** into destination **d**. Cubemap textures are special two-dimensional layered textures consisting of six layers that represent the faces of a cube. All layers in a cubemap are of the same size and are square (i.e., width equals height). When accessing a cubemap, the texture coordinate vector **c** has type .v4.f32, and comprises three floating-point coordinates (s, t, r) and a fourth padding argument which is ignored. Coordinates (s, t, r) are projected onto one of the six cube faces. The (s, t, r) coordinates can be thought of as a direction vector emanating from the center of the cube. Of the three coordinates (s, t, r), the coordinate of the largest magnitude (the major axis) selects the cube face. Then, the other two coordinates (the minor axes) are divided by the absolute value of the

	major axis to produce a new (s, t) coordinate pair to lookup into the selected cube face. An optional texture sampler **b** may be specified. If no sampler is specified, the sampler behavior is a property of the named texture. **tex.acube** Cubemap array selection, followed by cubemap lookup. The instruction first selects a cubemap texture from the cubemap array named by operand **a** using the index given by the first element of the array coordinate vector **c**. The instruction then loads data from the selected cubemap texture at coordinates given by the remaining elements of operand **c** into destination **d**. *Cubemap array* textures consist of an array of cubemaps, i.e. the total number of layers is a multiple of six. When accessing a cubemap array texture, the coordinate vector **c** has type .v4.b32. The first element is interpreted as an unsigned integer index (.u32) into the cubemap array, and the remaining three elements are interpreted as floating-point cubemap coordinates (s, t, r), used to lookup in the selected cubemap as described above. An optional texture sampler **b** may be specified. If no sampler is specified, the sampler behavior is a property of the named texture. **Mipmaps** .base (lod zero) Pick level 0 (base level). This is the default if no mipmap mode is specified. No additional arguments. .level (lod explicit) Requires an additional 32-bit scalar argument, lod, which contains the LOD to fetch from. The type of lod follows .ctype (either .s32 or .f32). .grad (lod gradient) Requires two .f32 vectors, dPdx and dPdy, that specify the partials. The vectors are singletons for 1d and a1d textures; are two-element vectors for 2d and a2d textures; and are four-element vectors for 3d textures, where the fourth element is ignored. Geometries .cube and .acube are not supported in this mode. **Indirect texture access** Beginning with PTX ISA version 3.1, indirect texture access is supported in unified mode for target architecture sm_20 or higher. In indirect access, operand **a** is a .u64 register holding the address of a .texref variable.
Notes	For compatibility with prior versions of PTX, the square brackets are not required and .v4 coordinate vectors are allowed for any geometry, with the extra elements being ignored.
PTX ISA Notes	Unified mode texturing introduced in PTX ISA version 1.0. Extension using opaque .texref and .samplerref types and independent mode texturing introduced in PTX ISA version 1.5. Texture arrays tex.{a1d,a2d} introduced in PTX ISA version 2.3. Cubemaps and cubemap arrays introduced in PTX ISA version 3.0. Support for mipmaps introduced in PTX ISA version 3.1. Indirect texture access introduced in PTX ISA version 3.1.
Target ISA Notes	Supported on all target architectures. The cubemap array geometry (.acube) requires sm_20 or higher. Mipmaps require sm_20 or higher. Indirect texture access requires sm_20 or higher.

Examples	

```
// Example of unified mode texturing
// - f4 is required to pad four-element tuple and is ignored
tex.3d.v4.s32.s32  {r1,r2,r3,r4}, [tex_a,{f1,f2,f3,f4}];

// Example of independent mode texturing
tex.1d.v4.s32.f32  {r1,r2,r3,r4}, [tex_a,smpl_x,{f1}];

// Example of 1D texture array, independent texturing mode
tex.a1d.v4.s32.s32 {r1,r2,r3,r4}, [tex_a,smpl_x,{idx,s1}];

// Example of 2D texture array, unified texturing mode
// - f3 is required to pad four-element tuple and is ignored
tex.a2d.v4.s32.f32 {r1,r2,r3,r4}, [tex_a,{idx,f1,f2,f3}];

// Example of cubemap array, unified textureing mode
tex.acube.v4.f32.f32 {r0,r1,r2,r3}, [tex_cuarray,{idx,f1,f2,f3}];
```

Table 98. Texture Instructions: tld4

tld4	Perform a texture fetch of the 4-texel bilerp footprint.
Syntax	`tld4.`*comp*`.2d.`*v4.dtype*`.f32 d, [a, c];` `tld4.`*comp*`.2d.`*v4.dtype*`.f32 d, [a, b, c]; // explicit sampler` *.comp* `= { .r, .g, .b, .a };` *.dtype* `= { .u32, .s32, .f32 };`
Description	Texture fetch of the 4-texel bilerp footprint using a texture coordinate vector. The instruction loads the bilerp footprint from the 2D texture named by operand **a** at coordinates given by operand **c** into vector destination **d**. The texture component fetched for each texel sample is specified by *.comp*. The four texel samples are placed into destination vector **d** in counter-clockwise order starting at lower left. Operand **c** specifies coordinates as a two-element, 32-bit floating-point vector. An optional texture sampler **b** may be specified. If no sampler is specified, the sampler behavior is a property of the named texture. A texture base address is assumed to be aligned to a 16 byte boundary, and the address given by the coordinate vector must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. **Indirect texture access** Beginning with PTX ISA version 3.1, indirect texture access is supported in unified mode for target architecture sm_20 or higher. In indirect access, operand **a** is a .u64 register holding the address of a .texref variable.
PTX ISA Notes	Introduced in PTX ISA version 2.2. Indirect texture access introduced in PTX ISA version 3.1.
Target ISA Notes	tld4 requires sm_20 or higher. Indirect texture access requires sm_20 or higher.
Examples	`//Example of unified mode texturing` `tld4.r.2d.v4.s32.f32 {r1,r2,r3,r4}, [tex_a,{f1,f2}];` `// Example of independent mode texturing` `tld4.r.2d.v4.u32.f32 {u1,u2,u3,u4}, [tex_a,smpl_x,{f1,f2}];`

Table 99. Texture Instructions: txq

<table>
<tr><td>txq</td><td>Query texture and sampler attributes.</td></tr>
<tr><td>Syntax</td><td>
<pre>
txq.tquery.b32   d, [a]; // texture attributes
txq.squery.b32   d, [a]; // sampler attributes

.tquery = { .width, .height, .depth,
            .channel_data_type, .channel_order,
            .normalized_coords };
.squery = { .force_unnormalized_coords, .filter_mode,
            .addr_mode_0, addr_mode_1, addr_mode_2 };
</pre>
</td></tr>
<tr><td>Description</td><td>
Query an attribute of a texture or sampler. Operand a is either a .texref or .samplerref variable, or a .u64 register.
<table>
<tr><th>Query:</th><th>Returns:</th></tr>
<tr><td>.width
.height
.depth</td><td>value in elements</td></tr>
<tr><td>.channel_data_type</td><td>Unsigned integer corresponding to source language’s channel data type enumeration. If the source language combines channel data type and channel order into a single enumeration type, that value is returned for both channel_data_type and channel_order queries.</td></tr>
<tr><td>.channel_order</td><td>Unsigned integer corresponding to source language’s channel order enumeration. If the source language combines channel data type and channel order into a single enumeration type, that value is returned for both channel_data_type and channel_order queries.</td></tr>
<tr><td>.normalized_coords</td><td>1 (True) or 0 (False).</td></tr>
<tr><td>.force_unnormalized_coords</td><td>1 (True) or 0 (False). Defined only for .samplerref variables in independent texture mode. Overrides the normalized_coords field of a .texref variable used with a .samplerref in a tex instruction.</td></tr>
<tr><td>.filter_mode</td><td>Integer from enum { nearest, linear }</td></tr>
<tr><td>.addr_mode_0
.addr_mode_1
.addr_mode_2</td><td>Integer from enum { wrap, mirror, clamp_ogl, clamp_to_edge, clamp_to_border }</td></tr>
</table>
Texture attributes are queried by supplying a .texref argument to txq. In unified mode, sampler attributes are also accessed via a .texref argument, and in independent mode sampler attributes are accessed via a separate .samplerref argument.

Indirect texture access
Beginning with PTX ISA version 3.1, indirect texture access is supported in unified mode for target architecture sm_20 or higher. In indirect access, operand a is a .u64 register holding the address of a .texref variable.
</td></tr>
<tr><td>PTX ISA Notes</td><td>
Introduced in PTX ISA version 1.5.
Channel data type and channel order queries were added in PTX ISA version 2.1.
The .force_unnormalized_coords query was added in PTX ISA version 2.2.
Indirect texture access introduced in PTX ISA version 3.1.
</td></tr>
<tr><td>Target ISA Notes</td><td>
Supported on all target architectures.
Indirect texture access requires sm_20 or higher.
</td></tr>
<tr><td>Examples</td><td>
<pre>
txq.width.b32   %r1, [tex_A];
txq.filter_mode.b32 %r1, [tex_A];   // unified mode
txq.addr_mode_0.b32 %r1, [smpl_B]; // independent mode
</pre>
</td></tr>
</table>

8.7.8 Surface Instructions

This section describes PTX instructions for accessing surfaces. PTX supports the following operations on surface descriptors:

- Static initialization of surface descriptors.
- Module-scope and per-entry scope definitions of surface descriptors.
- Ability to query fields within surface descriptors.

These instructions provide access to surface memory.

- suld
- sust
- sured
- suq

Table 100. Surface Instructions: suld

suld	Load from surface memory.
Syntax	`suld.b.geom{.cop}.vec.dtype.clamp  d, [a, b];    // unformatted` `.geom  = { .1d, .2d, .3d, .a1d, .a2d };` `.cop   = { .ca, .cg, .cs, .cv };  // cache operation` `.vec   = { none, .v2, .v4 };` `.dtype = { .b8 , .b16, .b32, .b64 };` `.clamp = { .trap, .clamp, .zero };`
Description	**suld.b.{1d,2d,3d}** Load from surface memory using a surface coordinate vector. The instruction loads data from the surface named by operand **a** at coordinates given by operand **b** into destination **d**. Operand **a** is a .surfref variable or .u64 register. Operand **b** is a scalar or singleton tuple for 1d surfaces; is a two-element vector for 2d surfaces; and is a four-element vector for 3d surfaces, where the fourth element is ignored. Coordinate elements are of type .s32. suld.b performs an unformatted load of binary data. The lowest dimension coordinate represents a byte offset into the surface and is not scaled, and the size of the data transfer matches the size of destination operand **d**. **suld.b.{a1d,a2d}** Surface layer selection, followed by a load from the selected surface. The instruction first selects a surface layer from the surface array named by operand **a** using the index given by the first element of the array coordinate vector **b**. The instruction then loads data from the selected surface at coordinates given by the remaining elements of operand **b** into destination **d**. Operand **a** is a .surfref variable or .u64 register. Operand **b** is a bit-size type vector or tuple containing an index into the array of surfaces followed by coordinates within the selected surface, as follows: For 1d surface arrays, operand **b** has type .v2.b32. The first element is interpreted as an unsigned integer index (.u32) into the surface array, and the second element is interpreted as a 1d surface coordinate of type .s32. For 2d surface arrays, operand **b** has type .v4.b32. The first element is interpreted as an unsigned integer index (.u32) into the surface array, and the next two elements are interpreted as 2d surface coordinates of type .s32. The fourth element is ignored. A surface base address is assumed to be aligned to a 16 byte boundary, and the address given by the coordinate vector must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The .clamp field specifies how to handle out-of-bounds addresses: .trap causes an execution trap on out-of-bounds addresses .clamp loads data at the nearest surface location (sized appropriately) .zero loads zero for out-of-bounds addresses **Indirect surface access** Beginning with PTX ISA version 3.1, indirect surface access is supported for target architecture sm_20 or higher. In indirect access, operand **a** is a .u64 register holding the address of a .surfref variable.
PTX ISA Notes	suld.b.trap introduced in PTX ISA version 1.5. Additional clamp modifiers and cache operations introduced in PTX ISA version 2.0. suld.b.3d and suld.b.{a1d,a2d} introduced in PTX ISA version 3.0. Indirect surface access introduced in PTX ISA version 3.1.
Target ISA Notes	suld.b supported on all target architectures. sm_1x targets support only the .trap clamping modifier. suld.3d and suld.{a1d,a2d} require sm_20 or higher.

	Indirect surface access requires sm_20 or higher. Cache operations require sm_20 or higher.
Examples	`suld.b.1d.v4.b32.trap  {s1,s2,s3,s4}, [surf_B, {x}];` `suld.b.3d.v2.b64.trap  {r1,r2}, [surf_A, {x,y,z,w}];` `suld.b.a1d.v2.b32  {r0,r1}, [surf_C, {idx,x}];` `suld.b.a2d.b32  r0, [surf_D, {idx,x,y,z}];  // z ignored`

Table 101. Surface Instructions: sust

sust	Store to surface memory.
Syntax	`sust.b.{1d,2d,3d}{.cop}.vec.ctype.clamp [a, b], c; // unformatted` `sust.p.{1d,2d,3d}.vec.b32.clamp [a, b], c; // formatted` `sust.b.{a1d,a2d}{.cop}.vec.ctype.clamp [a, b], c; // unformatted` `.cop = { .wb, .cg, .cs, .wt }; // cache operation` `.vec = { none, .v2, .v4 };` `.ctype = { .b8 , .b16, .b32, .b64 };` `.clamp = { .trap, .clamp, .zero };`
Description	**sust.{1d,2d,3d}** Store to surface memory using a surface coordinate vector. The instruction stores data from operand **c** to the surface named by operand **a** at coordinates given by operand **b**. Operand **a** is a .surfref variable or .u64 register. Operand **b** is a scalar or singleton tuple for 1d surfaces; is a two-element vector for 2d surfaces; and is a four-element vector for 3d surfaces, where the fourth element is ignored. Coordinate elements are of type .s32. **sust.b** performs an unformatted store of binary data. The lowest dimension coordinate represents a byte offset into the surface and is not scaled. The size of the data transfer matches the size of source operand **c**. **sust.p** performs a formatted store of a vector of 32-bit data values to a surface sample. The source vector elements are interpreted left-to-right as R, G, B, and A surface components. These elements are written to the corresponding surface sample components. Source elements that do not occur in the surface sample are ignored. Surface sample components that do not occur in the source vector will be written with an unpredictable value. The lowest dimension coordinate represents a sample offset rather than a byte offset. The source data interpretation is based on the surface sample format as follows: If the surface format contains UNORM, SNORM, or FLOAT data, then .f32 is assumed; if the surface format contains UINT data, then .u32 is assumed; if the surface format contains SINT data, then .s32 is assumed. The source data is then converted from this type to the surface sample format. **sust.b.{a1d,a2d}** Surface layer selection, followed by an unformatted store to the selected surface. The instruction first selects a surface layer from the surface array named by operand **a** using the index given by the first element of the array coordinate vector **b**. The instruction then stores the data in operand **c** to the selected surface at coordinates given by the remaining elements of operand **b**. Operand **a** is a .surfref variable or .u64 register. Operand **b** is a bit-size type vector or tuple containing an index into the array of surfaces followed by coordinates within the selected surface, as follows: For 1d surface arrays, operand **b** has type .v2.b32. The first element is interpreted as an unsigned integer index (.u32) into the surface array, and the second element is interpreted as a 1d surface coordinate of type .s32. For 2d surface arrays, operand **b** has type .v4.b32. The first element is interpreted as an unsigned integer index (.u32) into the surface array, and the next two elements are interpreted as 2d surface coordinates of type .s32. The fourth element is ignored. A surface base address is assumed to be aligned to a 16 byte boundary, and the address given by the coordinate vector must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The .clamp field specifies how to handle out-of-bounds addresses: .trap causes an execution trap on out-of-bounds addresses .clamp stores data at the nearest surface location (sized appropriately) .zero drops stores to out-of-bounds addresses **Indirect surface access**

	Beginning with PTX ISA version 3.1, indirect surface access is supported for target architecture sm_20 or higher. In indirect access, operand **a** is a .u64 register holding the address of a .surfref variable.
PTX ISA Notes	sust.b.trap introduced in PTX ISA version 1.5. sust.p, additional clamp modifiers, and cache operations introduced in PTX ISA version 2.0. sust.b.3d and sust.b.{a1d,a2d} introduced in PTX ISA version 3.0. Indirect surface access introduced in PTX ISA version 3.1.
Target ISA Notes	sust.b supported on all target architectures. sm_1x targets support only the .trap clamping modifier. sust.3d and sust.{a1d,a2d} require sm_20 or higher. sust.p requires sm_20 or higher. Indirect surface access requires sm_20 or higher. Cache operations require sm_20 or higher.
Examples	`sust.p.1d.v4.b32.trap [surf_B, {x}], {f1,f2,f3,f4};` `sust.b.3d.v2.b64.trap [surf_A, {x,y,z,w}], {r1,r2};` `sust.b.a1d.v2.b64 [surf_C, {idx,x}], {r1,r2};` `sust.b.a2d.b32 [surf_D, {idx,x,y,z}], r0; // z ignored`

Table 102. Surface Instructions: sured

sured	Reduction in surface memory.
Syntax	`sured.b.op.geom.ctype.clamp [a,b],c; // byte addressing` `sured.p.op.geom.ctype.clamp [a,b],c; // sample addressing` `.op = { .add, .min, .max, .and, .or };` `.geom = { .1d, .2d, .3d };` `.ctype = { .u32, .u64, .s32, .b32 }; // for sured.b` `.ctype = { .b32 }; // for sured.p` `.clamp = { .trap, .clamp, .zero };`
Description	Reduction to surface memory using a surface coordinate vector. The instruction performs a reduction operation with data from operand **c** to the surface named by operand **a** at coordinates given by operand **b**. Operand **a** is a .surfref variable or .u64 register. Operand **b** is a scalar or singleton tuple for 1d surfaces; is a two-element vector for 2d surfaces; and is a four-element vector for 3d surfaces, where the fourth element is ignored. Coordinate elements are of type .s32. sured.b performs an unformatted reduction on .u32, .s32, .b32, or .u64 data. The lowest dimension coordinate represents a byte offset into the surface and is not scaled. Operation **add** applies to .u32, .u64, and .s32 types; **min** and **max** apply to .u32 and .s32 types; operations **and** and **or** apply to .b32 type. sured.p performs a reduction on sample-addressed 32-bit data. The lowest dimension coordinate represents a sample offset rather than a byte offset. The instruction type is restricted to .b32, and the data is interpreted as .s32 or .u32 based on the surface sample format as follows: if the surface format contains UINT data, then .u32 is assumed; if the surface format contains SINT data, then .s32 is assumed. A surface base address is assumed to be aligned to a 16 byte boundary, and the address given by the coordinate vector must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The .clamp field specifies how to handle out-of-bounds addresses: .trap causes an execution trap on out-of-bounds addresses .clamp performs reduction at the nearest surface location (sized appropriately) .zero drops operations to out-of-bounds addresses **Indirect surface access** Beginning with PTX ISA version 3.1, indirect surface access is supported for target architecture sm_20 or higher. In indirect access, operand **a** is a .u64 register holding the address of a .surfref variable.
PTX ISA Notes	Introduced in PTX ISA version 2.0. Indirect surface access introduced in PTX ISA version 3.1.
Target ISA Notes	sured requires sm_20 or higher. Indirect surface access requires sm_20 or higher.
Examples	`sured.b.add.2d.u32.trap [surf_A, {x,y}], r1;` `sured.p.min.1d.b32.trap [surf_B, {x}], r1;`

Table 103. Surface Instructions: suq

<table>
<tr><td>suq</td><td>Query a surface attribute.</td></tr>
<tr><td>Syntax</td><td>suq.query.b32 d, [a];
.query = { .width, .height, .depth, .channel_data_type, .channel_order };</td></tr>
<tr><td>Description</td><td>Query an attribute of a surface. Operand a is a .surfref variable or a .u64 register.

<table>
<tr><th>Query:</th><th>Returns:</th></tr>
<tr><td>.width
.height
.depth</td><td>value in elements</td></tr>
<tr><td>.channel_data_type</td><td>Unsigned integer corresponding to source language’s channel data type enumeration. If the source language combines channel data type and channel order into a single enumeration type, that value is returned for both channel_data_type and channel_order queries.</td></tr>
<tr><td>.channel_order</td><td>Unsigned integer corresponding to source language’s channel order enumeration. If the source language combines channel data type and channel order into a single enumeration type, that value is returned for both channel_data_type and channel_order queries.</td></tr>
</table>

Indirect surface access
Beginning with PTX ISA version 3.1, indirect surface access is supported for target architecture sm_20 or higher. In indirect access, operand a is a .u64 register holding the address of a .surfref variable.</td></tr>
<tr><td>PTX ISA Notes</td><td>Introduced in PTX ISA version 1.5.
Channel data type and channel order queries added in PTX ISA version 2.1.
Indirect surface access introduced in PTX ISA version 3.1.</td></tr>
<tr><td>Target ISA Notes</td><td>Supported on all target architectures.
Indirect surface access requires sm_20 or higher.</td></tr>
<tr><td>Examples</td><td>suq.width.b32 %r1, [surf_A];</td></tr>
</table>

8.7.9 Control Flow Instructions

The following PTX instructions and syntax are for controlling execution in a PTX program:

- { }
- @
- bra
- call
- ret
- exit

Table 104. Control Flow Instructions: { }

{ }	Instruction grouping.
Syntax	`{ instructionList }`
Description	The curly braces create a group of instructions, used primarily for defining a function body. The curly braces also provide a mechanism for determining the scope of a variable: any variable declared within a scope is not available outside the scope.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`{ add.s32 a,b,c; mov.s32 d,a; }`

Table 105. Control Flow Instructions: @

@	Predicated execution.
Syntax	`@{!}p instruction;`
Description	Execute an instruction or instruction block for threads that have the guard predicate True. Threads with a False guard predicate do nothing.
Semantics	If {!}p then instruction
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`setp.eq.f32 p,y,0; // is y zero?` `@!p div.f32 ratio,x,y // avoid division by zero` `@q bra L23; // conditional branch`

Table 106. Control Flow Instructions: bra

bra	Branch to a target and continue execution there.
Syntax	`@p bra{.uni} tgt; // direct branch, tgt is a label` `bra{.uni} tgt; // unconditional branch` `@p bra{.uni} tgt{, tlist}; // indirect branch, tgt is reg` `bra{.uni} tgt{, tlist};`
Description	Continue execution at the target. Conditional branches are specified by using a guard predicate. The branch target must be a label. The branch target can be either a label or an address of a label held in a register. bra.uni is guaranteed to be non-divergent, meaning that all threads in a warp have identical values for the guard predicate and branch target. Indirect branches support an optional second operand, *tlist*, to communicate the list of potential targets. This operand is either the name of an array (jump table) initialized to a list of labels; or a label associated with a .branchtargets directive, which declares a list of potential target labels. The *tgt* register must hold the address of one of the control flow labels in the jump table or .branchtargets list indicated by *tlist*. If no *tlist* is provided, the branch target may be any label within the current function whose address is taken (i.e., any label used in a variable initialize or as the source operand of a mov instruction. Note that if no *tlist* is given, the optimizer will build a conservative control flow graph which may degrade performance. If *tlist* is given and the actual target is not in *tlist*, the code is incorrect and the program may generate incorrect results or fail to execute. Jump tables and .branchtargets declarations must be within the local function scope and refer only to control flow labels within the current function. Jump tables may be defined in either the constant or global state space.
Semantics	if (p) { pc = *tgt*; }
PTX ISA Notes	Direct branch introduced in PTX ISA version 1.0. Indirect branch introduced in PTX ISA version 2.1. **Note: indirect branch is currently unimplemented.**
Target ISA Notes	Direct branch supported on all target architectures. Indirect branch requires sm_20 or higher.
Examples	`bra.uni L_exit; // uniform unconditional jump` `@q bra L23; // conditional branch` `// indirect branch using jump table` `.global .u32 jmptbl[5] = { Loop, Done, L1, L2, L3 };` `…` `@p ld.global.u32 %r0, [jmptbl+4];` `@q ld.global.u32 %r0, [jmptbl+8];` `bra %r0, jmptbl;` `// indirect branch using .branchtargets directive` `…` `@p mov.u32 %r0, Done;` `@q mov.u32 %r0, L1;` `Btgt: .branchtargets Done, L1;` `bra %r0, Btgt;` `// indirect branch with no target list provided` `…` `@p mov.u32 %r0, Done;` `@q mov.u32 %r0, L1;` `bra %r0;`

Table 107. Control Flow Instructions: call

call	Call a function, recording the return location.
Syntax	`// direct call to named function, func is a symbol` `call{.uni} (ret-param), func, (param-list);` `call{.uni} func, (param-list);` `call{.uni} func;` `// indirect call via pointer, with full list of call targets` `call{.uni} (ret-param), fptr, (param-list), flist;` `call{.uni} fptr, (param-list), flist;` `call{.uni} fptr, flist;` `// indirect call via pointer, with no knowledge of call targets` `call{.uni} (ret-param), fptr, (param-list), fproto;` `call{.uni} fptr, (param-list), fproto;` `call{.uni} fptr, fproto;`
Description	The call instruction stores the address of the next instruction, so execution can resume at that point after executing a ret instruction. A call is assumed to be divergent unless the .uni suffix is present, indicating that the call is guaranteed to be non-divergent, meaning that all threads in a warp have identical values for the guard predicate and call target. For direct calls, the called location *func* must be a symbolic function name; for indirect calls, the called location *fptr* must be an address of a function held in a register. Input arguments and return values are optional. Arguments may be registers, immediate constants, or variables in .param space. Arguments are pass-by-value. Indirect calls require an additional operand, *flist* or *fproto*, to communicate the list of potential call targets or the common function prototype of all call targets, respectively. In the first case, *flist* gives a complete list of potential call targets and the optimizing backend is free to optimize the calling convention. In the second case, where the complete list of potential call targets may not be known, the common function prototype is given and the call must obey the ABI's calling convention. The *flist* operand is either the name of an array (call table) initialized to a list of function names; or a label associated with a .calltargets directive, which declares a list of potential call targets. In both cases the *fptr* register holds the address of a function listed in the call table or .calltargets list, and the call operands are type-checked against the type signature of the functions indicated by *flist*. The *fproto* operand is the name of a label associated with a .callprototype directive. This operand is used when a complete list of potential targets is not known. The call operands are type-checked against the prototype, and code generation will follow the ABI calling convention. If a function that doesn't match the prototype is called, the behavior is undefined. Call tables may be declared at module scope or local scope, in either the constant or global state space. The .calltargets and .callprototype directives must be declared within a function body. All functions must be declared prior to being referenced in a call table initializer or .calltargets directive.
PTX ISA Notes	Direct call introduced in PTX ISA version 1.0. Indirect call introduced in PTX ISA version 2.1.
Target ISA Notes	Direct call supported on all target architectures. Indirect call requires sm_20 or higher.

Examples

```
// examples of direct call
    call     init;    // call function ‘init’
    call.uni g, (a);  // call function ‘g’ with parameter ‘a’
@p  call     (d), h, (a, b);  // return value into register d

// call-via-pointer using jump table
.func (.reg .u32 rv) foo (.reg .u32 a, .reg .u32 b) …
.func (.reg .u32 rv) bar (.reg .u32 a, .reg .u32 b) …
.func (.reg .u32 rv) baz (.reg .u32 a, .reg .u32 b) …

.global .u32 jmptbl[5] = { foo, bar, baz };
      …
@p    ld.global.u32  %r0, [jmptbl+4];
@p    ld.global.u32  %r0, [jmptbl+8];
      call  (retval), %r0, (x, y), jmptbl;

// call-via-pointer using .calltargets directive
.func (.reg .u32 rv) foo (.reg .u32 a, .reg .u32 b) …
.func (.reg .u32 rv) bar (.reg .u32 a, .reg .u32 b) …
.func (.reg .u32 rv) baz (.reg .u32 a, .reg .u32 b) …
      …
@p    mov.u32  %r0, foo;
@q    mov.u32  %r0, baz;
Ftgt: .calltargets foo, bar, baz;
      call  (retval), %r0, (x, y), Ftgt;

// call-via-pointer using .callprototype directive
.func dispatch (.reg .u32 fptr, .reg .u32 idx)
{
…
Fproto: .callprototype _ (.param .u32 _, .param .u32 _);
      call  %fptr, (x, y), Fproto;
…
```

Table 108. Control Flow Instructions: ret

ret	Return from function to instruction after call.
Syntax	`ret{.uni};`
Description	Return execution to caller's environment. A divergent return suspends threads until all threads are ready to return to the caller. This allows multiple divergent ret instructions. A ret is assumed to be divergent unless the .uni suffix is present, indicating that the return is guaranteed to be non-divergent. Any values returned from a function should be moved into the return parameter variables prior to executing the ret instruction. A return instruction executed in a top-level entry routine will terminate thread execution.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`ret;` `@p ret;`

Table 109. Control Flow Instructions: exit

exit	Terminate a thread.
Syntax	`exit;`
Description	Ends execution of a thread. As threads exit, barriers waiting on all threads are checked to see if the exiting threads are the only threads that have not yet made it to a barrier for all threads in the CTA. If the exiting threads are holding up the barrier, the barrier is released.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`exit;` `@p exit;`

8.7.10 Parallel Synchronization and Communication Instructions

These instructions are:

- bar
- membar
- atom
- red
- vote

Table 110. Parallel Synchronization and Communication Instructions: bar

bar	Barrier synchronization
Syntax	`bar.sync a{, b};` `bar.arrive a, b;` `bar.red.popc.u32 d, a{, b}, {!}c;` `bar.red.`*`op`*`.pred p, a{, b}, {!}c;` `.op = { .and, .or };`
Description	Performs barrier synchronization and communication within a CTA. Each CTA instance has sixteen barriers numbered 0..15. Cooperative thread arrays use the bar instruction for barrier synchronization and communication between threads. The barrier instructions signal the arrival of the executing threads at the named barrier. In addition to signaling its arrival at the barrier, the bar.sync and bar.red instructions cause the executing thread to wait until all or a specified number of threads in the CTA arrive at the barrier before resuming execution. bar.red performs a predicate reduction across the threads participating in the barrier. bar.arrive does not cause any waiting by the executing threads; it simply marks a thread's arrival at the barrier. bar.sync and bar.red also guarantee memory ordering among threads identical to membar.cta. Thus, threads within a CTA that wish to communicate via memory can store to memory, execute a bar.sync or bar.red instruction, and then safely read values stored by other threads prior to the barrier. Operands **a**, **b**, and **d** have type .u32; operands **p** and **c** are predicates. Source operand **a** specifies a logical barrier resource as an immediate constant or register with value 0 through 15. Operand **b** specifies the number of threads participating in the barrier. If no thread count is specified, all threads in the CTA participate in the barrier. When specifying a thread count, the value must be a multiple of the warp size. When a barrier completes, the waiting threads are restarted without delay, and the barrier is reinitialized so that it can be immediately reused. Note that a non-zero thread count is required for bar.arrive. bar.red performs a reduction operation across threads. bar.red delays the executing threads (similar to bar.sync) until the barrier count is met. The **c** predicate (or its complement) from all threads in the CTA are combined using the specified reduction operator. Once the barrier count is reached, the final value is written to the destination register in all threads waiting at the barrier. The reduction operations for bar.red are population-count (.popc), all-threads-True (.and), and any-thread-True (.or). The result of .popc is the number of threads with a True predicate, while .and and .or indicate if all the threads had a True predicate or if any of the threads had a True predicate. Barriers are executed on a per-warp basis as if all the threads in a warp are active. Thus, if any thread in a warp executes a bar instruction, it is as if all the threads in the warp have executed the bar instruction. All threads in the warp are stalled until the barrier completes, and the arrival count for the barrier is incremented by the warp size (not the number of active threads in the warp). In conditionally executed code, a bar instruction should only be used if it is known that all threads evaluate the condition identically (the warp does not diverge). Since barriers are executed on a per-warp basis, the optional thread count must be a multiple of the warp size. Different warps may execute different forms of the barrier instruction using the same barrier name and thread count. One example mixes bar.sync and bar.arrive to implement producer/consumer models. The producer threads execute bar.arrive to announce their arrival at the barrier and continue execution without delay to produce the next value, while the consumer threads execute the bar.sync to wait for a resource to be produced. The roles are then reversed, using a different barrier, where the producer threads execute a bar.sync to wait for a resource to consumed, while the consumer threads announce that the resource has been consumed with bar.arrive. Care must be taken to keep a warp from executing more barrier instructions than intended (bar.arrive followed by any other bar instruction to the same barrier) prior to the reset of the barrier.

	bar.red should not be intermixed with bar.sync or bar.arrive using the same active barrier. Execution in this case is unpredictable.
PTX ISA Notes	bar.sync without a thread count introduced in PTX ISA version 1.0. Register operands, thread count, and bar.{arrive,red} introduced in PTX ISA version 2.0.
Target ISA Notes	Register operands, thread count, and bar.{arrive,red} require sm_20 or higher. Only bar.sync with an immediate barrier number is supported for sm_1x targets.
Examples	`// Use bar.sync to arrive at a pre-computed barrier number and` `// wait for all threads in CTA to also arrive:` `  st.shared [r0],r1;  // write my result to shared memory` `  bar.sync  1;  // arrive, wait for others to arrive` `  ld.shared r2,[r3];  // use shared results from other threads` `// Use bar.sync to arrive at a pre-computed barrier number and` `// wait for fixed number of cooperating threads to arrive:` `  #define CNT1 (8*12) // Number of cooperating threads` `  st.shared [r0],r1;  // write my result to shared memory` `  bar.sync  1, CNT1;  // arrive, wait for others to arrive` `  ld.shared r2,[r3];  // use shared results from other threads` `// Use bar.red.and to compare results across the entire CTA:` `  setp.eq.u32 p,r1,r2;  // p is True if r1==r2` `  bar.red.and.pred r3,1,p; // r3=AND(p) forall threads in CTA` `// Use bar.red.popc to compute the size of a group of threads` `// that have a specific condition True:` `  setp.eq.u32 p,r1,r2;  // p is True if r1==r2` `  bar.red.popc.u32 r3,1,p; // r3=SUM(p) forall threads in CTA` `/* Producer/consumer model. The producer deposits a value in` ` * shared memory, signals that it is complete but does not wait` ` * using bar.arrive, and begins fetching more data from memory.` ` * Once the data returns from memory, the producer must wait` ` * until the consumer signals that it has read the value from` ` * the shared memory location. In the meantime, a consumer` ` * thread waits until the data is stored by the producer, reads` ` * it, and then signals that it is done (without waiting).` ` */` `  // Producer code places produced value in shared memory.` `  st.shared  [r0],r1;` `  bar.arrive  0,64;` `  ld.global  r1,[r2];` `  bar.sync  1,64;` `  ...` `  // Consumer code, reads value from shared memory` `  bar.sync  0,64;` `  ld.shared  r1,[r0];` `  bar.arrive 1,64;` `  ...`

Table 111. Parallel Synchronization and Communication Instructions: membar

membar	Memory barrier.
Syntax	`membar.`*`level`*`;` `.`*`level`*` = { .cta, ,gl, ,sys };`
Description	Waits for all prior memory accesses requested by this thread to be *performed* at the CTA, global, or system memory level. *level* describes the scope of other clients for which membar is an ordering event. Thread execution resumes after a membar when the thread's prior memory writes are visible to other threads at the specified level, and memory reads by this thread can no longer be affected by other thread writes. A memory read (e.g. by ld or atom) has been performed when the value read has been transmitted from memory and cannot be modified by another thread at the indicated level. A memory write (e.g. by st, red or atom) has been performed when the value written has become visible to other clients at the specified level, that is, when the previous value can no longer be read. membar.cta — Waits until all prior memory writes are visible to other threads in the same CTA. Waits until prior memory reads have been performed with respect to other threads in the CTA. membar.gl — Waits until all prior memory requests have been performed with respect to all other threads in the GPU. For communication between threads in different CTAs or even different SMs, this is the appropriate level of membar. membar.gl will typically have a longer latency than membar.cta. membar.sys — Waits until all prior memory requests have been performed with respect to all clients, including thoses communicating via PCI-E such as system and peer-to-peer memory. This level of membar is required to insure performance with respect to a host CPU or other PCI-E peers. membar.sys will typically have much longer latency than membar.gl.
PTX ISA Notes	membar.{cta,gl} introduced in PTX ISA version 1.4. membar.sys introduced in PTX ISA version 2.0.
Target ISA Notes	membar.{cta,gl} supported on all target architectures. membar.sys requires sm_20 or higher.
Examples	`membar.gl;` `membar.cta;` `membar.sys;`

Table 112. Parallel Synchronization and Communication Instructions: atom

atom	Atomic reduction operations for thread-to-thread communication.
Syntax	`atom{.space}.op.type  d, [a], b;` `atom{.space}.op.type  d, [a], b, c;` `.space = { .global, .shared };` `.op    = { .and, .or, .xor,        // .b32, .b64` `.cas, .exch,            // .b32, .b64` `.add,                  // .u32, .s32, .f32, .u64` `.inc, .dec,              // .u32 only` `.min, .max };           // .u32, .s32, .u64, .s64` `.type  = { .b32, .b64, .u32, .u64, .s32, .s64, .f32 };`
Description	Atomically loads the original value at location **a** into destination register **d**, performs a reduction operation with operand **b** and the value in location **a**, and stores the result of the specified operation at location **a**, overwriting the original value. Operand **a** specifies a location in the specified state space. If no state space is given, perform the memory accesses using generic addressing. In generic addressing, an address maps to global memory unless it falls within a window for const, local, or shared memory. Within these windows, an address maps to the corresponding location in const, local, or shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. For atom, accesses to const and local memory are illegal. Atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer's responsibility to guarantee correctness of programs that use shared memory atomic instructions, e.g. by inserting barriers between normal stores and atomic operations to a common address, or by using atom.exch to store to locations accessed by other atomic operations. The addressable operand **a** is one of: [var] the name of an addressable variable **var**, [reg] a de-referenced register **reg** containing a byte address, [reg+immOff] a de-referenced sum of register **reg** containing a byte address plus a constant integer byte offset, or [immAddr] an immediate absolute byte address. The address must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. A register containing an address may be declared as a bit-size type or integer type. The bit-size operations are .and, .or, .xor, .cas (compare-and-swap), and .exch (exchange). The integer operations are .add, .inc, .dec, .min, .max. The .inc and .dec operations return a result in the range [0..b]. The floating-point operation .add is a single-precision, 32-bit operation. atom.add.f32 rounds to nearest even and flushes subnormal inputs and results to sign-preserving zero.
Semantics	`atomic {` `  d = *a;` `  *a = (operation == cas) ? operation(*a, b, c)` `  : operation(*a, b);` `}` `where` `  inc(r, s)  = (r >= s) ? 0 : r+1;` `  dec(r, s)  = (r==0 || r > s)  ? s : r-1;` `  exch(r, s) =  s;` `  cas(r,s,t) = (r == s) ? t : r;`

Notes	Operand **a** must reside in either the global or shared state space. Simple reductions may be specified by using the “bit bucket” destination operand ‘_’.
PTX ISA Notes	32-bit atom.global introduced in PTX ISA version 1.1. atom.shared and 64-bit atom.global.{add,cas,exch} introduced in PTX ISA 1.2. atom.add.f32 and 64-bit atom.shared.{add,cas,exch} introduced in PTX ISA 2.0. 64-bit atom.{and,or,xor,min,max} introduced in PTX ISA 3.1.
Target ISA Notes	atom.global requires sm_11 or higher. atom.shared requires sm_12 or higher. 64-bit atom.global.{add,cas,exch} require sm_12 or higher. 64-bit atom.shared.{add,cas,exch} require sm_20 or higher. 64-bit atom.{and,or.xor,min,max} require sm_35 or higher. atom.add.f32 requires sm_20 or higher. Use of generic addressing requires sm_20 or higher.
Examples	`atom.global.add.s32  d,[a],1;` `atom.shared.max.u32  d,[x+4],0;` `@p  atom.global.cas.b32  d,[p],my_val,my_new_val;`

Table 113. Parallel Synchronization and Communication Instructions: red

red	Reduction operations on global and shared memory.
Syntax	`red{.space}.op.type [a], b;` `.space = { .global, .shared };` `.op = { .and, .or, .xor, // .b32, .b64` `.add, // .u32, .s32, .f32, .u64` `.inc, .dec, // .u32 only` `.min, .max }; // .u32, .s32, .u64, .s64` `.type = { .b32, .b64, .u32, .u64, .s32, .s64, .f32 };`
Description	Performs a reduction operation with operand **b** and the value in location **a**, and stores the result of the specified operation at location **a**, overwriting the original value. Operand **a** specifies a location in the specified state space. If no state space is given, perform the memory accesses using generic addressing. In generic addressing, an address maps to global memory unless it falls within a window for const, local, or shared memory. Within these windows, an address maps to the corresponding location in const, local, shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. For red, accesses to const and local memory are illegal. Reduction operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer's responsibility to guarantee correctness of programs that use shared memory reduction instructions, e.g. by inserting barriers between normal stores and reduction operations to a common address, or by using atom.exch to store to locations accessed by other reduction operations. The addressable operand **a** is one of: [var] the name of an addressable variable **var**, [reg] a de-referenced register **reg** containing a byte address, [reg+immOff] a de-referenced sum of register **reg** containing a byte address plus a constant integer byte offset, or [immAddr] an immediate absolute byte address. The address must be naturally aligned to a multiple of the access size. If an address is not properly aligned, the resulting behavior is undefined; i.e., the access may proceed by silently masking off low-order address bits to achieve proper rounding, or the instruction may fault. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. A register containing an address may be declared as a bit-size type or integer type. The bit-size operations are .and, .or, and .xor. The integer operations are .add, .inc, .dec, .min, .max. The .inc and .dec operations return a result in the range [0..b]. The floating-point operation .add is a single-precision, 32-bit operation. red.add.f32 rounds to nearest even and flushes subnormal inputs and results to sign-preserving zero.
Semantics	`*a = operation(*a, b);` `where` `inc(r, s) = (r >= s) ? 0 : r+1;` `dec(r, s) = (r==0 || r > s) ? s : r-1;`
Notes	Operand **a** must reside in either the global or shared state space.
PTX ISA Notes	Introduced in PTX ISA version 1.2. red.add.f32 and red.shared.add.u64 introduced in PTX ISA 2.0. 64-bit red.{and,or,xor,min,max} introduced in PTX ISA 3.1.

Target ISA Notes	red.global requires sm_11 or higher red.shared requires sm_12 or higher. red.global.add.u64 requires sm_12 or higher. red.shared.add.u64 requires sm_20 or higher. 64-bit red.{and,or.xor,min,max} require sm_35 or higher. red.add.f32 requires sm_20 or higher. Use of generic addressing requires sm_20 or higher.
Examples	`red.global.add.s32 [a],1;` `red.shared.max.u32 [x+4],0;` `@p red.global.and.b32 [p],my_val;`

Table 114. Parallel Synchronization and Communication Instructions: vote

vote	Vote across thread group.
Syntax	`vote.`*`mode`*`.pred  d, {!}a;` `vote.ballot.b32 d, {!}a;  // 'ballot' form, returns bitmask` *`.mode`* `= { .all, .any, .uni };`
Description	Performs a reduction of the source predicate across threads in a warp. The destination predicate value is the same across all threads in the warp. The reduction modes are: .all True if source predicate is True for all active threads in warp. Negate the source predicate to compute .none. .any True if source predicate is True for some active thread in warp. Negate the source predicate to compute .not_all. .uni True if source predicate has the same value in all active threads in warp. Negating the source predicate also computes .uni. In the 'ballot' form, vote.ballot.b32 simply copies the predicate from each thread in a warp into the corresponding bit position of destination register d, where the bit position corresponds to the thread's lane id.
PTX ISA Notes	Introduced in PTX ISA version 1.2.
Target ISA Notes	vote requires sm_12 or higher. vote.ballot.b32 requires sm_20 or higher.
Release Notes	Note that vote applies to threads in a single warp, not across an entire CTA.
Examples	`vote.all.pred  p,q;` `vote.uni.pred  p,q;` `vote.ballot.b32  r1,p;  // get 'ballot' across warp`

8.7.11 Video Instructions

All video instructions operate on 32-bit register operands. However, the video instructions may be classified as either scalar or SIMD based on whether their core operation applies to one or multiple values.

The video instructions are:

- vadd, vadd2, vadd4
- vsub, vsub2, vsub4
- vmad
- vavrg2, vavrg4
- vabsdiff, vabsdiff2, vabsdiff4
- vmin, vmin2, vmin4
- vmax, vmax2, vmax4
- vshl
- vshr
- vset, vset2, vset4

8.7.12 Scalar Video Instructions

All scalar video instructions operate on 32-bit register operands. The scalar video instructions are:

- vadd
- vsub
- vabsdiff
- vmin
- vmax
- vshl
- vshr
- vmad
- vset

The scalar video instructions execute the following stages:

1. extract and sign- or zero-extend byte, half-word, or word values from its source operands, to produce signed 33-bit input values,
2. perform a scalar arithmetic operation to produce a signed 34-bit result,
3. optionally clamp the result to the range of the destination type,
4. optionally perform one of the following:
   a) apply a second operation to the intermediate result and a third operand, or
   b) truncate the intermediate result to a byte or half-word value and merge into a specified position in the third operand to produce the final result.

The general format of scalar video instructions is as follows:

```
// 32-bit scalar operation, with optional secondary operation
vop.dtype.atype.btype{.sat}        d, a{.asel}, b{.bsel};
vop.dtype.atype.btype{.sat}.secop  d, a{.asel}, b{.bsel}, c;

// 32-bit scalar operation, with optional data merge
vop.dtype.atype.btype{.sat}   d.dsel, a{.asel}, b{.bsel}, c;

.dtype = .atype = .btype = { .u32, .s32 };
.dsel  = .asel  = .bsel  = { .b0, .b1, .b2, .b3, .h0, .h1 };
.secop = { .add, .min, .max };
```

The source and destination operands are all 32-bit registers. The type of each operand (.u32 or .s32) is specified in the instruction type; all combinations of dtype, atype, and btype are valid. Using the atype/btype and asel/bsel specifiers, the input values are extracted and sign- or zero- extended internally to .s33 values. The primary operation is

then performed to produce an .s34 intermediate result. The sign of the intermediate result depends on dtype.

The intermediate result is optionally clamped to the range of the destination type (signed or unsigned), taking into account the subword destination size in the case of optional data merging.

```
.s33 optSaturate( .s34 tmp, Bool sat, Bool sign, Modifier dsel ) {
    if ( !sat )  return tmp;

    switch ( dsel ) {
      case .b0, .b1, .b2, .b3:
          if ( sign )  return CLAMP( tmp, S8_MAX, S8_MIN );
          else         return CLAMP( tmp, U8_MAX, U8_MIN );
      case .h0, .h1:
          if ( sign )  return CLAMP( tmp, S16_MAX, S16_MIN );
          else         return CLAMP( tmp, U16_MAX, U16_MIN );
      default:
          if ( sign )  return CLAMP( tmp, S32_MAX, S32_MIN );
          else         return CLAMP( tmp, U32_MAX, U32_MIN );
    }
}
```

This intermediate result is then optionally combined with the third source operand using a secondary arithmetic operation or subword data merge, as shown in the following pseudocode. The sign of the third operand is based on dtype.

```
.s33 optSecOp(Modifier secop, .s33 tmp, .s33 c) {
    switch ( secop ) {
        .add:     return tmp + c;
        .min:     return MIN(tmp, c);
        .max      return MAX(tmp, c);
        default:  return tmp;
    }
}
```

```
.s33 optMerge( Modifier dsel, .s33 tmp, .s33 c ) {
    switch ( dsel ) {
        case .h0:  return ((tmp & 0xffff)        | (0xffff0000 & c);
        case .h1:  return ((tmp & 0xffff) << 16) | (0x0000ffff & c);
        case .b0:  return ((tmp & 0xff)          | (0xffffff00 & c);
        case .b1:  return ((tmp & 0xff) <<  8)   | (0xffff00ff & c);
        case .b2:  return ((tmp & 0xff) << 16)   | (0xff00ffff & c);
        case .b3:  return ((tmp & 0xff) << 24)   | (0x00ffffff & c);
        default:   return tmp;
    }
}
```

The lower 32-bits are then written to the destination operand.

Table 115. Scalar Video Instructions: vadd, vsub, vabsdiff, vmin, vmax

vadd, vsub **vabsdiff** **vmin, vmax**	Integer byte/half-word/word addition / subtraction. Integer byte/half-word/word absolute value of difference. Integer byte/half-word/word minimum / maximum.
Syntax	**// 32-bit scalar operation, with optional secondary operation** *vop*.*dtype*.*atype*.*btype*{.sat} d, a{.*asel*}, b{.*bsel*}; *vop*.*dtype*.*atype*.*btype*{.sat}.*op2* d, a{.*asel*}, b{.*bsel*}, c; **// 32-bit scalar operation, with optional data merge** vop.dtype.atype.btype{.sat} d.dsel, a{.asel}, b{.bsel}, c; *vop* = { vadd, vsub, vabsdiff, vmin, vmax }; *.dtype* = *.atype* = *.btype* = { .u32, .s32 }; *.dsel* = *.asel* = *.bsel* = { .b0, .b1, .b2, .b3, .h0, .h1 }; *.op2* = { .add, .min, .max };
Description	Perform scalar arithmetic operation with optional saturate, and optional secondary arithmetic operation or subword data merge.
Semantics	// extract byte/half-word/word and sign- or zero-extend based on source operand type ta = partSelectSignExtend(a, *atype*, *asel*); tb = partSelectSignExtend(b, *btype*, *bsel*); switch (*vop*) { case vadd: tmp = ta + tb; case vsub: tmp = ta – tb; case vabsdiff: tmp = | ta – tb |; case vmin: tmp = MIN(ta, tb); case vmax: tmp = MAX(ta, tb); } // saturate, taking into account destination type and merge operations tmp = optSaturate(tmp, *sat*, isSigned(*dtype*), *dsel*); d = optSecondaryOp(*op2*, tmp, c); // optional secondary operation d = optMerge(*dsel*, tmp, c); // optional merge with c operand
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	vadd, vsub, vabsdiff, vmin, vmax require sm_20 or higher.
Examples	vadd.s32.u32.s32.sat r1, r2.b0, r3.h0; vsub.s32.s32.u32.sat r1, r2.h1, r3.h1; vabsdiff.s32.s32.s32.sat r1.h0, r2.b0, r3.b2, c; vmin.s32.s32.s32.sat.add r1, r2, r3, c;

Table 116. Scalar Video Instructions: vshl, vshr

vshl, vshr	Integer byte/half-word/word left / right shift.
Syntax	**// 32-bit scalar operation, with optional secondary operation** *vop*.*dtype*.*atype*.u32{.sat}.*mode* d, a{.*asel*}, b{.*bsel*}; *vop*.*dtype*.*atype*.u32{.sat}.*mode*.*op2* d, a{.*asel*}, b{.*bsel*}, c; **// 32-bit scalar operation, with optional data merge** *vop*.*dtype*.*atype*.u32{.sat}.mode d.*dsel*, a{.*asel*}, b{.*bsel*}, c; *vop* = { vshl, vshr }; .*dtype* = .*atype* = { .u32, .s32 }; .*mode* = { .clamp, .wrap }; .*dsel* = .*asel* = .*bsel* = { .b0, .b1, .b2, .b3, .h0, .h1 }; .*op2* = { .add, .min, .max };
Description	vshl: Shift **a** left by unsigned amount in **b** with optional saturate, and optional secondary arithmetic operation or subword data merge. Left shift fills with zero. vshr: Shift **a** right by unsigned amount in **b** with optional saturate, and optional secondary arithmetic operation or subword data merge. Signed shift fills with the sign bit, unsigned shift fills with zero.
Semantics	// extract byte/half-word/word and sign- or zero-extend based on source operand type ta = partSelectSignExtend(a, *atype*, *asel*); tb = partSelectSignExtend(b, .u32, *bsel*); if (*mode* == .clamp && tb > 32) tb = 32; if (*mode* == .wrap) tb = tb & 0x1f; switch (*vop*) { case vshl: tmp = ta << tb; case vshr: tmp = ta >> tb; } // saturate, taking into account destination type and merge operations tmp = optSaturate(tmp, *sat*, isSigned(*dtype*), *dsel*); d = optSecondaryOp(*op2*, tmp, c); // optional secondary operation d = optMerge(*dsel*, tmp, c); // optional merge with c operand
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	vshl, vshr require sm_20 or higher.
Examples	vshl.s32.u32.u32.clamp r1, r2, r3; vshr.u32.u32.u32.wrap r1, r2, r3.h1;

Table 117. Scalar Video Instructions: vmad

vmad	Integer byte/half-word/word multiply-accumulate.
Syntax	**// 32-bit scalar operation** `vmad.dtype.atype.btype{.sat}{.scale}  d, {-}a{.asel}, {-}b{.bsel}, {-}c;` `vmad.dtype.atype.btype.po{.sat}{.scale}  d, a{.asel}, b{.bsel}, c;` `.dtype = .atype = .btype = { .u32, .s32 };` `.asel = .bsel = { .b0, .b1, .b2, .b3, .h0, .h1 };` `.scale = { .shr7, .shr15 };`
Description	Calculate **(a*b) + c**, with optional operand negates, “plus one” mode, and scaling. The source operands support optional negation with some restrictions. Although PTX syntax allows separate negation of the **a** and **b** operands, internally this is represented as negation of the product **(a*b)**. That is, **(a*b)** is negated if and only if exactly one of **a** or **b** is negated. PTX allows negation of either **(a*b)** or **c**. The “plus one” mode (.po) computes **(a*b) + c + 1**, which is used in computing averages. Source operands may not be negated in .po mode. The intermediate result of **(a*b)** is unsigned if *atype* and *btype* are unsigned and the product **(a*b)** is not negated; otherwise, the intermediate result is signed. Input **c** has the same sign as the intermediate result. The final result is unsigned if the intermediate result is unsigned and **c** is not negated. Depending on the sign of the **a** and **b** operands, and the operand negates, the following combinations of operands are supported for VMAD: `(u32 * u32) + u32  // intermediate unsigned; final unsigned` `-(u32 * u32) + s32  // intermediate   signed; final   signed` `(u32 * u32) - u32  // intermediate unsigned; final   signed` `(u32 * s32) + s32  // intermediate   signed; final   signed` `-(u32 * s32) + s32  // intermediate   signed; final   signed` `(u32 * s32) - s32  // intermediate   signed; final   signed` `(s32 * u32) + s32  // intermediate   signed; final   signed` `-(s32 * u32) + s32  // intermediate   signed; final   signed` `(s32 * u32) - s32  // intermediate   signed; final   signed` `(s32 * s32) + s32  // intermediate   signed; final   signed` `-(s32 * s32) + s32  // intermediate   signed; final   signed` `(s32 * s32) - s32  // intermediate   signed; final   signed` The intermediate result is optionally scaled via right-shift; this result is sign-extended if the final result is signed, and zero-extended otherwise. The final result is optionally saturated to the appropriate 32-bit range based on the type (signed or unsigned) of the final result.

Semantics	// extract byte/half-word/word and sign- or zero-extend based on source operand type ta = partSelectSignExtend(a, *atype*, *asel*); tb = partSelectSignExtend(b, *btype*, *bsel*); signedFinal = isSigned(*atype*) || isSigned(*btype*) || (a.negate ^ b.negate) || c.negate; tmp[127:0] = ta * tb; lsb = 0; if (.po) { lsb = 1; } else if (a.negate ^ b.negate) { tmp = ~tmp; lsb = 1; } else if (c.negate) { c = ~c; lsb = 1; } c128[127:0] = (signedFinal) sext32(c) : zext (c); tmp = tmp + c128 + lsb; switch(*scale*) { case .shr7: result = (tmp >> 7) & 0xffffffffffffffff; case .shr15: result = (tmp >> 15) & 0xffffffffffffffff; } if (.sat) { if (signedFinal) result = CLAMP(result, S32_MAX, S32_MIN); else result = CLAMP(result, U32_MAX, U32_MIN); }
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	vmad requires sm_20 or higher.
Examples	`vmad.s32.s32.u32.sat r0, r1, r2, -r3;` `vmad.u32.u32.u32.shr15 r0, r1.h0, r2.h0, r3;`

Table 118. Scalar Video Instructions: vset

vset	Integer byte/half-word/word comparison.
Syntax	**// 32-bit scalar operation, with optional secondary operation** vset.*atype*.*btype*.*cmp* d, a{.*asel*}, b{.*bsel*}; vset.*atype*.*btype*.*cmp*.*op2* d, a{.*asel*}, b{.*bsel*}, c; **// 32-bit scalar operation, with optional data merge** vset.*atype*.*btype*.*cmp* d.*dsel*, a{.*asel*}, b{.*bsel*}, c; *.atype* = *.btype* = { .u32, .s32 }; *.cmp* = { .eq, .ne, .lt, .le, .gt, .ge }; *.dsel* = *.asel* = *.bsel* = { .b0, .b1, .b2, .b3, .h0, .h1 }; *.op2* = { .add, .min, .max };
Description	Compare input values using specified comparison, with optional secondary arithmetic operation or subword data merge. The intermediate result of the comparison is always unsigned, and therefore destination **d** and operand **c** are also unsigned.
Semantics	// extract byte/half-word/word and sign- or zero-extend based on source operand type ta = partSelectSignExtend(a, *atype*, *asel*); tb = partSelectSignExtend(b, *btype*, *bsel*); tmp = compare(ta, tb, *cmp*) ? 1 : 0; d = optSecondaryOp(*op2*, tmp, c); // optional secondary operation d = optMerge(*dsel*, tmp, c); // optional merge with c operand
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	vset requires sm_20 or higher.
Examples	vset.s32.u32.lt r1, r2, r3; vset.u32.u32.ne r1, r2, r3.h1;

8.7.13 SIMD Video Instructions

The SIMD video instructions operate on pairs of 16-bit values and quads of 8-bit values.

The SIMD video instructions are:

- vadd2, vadd4
- vsub2, vsub4
- vavrg2, vavrg4
- vabsdiff2, vabsdiff4
- vmin2, vmin4
- vmax2, vmax4
- vset2, vset4

PTX includes SIMD video instructions for operation on pairs of 16-bit values and quads of 8-bit values. The SIMD video instructions execute the following stages:

1. form input vectors by extracting and sign- or zero-extending byte or half-word values from the source operands, to form pairs of signed 17-bit values;
2. perform a SIMD arithmetic operation on the input pairs;
3. optionally clamp the result to the appropriate signed or unsigned range, as determinted by the destination type;
4. optionally perform one of the following:
   a) perform a second SIMD merge operation, or
   b) apply a scalar accumulate operation to reduce the intermediate SIMD results to a single scalar.

The general format of dual half-word SIMD video instructions is as follows:

```
// 2-way SIMD operation, with second SIMD merge or accumulate
vop2.dtype.atype.btype{.sat}{.add}  d{.mask}, a{.asel}, b{.bsel}, c;

.dtype = .atype = .btype = { .u32, .s32 };
.mask  = { .h0, .h1, .h10 };
.asel  = .bsel = { .hxy, where x,y are from { 0, 1, 2, 3 } };
```

The general format of quad byte SIMD video instructions is as follows:

```
// 4-way SIMD operation, with second SIMD merge or accumulate
vop4.dtype.atype.btype{.sat}{.add}  d{.mask}, a{.asel}, b{.bsel}, c;

.dtype = .atype = .btype = { .u32, .s32 };
.mask  = { .b0,
           .b1, .b10
           .b2, .b20, .b21, .b210,
           .b3, .b30, .b31, .b310, .b32, .b320, .b321, .b3210 };
.asel = .bsel = .bxyzw, where x,y,z,w are from { 0, …, 7 };
```

The source and destination operands are all 32-bit registers. The type of each operand (.u32 or .s32) is specified in the instruction type; all combinations of *dtype*, *atype*, and

btype are valid. Using the *atype/btype* and *asel/bsel* specifiers, the input values are extracted and sign- or zero- extended internally to .s33 values. The primary operation is then performed to produce an .s34 intermediate result. The sign of the intermediate result depends on *dtype*.

The intermediate result is optionally clamped to the range of the destination type (signed or unsigned), taking into account the subword destination size in the case of optional data merging.

Table 119. SIMD Video Instructions: vadd2, vsub2, vavrg2, vabsdiff2, vmin2, vmax2

vadd2, vsub2 **vavrg2,** **vabsdiff2** **vmin2, vmax2**	Integer dual half-word SIMD addition / subtraction. Integer dual half-word SIMD average. Integer dual half-word SIMD absolute value of difference. Integer dual half-word SIMD minimum / maximum.
Syntax	**// SIMD instruction with secondary SIMD merge operation** *vop2*.*dtype*.*atype*.*btype*{.sat} d{.*mask*}, a{.*asel*}, b{.*bsel*}, c; **// SIMD instruction with secondary accumulate operation** vop2.dtype.atype.btype.add d{.mask}, a{.asel}, b{.bsel}, c; *vop2* = { vadd2, vsub2, vavrg2, vabsdiff2, vmin2, vmax2 }; .*dtype* = .*atype* = .*btype* = { .u32, .s32 }; .*mask* = { .h0, .h1, .h10 }; // defaults to .h10 .*asel* = .*bsel* = { .h*xy*, where *x*,*y* are from { 0, 1, 2, 3 } }; .*asel* defaults to .h10 .*bsel* defaults to .h32
Description	Two-way SIMD parallel arithmetic operation with secondary operation. Elements of each dual half-word source to the operation are selected from any of the four half-words in the two source operands **a** and **b** using the *asel* and *bsel* modifiers. The selected half-words are then operated on in parallel. The results are optionally clamped to the appropriate range determined by the destination type (signed or unsigned). Saturation cannot be used with the secondary accumulate operation. For instructions with a secondary SIMD merge operation: For half-word positions indicated in *mask*, the selected half-word results are copied into destination **d**. For all other positions, the corresponding half-word from source operand **c** is copied to **d**. For instructions with a secondary accumulate operation: For half-word positions indicated in *mask*, the selected half-word results are added to operand **c**, producing a result in **d**.

Semantics	// extract pairs of half-words and sign- or zero-extend based on operand type Va = extractAndSignExt_2(a, b, *.asel*, *.atype*); Vb = extractAndSignExt_2(a, b, *.bsel*, *.btype*); Vc = extractAndSignExt_2(c); for (i=0; i<2; i++) { switch (*vop2*) { case vadd2: t[i] = Va[i] + Vb[i]; case vsub2: t[i] = Va[i] – Vb[i]; case vavrg2: t[i] = (Va[i] + Vb[i] + 1) / 2; case vabsdiff2: t[i] = | Va[i] – Vb[i] |; case vmin2: t[i] = MIN(Va[i], Vb[i]); case vmax2: t[i] = MAX(Va[i], Vb[i]); } if (.sat) { if (*.dtype* == .s32) t[i] = CLAMP(t[i], S16_MAX, S16_MIN); else t[i] = CLAMP(t[i], U16_MAX, U16_MIN); } } // secondary accumulate or SIMD merge mask = extractMaskBits(*.mask*); if (.add) { d = c; for (i=0; i<2; i++) { d += mask[i] ? t[i] : 0; } } else { d = 0; for (i=0; i<2; i++) { d |= mask[i] ? t[i] : Vc[i]; } }
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	vadd2, vsub2, varvg2, vabsdiff2, vmin2, vmax2 require sm_30 or higher.
Examples	`vadd2.s32.s32.u32.sat r1, r2, r3, r1;` `vsub2.s32.s32.s32.sat r1.h0, r2.h10, r3.h32, r1;` `vmin2.s32.u32.u32.add r1.h10, r2.h00, r3.h22, r1;`

Table 120. SIMD Video Instructions: vset2

vset2	Integer dual half-word SIMD comparison.
Syntax	**// SIMD instruction with secondary SIMD merge operation** vset2.*atype*.*btype*.*cmp* d{.*mask*}, a{.*asel*}, b{.*bsel*}, c; **// SIMD instruction with secondary accumulate operation** vset2.*atype*.*btype*.*cmp*.add d{.*mask*}, a{.*asel*}, b{.*bsel*}, c; *.atype* = *.btype* = { .u32, .s32 }; *.cmp* = { .eq, .ne, .lt, .le, .gt, .ge }; *.mask* = { .h0, .h1, .h10 }; // defaults to .h10 *.asel* = *.bsel* = { .h*xy*, where *x*,*y* are from { 0, 1, 2, 3 } }; *.asel* defaults to .h10 *.bsel* defaults to .h32
Description	Two-way SIMD parallel comparison with secondary operation. Elements of each dual half-word source to the operation are selected from any of the four half-words in the two source operands **a** and **b** using the *asel* and *bsel* modifiers. The selected half-words are then compared in parallel. The intermediate result of the comparison is always unsigned, and therefore the half-words of destination **d** and operand **c** are also unsigned. For instructions with a secondary SIMD merge operation: For half-word positions indicated in *mask*, the selected half-word results are copied into destination **d**. For all other positions, the corresponding half-word from source operand **c** is copied to **d**. For instructions with a secondary accumulate operation: For half-word positions indicated in *mask*, the selected half-word results are added to operand **c**, producing a result in **d**.
Semantics	// extract pairs of half-words and sign- or zero-extend based on operand type Va = extractAndSignExt_2(a, b, *.asel*, *.atype*); Vb = extractAndSignExt_2(a, b, *.bsel*, *.btype*); Vc = extractAndSignExt_2(c); for (i=0; i<2; i++) { t[i] = compare(Va[i], Vb[i], *.cmp*) ? 1 : 0; } // secondary accumulate or SIMD merge mask = extractMaskBits(*.mask*); if (.add) { d = c; for (i=0; i<2; i++) { d += mask[i] ? t[i] : 0; } } else { d = 0; for (i=0; i<2; i++) { d |= mask[i] ? t[i] : Vc[i]; } }
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	vset2 requires sm_30 or higher.
Examples	vset2.s32.u32.lt r1, r2, r3, r0; vset2.u32.u32.ne.add r1, r2, r3, r0;

Table 121. SIMD Video Instructions: vadd4, vsub4, vavrg4, vabsdiff4, vmin4, vmax4

vadd4, vsub4 **vavrg4,** **vabsdiff4** **vmin4, vmax4**	Integer quad byte SIMD addition / subtraction. Integer quad byte SIMD average. Integer quad byte SIMD absolute value of difference. Integer quad byte SIMD minimum / maximum.
Syntax	**// SIMD instruction with secondary SIMD merge operation** *vop4*.*dtype*.*atype*.*btype*{.sat} d{.*mask*}, a{.*asel*}, b{.*bsel*}, c; **// SIMD instruction with secondary accumulate operation** vop4.dtype.atype.btype.add d{.mask}, a{.asel}, b{.bsel}, c; *vop4* = { vadd4, vsub4, vavrg4, vabsdiff4, vmin4, vmax4 }; *.dtype* = *.atype* = *.btype* = { .u32, .s32 }; *.mask* = { .b0, .b1, .b10 .b2, .b20, .b21, .b210, .b3, .b30, .b31, .b310, .b32, .b320, .b321, .b3210 }; defaults to .b3210 *.asel* = *.bsel* = .b*xyzw*, where *x,y,z,w* are from { 0, …, 7 }; *.asel* defaults to .b3210 *.bsel* defaults to .b7654
Description	Four-way SIMD parallel arithmetic operation with secondary operation. Elements of each quad byte source to the operation are selected from any of the eight bytes in the two source operands **a** and **b** using the *asel* and *bsel* modifiers. The selected bytes are then operated on in parallel. The results are optionally clamped to the appropriate range determined by the destination type (signed or unsigned). Saturation cannot be used with the secondary accumulate operation. For instructions with a secondary SIMD merge operation: For byte positions indicated in *mask*, the selected byte results are copied into destination **d**. For all other positions, the corresponding byte from source operand **c** is copied to **d**. For instructions with a secondary accumulate operation: For byte positions indicated in *mask*, the selected byte results are added to operand **c**, producing a result in **d**.

Semantics	// extract quads of bytes and sign- or zero-extend based on operand type Va = extractAndSignExt_4(a, b, *.asel*, *.atype*); Vb = extractAndSignExt_4(a, b, *.bsel*, *.btype*); Vc = extractAndSignExt_4(c); for (i=0; i<4; i++) { switch (*vop4*) { case vadd4: t[i] = Va[i] + Vb[i]; case vsub4: t[i] = Va[i] – Vb[i]; case vavrg4: t[i] = (Va[i] + Vb[i] + 1) / 2; case vabsdiff4: t[i] = | Va[i] – Vb[i] |; case vmin4: t[i] = MIN(Va[i], Vb[i]); case vmax4: t[i] = MAX(Va[i], Vb[i]); } if (.sat) { if (*.dtype* == .s32) t[i] = CLAMP(t[i], S8_MAX, S8_MIN); else t[i] = CLAMP(t[i], U8_MAX, U8_MIN); } } // secondary accumulate or SIMD merge mask = extractMaskBits(*.mask*); if (.add) { d = c; for (i=0; i<4; i++) { d += mask[i] ? t[i] : 0; } } else { d = 0; for (i=0; i<4; i++) { d |= mask[i] ? t[i] : Vc[i]; } }
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	vadd4, vsub4, varvg4, vabsdiff4, vmin4, vmax4 require sm_30 or higher.
Examples	`vadd4.s32.s32.u32.sat r1, r2, r3, r1;` `vsub4.s32.s32.s32.sat r1.b0, r2.b3210, r3.b7654, r1;` `vmin4.s32.u32.u32.add r1.b00, r2.b0000, r3.b2222, r1;`

Table 122. SIMD Video Instructions: vset4

vset4	Integer quad byte SIMD comparison.
Syntax	**// SIMD instruction with secondary SIMD merge operation** vset4.*atype*.*btype*.*cmp* d{.*mask*}, a{.*asel*}, b{.*bsel*}, c; **// SIMD instruction with secondary accumulate operation** vset4.*atype*.*btype*.*cmp*.add d{.*mask*}, a{.*asel*}, b{.*bsel*}, c; *.atype* = *.btype* = { .u32, .s32 }; *.cmp* = { .eq, .ne, .lt, .le, .gt, .ge }; *.mask* = { .b0, .b1, .b10 .b2, .b20, .b21, .b210, .b3, .b30, .b31, .b310, .b32, .b320, .b321, .b3210 }; defaults to .b3210 *.asel* = *.bsel* = .b*xyzw*, where *x,y,z,w* are from { 0, …, 7 }; *.asel* defaults to .b3210 *.bsel* defaults to .b7654
Description	Four-way SIMD parallel comparison with secondary operation. Elements of each quad byte source to the operation are selected from any of the eight bytes in the two source operands **a** and **b** using the *asel* and *bsel* modifiers. The selected bytes are then compared in parallel. The intermediate result of the comparison is always unsigned, and therefore the bytes of destination **d** and operand **c** are also unsigned. For instructions with a secondary SIMD merge operation: For byte positions indicated in *mask*, the selected byte results are copied into destination **d**. For all other positions, the corresponding byte from source operand **c** is copied to **d**. For instructions with a secondary accumulate operation: For byte positions indicated in *mask*, the selected byte results are added to operand **c**, producing a result in **d**.
Semantics	// extract quads of bytes and sign- or zero-extend based on operand type Va = extractAndSignExt_4(a, b, *.asel*, *.atype*); Vb = extractAndSignExt_4(a, b, *.bsel*, *.btype*); Vc = extractAndSignExt_4(c); for (i=0; i<4; i++) { t[i] = compare(Va[i], Vb[i], *cmp*) ? 1 : 0; } // secondary accumulate or SIMD merge mask = extractMaskBits(*.mask*); if (.add) { d = c; for (i=0; i<4; i++) { d += mask[i] ? t[i] : 0; } } else { d = 0; for (i=0; i<4; i++) { d |= mask[i] ? t[i] : Vc[i]; } }
PTX ISA Notes	Introduced in PTX ISA version 3.0.
Target ISA Notes	vset4 requires sm_30 or higher.
Examples	vset4.s32.u32.lt r1, r2, r3, r0; vset4.u32.u32.ne.max r1, r2, r3, r0;

8.7.14 Miscellaneous Instructions

The Miscellaneous instructions are:

- trap
- brkpt
- pmevent

Table 123. Miscellaneous Instructions: trap

trap	Perform trap operation.
Syntax	`trap;`
Description	Abort execution and generate an interrupt to the host CPU.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`trap;` `@p trap;`

Table 124. Miscellaneous Instructions: brkpt

brkpt	Breakpoint.
Syntax	`brkpt;`
Description	Suspends execution
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	brkpt requires sm_11 or higher.
Examples	`brkpt;` `@p brkpt;`

Table 125. Miscellaneous Instructions: pmevent

pmevent	Trigger one or more Performance Monitor events.
Syntax	`pmevent a; // trigger a single performance monitor event` `pmevent.mask a; // trigger one or more performance monitor events`
Description	Triggers one or more of a fixed number of performance monitor events, with event index or mask specified by immediate operand **a**. pmevent (without modifier .mask) triggers a single performance monitor event indexed by immediate operand **a**, in the range 0..15. pmevent.mask triggers one or more of the performance monitor events. Each bit in the 16-bit immediate operand **a** controls an event. Programmatic performance moniter events may be combined with other hardware events using Boolean functions to increment one of the four performance counters. The relationship between events and counters is programmed via API calls from the host.
Notes	Currently, there are sixteen performance monitor events, numbered 0 through 15.
PTX ISA Notes	pmevent introduced in PTX ISA version 1.4. pmevent.mask introduced in PTX ISA version 3.0.
Target ISA Notes	pmevent supported on all target architectures. pmevent.mask requires sm_20 or higher.
Examples	`pmevent 1;` `@p pmevent 7;` `@q pmevent.mask 0xff;`

Chapter 9. SPECIAL REGISTERS

PTX includes a number of predefined, read-only variables, which are visible as special registers and accessed through mov or cvt instructions.

The special registers are:

- %tid
- %ntid
- %laneid
- %warpid
- %nwarpid
- %ctaid
- %nctaid
- %smid
- %nsmid
- %gridid
- %lanemask_eq, %lanemask_le, %lanemask_lt, %lanemask_ge, %lanemask_gt
- %clock, %clock64
- %pm0, …, %pm3
- %envreg0, …, %envreg31

Table 126. Special Registers: %tid

%tid	Thread identifier within a CTA.
Syntax (predefined)	`.sreg .v4 .u32 %tid;  // thread id vector` `.sreg .u32 %tid.x, %tid.y, %tid.z;  // thread id components`
Description	A predefined, read-only, per-thread special register initialized with the thread identifier within the CTA. The %tid special register contains a 1D, 2D, or 3D vector to match the CTA shape; the %tid value in unused dimensions is 0. The fourth element is unused and always returns zero. The number of threads in each dimension are specified by the predefined special register %ntid. Every thread in the CTA has a unique %tid. %tid component values range from 0 through %ntid-1 in each CTA dimension. %tid.y == %tid.z == 0 in 1D CTAs. %tid.z == 0 in 2D CTAs. It is guaranteed that: 0 <= %tid.x < %ntid.x 0 <= %tid.y < %ntid.y 0 <= %tid.z < %ntid.z
PTX ISA Notes	Introduced in PTX ISA version 1.0 with type .v4.u16. Redefined as type .v4.u32 in PTX ISA version 2.0. For compatibility with legacy PTX code, 16-bit mov and cvt instructions may be used to read the lower 16-bits of each component of %tid.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 %r1,%tid.x;  // move tid.x to %rh` `// legacy code accessing 16-bit components of %tid` `mov.u16 %rh,%tid.x;` `cvt.u32.u16 %r2,%tid.z;  // zero-extend tid.z to %r2`

Table 127. Special Registers: %ntid

%ntid	Number of thread IDs per CTA.
Syntax (predefined)	`.sreg .v4 .u32 %ntid;  // CTA shape vector` `.sreg .u32 %ntid.x, %ntid.y, %ntid.z;  // CTA dimensions`
Description	A predefined, read-only special register initialized with the number of thread ids in each CTA dimension. The %ntid special register contains a 3D CTA shape vector that holds the CTA dimensions. CTA dimensions are non-zero; the fourth element is unused and always returns zero. The total number of threads in a CTA is (%ntid.x * %ntid.y * %ntid.z). %ntid.y == %ntid.z == 1 in 1D CTAs. %ntid.z == 1 in 2D CTAs. Maximum values of %ntid.{x,y,z} are as follows:

.target architecture	%ntid.x	%ntid.y	%ntid.z
sm_1x	512	512	64
sm_20, sm_30, sm_35	1024	1024	64

PTX ISA Notes	Introduced in PTX ISA version 1.0 with type .v4.u16. Redefined as type .v4.u32 in PTX ISA version 2.0. For compatibility with legacy PTX code, 16-bit mov and cvt instructions may be used to read the lower 16-bits of each component of %ntid.
Target ISA Notes	Supported on all target architectures.
Examples	`// compute unified thread id for 2D CTA` `mov.u32 %r0,%tid.x;` `mov.u32 %h1,%tid.y;` `mov.u32 %h2,%ntid.x;`

	mad.u32 %r0,%h1,%h2,%r0; mov.u16 %rh,%ntid.x; // legacy code

Table 128. Special Registers: %laneid

%laneid	Lane Identifier.
Syntax (predefined)	`.sreg .u32 %laneid;`
Description	A predefined, read-only special register that returns the thread's lane within the warp. The lane identifier ranges from zero to WARP_SZ-1.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 %r, %laneid;`

Table 129. Special Registers: %warpid

%warpid	Warp Identifier.
Syntax (predefined)	`.sreg .u32 %warpid;`
Description	A predefined, read-only special register that returns the thread's warp identifier. The warp identifier provides a unique warp number within a CTA but not across CTAs within a grid. The warp identifier will be the same for all threads within a single warp. Note that %warpid is volatile and returns the location of a thread at the moment when read, but its value may change during execution, e.g. due to rescheduling of threads following preemption. For this reason, %ctaid and %tid should be used to compute a virtual warp index if such a value is needed in kernel code; %warpid is intended mainly to enable profiling and diagnostic code to sample and log information such as work place mapping and load distribution.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 %r, %warpid;`

Table 130. Special Registers: %nwarpid

%nwarpid	Number of warp identifiers.
Syntax (predefined)	`.sreg .u32 %nwarpid;`
Description	A predefined, read-only special register that returns the maximum number of warp identifiers.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%nwarpid requires sm_20 or higher.
Examples	`mov.u32 %r, %nwarpid;`

Table 131. Special Registers: %ctaid

%ctaid	CTA identifier within a grid.
Syntax (predefined)	`.sreg .v4 .u32 %ctaid;` `// CTA id vector` `.sreg .u32 %ctaid.x, %ctaid.y, %ctaid.z;` `// CTA id components`
Description	A predefined, read-only special register initialized with the CTA identifier within the CTA grid. The %ctaid special register contains a 1D, 2D, or 3D vector, depending on the shape and rank of the CTA grid. The fourth element is unused and always returns zero. It is guaranteed that: 0 <= %ctaid.x < %nctaid.x 0 <= %ctaid.y < %nctaid.y 0 <= %ctaid.z < %nctaid.z
PTX ISA Notes	Introduced in PTX ISA version 1.0 with type .v4.u16. Redefined as type .v4.u32 in PTX ISA version 2.0. For compatibility with legacy PTX code, 16-bit mov and cvt instructions may be used to read the lower 16-bits of each component of %ctaid.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 %r0,%ctaid.x;` `mov.u16 %rh,%ctaid.y; // legacy code`

Table 132. Special Registers: %nctaid

%nctaid	Number of CTA ids per grid.
Syntax (predefined)	`.sreg .v4 .u32 %nctaid` `// Grid shape vector` `.sreg .u32 %nctaid.x,%nctaid.y,%nctaid.z;` `// Grid dimensions`
Description	A predefined, read-only special register initialized with the number of CTAs in each grid dimension. The %nctaid special register contains a 3D grid shape vector, with each element having a value of at least 1. The fourth element is unused and always returns zero. Maximum values of %nctaid.{x,y,z} are as follows:

.target architecture	%nctaid.x	%nctaid.y	%nctaid.z
sm_1x, sm_20	65535	65535	65535
sm_30, sm_35	2^{31} -1	65535	65535

PTX ISA Notes	Introduced in PTX ISA version 1.0 with type .v4.u16. Redefined as type .v4.u32 in PTX ISA version 2.0. For compatibility with legacy PTX code, 16-bit mov and cvt instructions may be used to read the lower 16-bits of each component of %nctaid.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 %r0,%nctaid.x;` `mov.u16 %rh,%nctaid.x; // legacy code`

Table 133. Special Registers: %smid

%smid	SM identifier.
Syntax (predefined)	`.sreg .u32 %smid;`
Description	A predefined, read-only special register that returns the processor (SM) identifier on which a particular thread is executing. The SM identifier ranges from 0 to %nsmid-1. The SM identifier numbering is not guaranteed to be contiguous.
Notes	Note that %smid is volatile and returns the location of a thread at the moment when read, but its value may change during execution, e.g. due to rescheduling of threads following preemption. %smid is intended mainly to enable profiling and diagnostic code to sample and log information such as work place mapping and load distribution.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 %r, %smid;`

Table 134. Special Registers: %nsmid

%nsmid	Number of SM identifiers.
Syntax (predefined)	`.sreg .u32 %nsmid;`
Description	A predefined, read-only special register that returns the maximum number of SM identifiers. The SM identifier numbering is not guaranteed to be contiguous, so %nsmid may be larger than the physical number of SMs in the device.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%nsmid requires sm_20 or higher.
Examples	`mov.u32 %r, %nsmid;`

Table 135. Special Registers: %gridid

%gridid	Grid identifier.
Syntax (predefined)	`.sreg .u64 %gridid;`
Description	A predefined, read-only special register initialized with the per-grid temporal grid identifier. The %gridid is used by debuggers to distinguish CTAs within concurrent (small) CTA grids. During execution, repeated launches of programs may occur, where each launch starts a grid-of-CTAs. This variable provides the temporal grid launch number for this context. For sm_1x targets, %gridid is limited to the range [0..2^{16}-1]. For sm_20, %gridid is limited to the range [0..2^{32}-1]. sm_30 supports the entire 64-bit range.
PTX ISA Notes	Introduced in PTX ISA version 1.0 as type .u16. Redefined as type .u32 in PTX ISA version 1.3. Redefined as type .u64 in PTX ISA version 3.0. For compatibility with legacy PTX code, 16-bit and 32-bit mov and cvt instructions may be used to read the lower 16-bits or 32-bits of each component of %gridid.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u64 %s, %gridid; // 64-bit read of %gridid` `mov.u32 %r, %gridid; // legacy code with 32-bit %gridid`

Table 136. Special Registers: %lanemask_eq

%lanemask_eq	32-bit mask with bit set in position equal to the thread's lane number in the warp.
Syntax (predefined)	`.sreg .u32 %lanemask_eq;`
Description	A predefined, read-only special register initialized with a 32-bit mask with a bit set in the position equal to the thread's lane number in the warp.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%lanemask_eq requires sm_20 or higher.
Examples	`mov.u32 %r, %lanemask_eq;`

Table 137. Special Registers: %lanemask_le

%lanemask_le	32-bit mask with bits set in positions less than or equal to the thread's lane number in the warp.
Syntax (predefined)	`.sreg .u32 %lanemask_le;`
Description	A predefined, read-only special register initialized with a 32-bit mask with bits set in positions less than or equal to the thread's lane number in the warp.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%lanemask_le requires sm_20 or higher.
Examples	`mov.u32 %r, %lanemask_le;`

Table 138. Special Registers: %lanemask_lt

%lanemask_lt	32-bit mask with bits set in positions less than the thread's lane number in the warp.
Syntax (predefined)	`.sreg .u32 %lanemask_lt;`
Description	A predefined, read-only special register initialized with a 32-bit mask with bits set in positions less than the thread's lane number in the warp.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%lanemask_lt requires sm_20 or higher.
Examples	`mov.u32 %r, %lanemask_lt;`

Table 139. Special Registers: %lanemask_ge

%lanemask_ge	32-bit mask with bits set in positions greater than or equal to the thread's lane number in the warp.
Syntax (predefined)	`.sreg .u32 %lanemask_ge;`
Description	A predefined, read-only special register initialized with a 32-bit mask with bits set in positions greater than or equal to the thread's lane number in the warp.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%lanemask_ge requires sm_20 or higher.
Examples	`mov.u32 %r, %lanemask_ge;`

Table 140. Special Registers: %lanemask_gt

%lanemask_gt	32-bit mask with bits set in positions greater than the thread's lane number in the warp.
Syntax (predefined)	`.sreg .u32 %lanemask_gt;`
Description	A predefined, read-only special register initialized with a 32-bit mask with bits set in positions greater than the thread's lane number in the warp.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%lanemask_gt requires sm_20 or higher.
Examples	`mov.u32 %r, %lanemask_gt;`

Table 141. Special Registers: %clock

%clock	A predefined, read-only 32-bit unsigned cycle counter.
Syntax (predefined)	`.sreg .u32 %clock;`
Description	Special register %clock is an unsigned 32-bit read-only cycle counter that wraps silently.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`mov.u32 r1,%clock;`

Table 142. Special Registers: %clock64

%clock	A predefined, read-only 64-bit unsigned cycle counter.
Syntax (predefined)	`.sreg .u64 %clock64;`
Description	Special register %clock64 is an unsigned 64-bit read-only cycle counter that wraps silently.
Notes	The lower 32-bits of %clock64 are identical to %clock.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	%clock64 requires sm_20 or higher.
Examples	`mov.u64 r1,%clock64;`

Table 143. Special Registers: %pm0..%pm7

%pm0..%pm7	Performance monitoring counters.
Syntax (predefined)	`.sreg .u32 %pm<8>;`
Description	Special registers %pm0..%pm7 are unsigned 32-bit read-only performance monitor counters. Their behavior is currently undefined.
PTX ISA Notes	%pm0..%pm3 introduced in PTX ISA version 1.3. %pm4..%pm7 introduced in PTX ISA version 3.0.
Target ISA Notes	%pm0..%pm3 supported on all target architectures. %pm4..%pm7 require sm_20 or higher.
Examples	`mov.u32 r1,%pm0;` `mov.u32 r1,%pm7;`

Table 144. Special Registers: %envreg<32>

%envreg0..31	Driver-defined read-only registers
Syntax (predefined)	`.sreg .b32 %envreg<32>;`
Description	A set of 32 pre-defined read-only registers used to capture execution environment of PTX program outside of PTX virtual machine. These registers are initialized by the driver prior to kernel launch and can contain cta-wide or grid-wide values. Precise semantics of these registers is defined in the driver documentation.
PTX ISA Notes	Introduced in PTX ISA version 2.1
Target ISA Notes	Supported on all target architectures.
Examples	`mov.b32 %r1,%envreg0; // move envreg0 to %r1`

Table 145. Special Registers: %globaltimer, %globaltimer_lo, %globaltimer_hi

%globaltimer %globaltimer_lo %globaltimer_hi	A predefined, 64-bit global nanosecond timer. The lower 32-bits of %globaltimer. The upper 32-bits of %globaltimer.
Syntax (predefined)	`.sreg .u64 %globaltimer;` `.sreg .u32 %globaltimer_lo, %globaltimer_hi;`
Description	Special registers intended for use by NVIDIA tools. The behavior is target-specific and may change or be removed in future GPUs. When JIT-compiled to other targets, the value of these registers is unspecified.
PTX ISA Notes	Introduced in PTX ISA version 3.1.
Target ISA Notes	Requires target sm_30 or higher.
Examples	`mov.u64 r1,%globaltimer;`

Chapter 10. DIRECTIVES

10.1 PTX MODULE DIRECTIVES

The following directives declare the PTX ISA version of the code in the module, the target architecture for which the code was generated, and the size of addresses within the PTX module.

- .version
- .target
- .address_size

Table 146. PTX Module Directives: .version

.version	PTX ISA version number.
Syntax	`.version` *major.minor* `// major, minor are integers`
Description	Specifies the PTX language version number. The *major* number is incremented when there are incompatible changes to the PTX language, such as changes to the syntax or semantics. The version major number is used by the PTX compiler to ensure correct execution of legacy PTX code. The *minor* number is incremented when new features are added to PTX.
Semantics	Indicates that this module must be compiled with tools that support an equal or greater version number. Each PTX module must begin with a .version directive, and no other .version directive is allowed anywhere else within the module.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.version 3.1` `.version 3.0` `.version 2.3`

Table 147. PTX Module Directives: .target

<table>
<tr><td>.target</td><td>Architecture and Platform target.</td></tr>
<tr><td>Syntax</td><td>
<pre>
.target stringlist   // comma separated list of target specifiers

string = { sm_30, sm_35   // sm_3x target architectures
          sm_20,   // sm_2x target architectures
          sm_10, sm_11, sm_12, sm_13, // sm_1x target architectures
          texmode_unified, texmode_independent,   // texturing mode
          debug,   // platform option
          map_f64_to_f32 };   // platform option
</pre>
</td></tr>
<tr><td>Description</td><td>Specifies the set of features in the target architecture for which the current PTX code was generated. In general, generations of SM architectures follow an “onion layer” model, where each generation adds new features and retains all features of previous generations. Therefore, PTX code generated for a given target can be run on later generation devices.</td></tr>
<tr><td>Semantics</td><td>
Each PTX module must begin with a .version directive, immediately followed by a .target directive containing a target architecture and optional platform options. A .target directive specifies a single target architecture, but subsequent .target directives can be used to change the set of target features allowed during parsing. A program with multiple .target directives will compile and run only on devices that support all features of the highest-numbered architecture listed in the program.

PTX features are checked against the specified target architecture, and an error is generated if an unsupported feature is used. The following table summarizes the features in PTX that vary according to target architecture.
<table>
<tr><th>Target</th><th>Description</th></tr>
<tr><td>sm_30</td><td>Baseline feature set for sm_30 architecture.</td></tr>
<tr><td>sm_35</td><td>Adds 64-bit {atom,red}.{and,or,xor,min,max} instructions.
Adds shf, ld.global.nc instructions.
Adds support for CUDA Dynamic Parallelism</td></tr>
</table>
<table>
<tr><th>Target</th><th>Description</th></tr>
<tr><td>sm_20</td><td>Baseline feature set for sm_20 architecture.</td></tr>
</table>
<table>
<tr><th>Target</th><th>Description</th></tr>
<tr><td>sm_10</td><td>Baseline feature set for sm_10 architecture.
Requires map_f64_to_f32 if any .f64 instructions used.</td></tr>
<tr><td>sm_11</td><td>Adds {atom,red}.global, brkpt instructions.
Requires map_f64_to_f32 if any .f64 instructions used.</td></tr>
<tr><td>sm_12</td><td>Adds {atom,red}.shared, 64-bit {atom,red}.global, vote instructions.
Requires map_f64_to_f32 if any .f64 instructions used.</td></tr>
<tr><td>sm_13</td><td>Adds double-precision support, including expanded rounding modifiers.
Disallows use of map_f64_to_f32.</td></tr>
</table>

Texturing mode: (default is .texmode_unified)
.texmode_unified — texture and sampler information is bound together and accessed via a single .texref descriptor.
.texmode_independent — texture and sampler information is referenced with independent .texref and .samplerref descriptors.

The texturing mode is specified for an entire module and cannot be changed within the module.

The .target debug option declares that the PTX file contains DWARF debug information, and
</td></tr>
</table>

	subsequent compilation of PTX will retain information needed for source-level debugging. If the debug option is declared, an error message is generated if no DWARF information is found in the file. The debug option requires PTX ISA version 3.0 or later. **map_f64_to_f32** indicates that all double-precision instructions map to single-precision regardless of the target architecture. This enables high-level language compilers to compile programs containing type double to target device that do not support double-precision operations. Note that .f64 storage remains as 64-bits, with only half being used by instructions converted from .f64 to .f32.
Notes	Targets of the form 'compute_xx' are also accepted as synonyms for 'sm_xx' targets.
PTX ISA Notes	Introduced in PTX ISA version 1.0. Target strings sm_10 and sm_11 introduced in PTX ISA version 1.0. Target strings sm_12 and sm_13 introduced in PTX ISA version 1.2. Target string sm_20 introduced in PTX ISA version 2.0. Target string sm_30 introduced in PTX ISA version 3.0. Target string sm_35 introduced in PTX ISA version 3.1. Texturing mode introduced in PTX ISA version 1.5. Platform option **debug** introduced in PTX ISA version 3.0.
Target ISA Notes	The .target directive is supported on all target architectures.
Examples	`.target sm_10 // baseline target architecture` `.target sm_13 // supports double-precision` `.target sm_20, texmode_independent`

Table 148. PTX Module Directives: .address_size

.address_size	Address size used throughout PTX module
Syntax	`.address_size` *address-size* *address-size* = { 32, 64 };
Description	Specifies the address size assumed throughout the module by the PTX code and the binary DWARF information in PTX. Redefinition of this directive within a module is not allowed. In the presence of separate compilation all modules must specify (or default to) the same address size. The .address_size directive is optional, but it must immediately follow the .target directive if present within a module.
Semantics	If the .address_size directive is omitted, the address size defaults to 32.
PTX ISA Notes	Introduced in PTX ISA version 2.3.
Target ISA Notes	Supported on all target architectures.
Examples	`// example directives` `.address_size 32 // addresses are 32 bit` `.address_size 64 // addresses are 64 bit` `// example of directive placement within a module` `.version 2.3` `.target sm_20` `.address_size 64` … `.entry foo () {` … `}`

10.2 SPECIFYING KERNEL ENTRY POINTS AND FUNCTIONS

The following directives specify kernel entry points and functions.

- .entry
- .func

Table 149. Kernel and Function Directives: .entry

.entry	Kernel entry point and body, with optional parameters.
Syntax	`.entry` *kernel-name* `(` *param-list* `)` *kernel-body* `.entry` *kernel-name* *kernel-body*
Description	Defines a kernel entry point name, parameters, and body for the kernel function. Parameters are passed via .param space memory and are listed within an optional parenthesized parameter list. Parameters may be referenced by name within the kernel body and loaded into registers using ld.param instructions. In addition to normal parameters, opaque .texref, .samplerref, and .surfref variables may be passed as parameters. These parameters can only be referenced by name within texture and surface load, store, and query instructions and cannot be accessed via ld.param instructions. The shape and size of the CTA executing the kernel are available in special registers.
Semantics	Specify the entry point for a kernel program. At kernel launch, the kernel dimensions and properties are established and made available via special registers, e.g. %ntid, %nctaid, etc.
PTX ISA Notes	For PTX ISA version 1.4 and later, parameter variables are declared in the kernel parameter list. For PTX ISA versions 1.0 through 1.3, parameter variables are declared in the kernel body. The maximum memory size supported by PTX for normal (non-opaque type) parameters is 4352 bytes. Prior to PTX ISA version 1.5, the maximum size was 256 bytes. The CUDA and OpenCL drivers support the following limits for parameter memory:
Target ISA Notes	Supported on all target architectures.
Examples	`.entry cta_fft` `.entry filter ( .param .b32 x, .param .b32 y, .param .b32 z )` `{` `  .reg .b32 %r<99>;` `  ld.param.b32  %r1, [x];` `  ld.param.b32  %r2, [y];` `  ld.param.b32  %r3, [z];` `  …` `}`

Driver	Parameter memory size
CUDA	256 bytes for sm_1x, 4096 bytes for sm_{20,30}
OpenCL	4352 bytes for all targets

Table 150. Kernel and Function Directives: .func

.func	Function definition.
Syntax	.func *fname function-body* .func *fname* (*param-list*) *function-body* .func (*ret-param*) *fname* (*param-list*) *function-body*
Description	Defines a function, including input and return parameters and optional function body. A .func definition with no body provides a function prototype. The parameter lists define locally-scoped variables in the function body. Parameters must be base types in either the register or parameter state space. Parameters in register state space may be referenced directly within instructions in the function body. Parameters in .param space are accessed using ld.param and st.param instructions in the body. Parameter passing is call-by-value. Variadic functions are represented using ellipsis following the last fixed argument, if any. The following built-in functions are provided for accessing the list of variable arguments: %va_start %va_arg %va_arg64 %va_end See Section 7.2 for a description of variadic functions.
Semantics	The PTX syntax hides all details of the underlying calling convention and ABI. The implementation of parameter passing is left to the optimizing translator, which may use a combination of registers and stack locations to pass parameters.
Release Notes	For PTX ISA version 1.x code, parameters must be in the register state space, there is no stack, and recursion is illegal. PTX ISA versions 2.0 and later with target sm_20 or higher allow parameters in the .param state space, implements an ABI with stack, and supports recursion. PTX ISA versions 2.0 and later with target sm_20 or higher support at most one return value. Variadic functions are currently unimplemented.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.func (.reg .b32 rval) foo (.reg .b32 N, .reg .f64 dbl)` `{` `.reg .b32 localVar;` `… use N, dbl;` `other code;` `mov.b32 rval,result;` `ret;` `}` `…` `call (fooval), foo, (val0, val1);  // return value in fooval` `…`

10.3 CONTROL FLOW DIRECTIVES

PTX provides directives for specifying potential targets for indirect branch and call instructions. See the descriptions of bra and call for more information.

- .branchtargets
- .calltargets
- .callprototype

Table 151. Control Flow Directives: .branchtargets

.branchtargets	Declare a list of potential branch targets.
Syntax	`Label: .branchtargets` *list-of-labels* `;`
Description	Declares a list of potential branch targets for a subsequent indirect branch, and assocates the list with the label at the start of the line. All control flow labels in the list must occur within the same function as the declaration. The list of labels may use the compact, shorthand syntax for enumerating a range of labels having a common prefix.
PTX ISA Notes	Introduced in PTX ISA version 2.1. **Note: indirect branch is currently unimplemented.**
Target ISA Notes	Requires sm_20 or higher.
Examples	`// includes Lbl0, …, Lbl9` `Tgtlist: .branchtargets Loop, Lbl<10>, Done;` `…` `@p bra %r1, Tgtlist;` `…`

Table 152. Control Flow Directives: .calltargets

.calltargets	Declare a list of potential call targets.
Syntax	`Label: .calltargets` *list-of-functions* `;`
Description	Declares a list of potential call targets for a subsequent indirect call, and assocates the list with the label at the start of the line. All functions named in the list must be declared prior to the .calltargets directive, and all functions must have the same type signature.
PTX ISA Notes	Introduced in PTX ISA version 2.1.
Target ISA Notes	Requires sm_20 or higher.
Examples	`calltgt: .calltargets fastsin, fastcos;` `…` `@p call (%f1), %r0, (%x), calltgt;` `…`

Table 153. Control Flow Directives: .callprototype

.callprototype	Declare a prototype for use in an indirect call.
Syntax	`label: .callprototype _ ;` // no input or return parameters `label: .callprototype _ (`*param-list*`);` // input params, no return params `label: .callprototype (`*ret-param*`) _ ;` // no input params, return params `label: .callprototype (`*ret-param*`) _ (`*param-list*`);` // input, return parameters
Description	Defines a prototype with no specific function name, and associates the prototype with a label. The prototype may then be used in indirect call instructions where there is incomplete knowledge of the possible call targets. Parameters may have either base types in the register or parameter state spaces, or array types in parameter state space. The sink symbol '_' may be used to avoid dummy parameter names.
PTX ISA Notes	Introduced in PTX ISA version 2.1.
Target ISA Notes	Requires sm_20 or higher.
Examples	`Fproto1: .callprototype _ ;` `Fproto2: .callprototype _ (.param .f32 _);` `Fproto3: .callprototype (.param .u32 _) _ ;` `Fproto4: .callprototype (.param .u32 _) _ (.param .f32 _);` `…` `@p call (%val), %r0, (%f1), Fproto4;` `…` `// example of array parameter` `Fproto5: .callprototype _ (.param .b8 _[12]);`

10.4 PERFORMANCE-TUNING DIRECTIVES

To provide a mechanism for low-level performance tuning, PTX supports the following directives, which pass information to the backend optimizing compiler.

- .maxnreg
- .maxntid
- .reqntid
- .minnctapersm
- .maxnctapersm (deprecated)
- .pragma

The .maxnreg directive specifies the maximum number of registers to be allocated to a single thread; the .maxntid directive specifies the maximum number of threads in a thread block (CTA); the .reqntid directive specifies the required number of threads in a thread block (CTA); and the .minnctapersm directive specifies a minimum number of thread blocks to be scheduled on a single multiprocessor (SM). These can be used, for example, to throttle the resource requirements (e.g. registers) to increase total thread count and provide a greater opportunity to hide memory latency. The .minnctapersm directive can be used together with either the .maxntid or .reqntid directive to trade-off registers–per-thread against multiprocessor utilization without needed to directly specify a maximum number of registers. This may achieve better performance when compiling PTX for multiple devices having different numbers of registers per SM.

Currently, the .maxnreg, .maxntid, .reqntid, and .minnctapersm directives may be applied per-entry and must appear between an .entry directive and its body. The directives take precedence over any module-level constraints passed to the optimizing backend. A warning message is generated if the directives' constraints are inconsistent or cannot be met for the specified target device.

A general .pragma directive is supported for passing information to the PTX backend. The directive passes a list of strings to the backend, and the strings have no semantics within the PTX virtual machine model. The interpretation of .pragma values is determined by the backend implementation and is beyond the scope of the PTX ISA. Note that .pragma directives may appear at module (file) scope, at entry-scope, or as statements within a kernel or device function body.

Table 154. Performance-Tuning Directives: .maxnreg

.maxnreg	Maximum number of registers that can be allocated per thread.
Syntax	`.maxnreg` *n*
Description	Declare the maximum number of registers per thread in a CTA.
Semantics	The compiler guarantees that this limit will not be exceeded. The actual number of registers used may be less; for example, the backend may be able to compile to fewer registers, or the maximum number of registers may be further constrained by .maxntid and .maxctapersm.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`.entry foo .maxnreg 16 { … }  // max regs per thread = 16`

Table 155. Performance-Tuning Directives: .maxntid

.maxntid	Maximum number of threads in thread block (CTA).
Syntax	`.maxntid` *nx* `.maxntid` *nx, ny* `.maxntid` *nx, ny, nz*
Description	Declare the maximum number of threads in the thread block (CTA). This maximum is specified by giving the maximum extent of each dimention of the 1D, 2D, or 3D CTA. The maximum number of threads is the product of the maximum extent in each dimension.
Semantics	The maximum number of threads in the thread block, computed as the produce of the maximum extent specified for each dimension, is guaranteed not to be exceeded in any invocation of the kernel in which this directive appears. Exceeding the maximum number of threads results in a runtime error or kernel launch failure. Note that this directive guarantees that the *total* number of threads does not exceed the maximum, but does not guarantee that the limit in any particular dimension is not exceeded.
Notes	The .maxntid directive cannot be used in conjunction with the .reqntid directive.
PTX ISA Notes	Introduced in PTX ISA version 1.3.
Target ISA Notes	Supported on all target architectures.
Examples	`.entry foo .maxntid 256 { … } // max threads = 256` `.entry bar .maxntid 16,16,4 { … } // max threads = 1024`

Table 156. Performance-Tuning Directives: .reqntid

.reqntid	Number of threads in thread block (CTA).
Syntax	`.reqntid` *nx* `.reqntid` *nx, ny* `.reqntid` *nx, ny, nz*
Description	Declare the number of threads in the thread block (CTA) by specifying the extent of each dimension of the 1D, 2D, or 3D CTA. The total number of threads is the product of the number of threads in each dimension.
Semantics	The size of each CTA dimension specified in any invocation of the kernel is required to be equal to that specified in this directive. Specifying a different CTA dimension at launch will result in a runtime error or kernel launch failure.
Notes	The .reqntid directive cannot be used in conjunction with the .maxntid directive.
PTX ISA Notes	Introduced in PTX ISA version 2.1.
Target ISA Notes	Supported on all target architectures.
Examples	`.entry foo .reqntid 256 { … } // num threads = 256` `.entry bar .reqntid 16,16,4 { … } // num threads = 1024`

Table 157. Performance-Tuning Directives: .minnctapersm

.minnctapersm	Minimum number of CTAs per SM.
Syntax	`.minnctapersm` *ncta*
Description	Declare the minimum number of CTAs from the kernel's grid to be mapped to a single multiprocessor (SM).
Notes	Optimizations based on .minnctapersm need either .maxntid or .reqntid to be specified as well. In PTX ISA version 2.1 or higher, a warning is generated if .minnctapersm is specified without specifying either .maxntid or .reqntid.
PTX ISA Notes	Introduced in PTX ISA version 2.0 as a replacement for .maxnctapersm.
Target ISA Notes	Supported on all target architectures.
Examples	`.entry foo .maxntid 256 .minnctapersm 4 { … }`

Table 158. Performance-Tuning Directives: .maxnctapersm (deprecated)

.maxnctapersm	Maximum number of CTAs per SM.
Syntax	`.maxnctapersm` *ncta*
Description	Declare the maximum number of CTAs from the kernel's grid that may be mapped to a single multiprocessor (SM).
Notes	Optimizations based on .maxnctapersm generally need .maxntid to be specified as well. The optimizing backend compiler uses .maxntid and .maxnctapersm to compute an upper-bound on per-thread register usage so that the specified number of CTAs can be mapped to a single multiprocessor. However, if the number of registers used by the backend is sufficiently lower than this bound, additional CTAs may be mapped to a single multiprocessor. For this reason, .maxnctapersm has been renamed to .minnctapersm in PTX ISA version 2.0.
PTX ISA Notes	Introduced in PTX ISA version 1.3. Deprecated in PTX ISA version 2.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.entry foo .maxntid 256 .maxnctapersm 4 { … }`

Table 159. Performance-Tuning Directives: .pragma

.pragma	Pass directives to PTX backend compiler.
Syntax	`.pragma` *`list-of-strings`* `;`
Description	Pass module-scoped, entry-scoped, or statement-level directives to the PTX backend compiler. The .pragma directive may occur at module-scope, at entry-scope, or at statement-level.
Semantics	The interpretation of .pragma directive strings is implementation-specific and has no impact on PTX semantics. See Appendix A. Descriptions of .pragma Strings for descriptions of the pragma strings defined in ptxas.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.pragma “nounroll”;  // disable unrolling in backend` `// disable unrolling for current kernel` `.entry foo .pragma “nounroll”; { … }`

10.5 DEBUGGING DIRECTIVES

DWARF-format debug information is passed through PTX modules using the following directives:

- @@DWARF
- .section
- .file
- .loc

The .section directive was introduced in PTX ISA version 2.0 and replaces the @@DWARF syntax. The @@DWARF syntax was deprecated in PTX ISA version 2.0 but is supported for legacy PTX ISA version 1.x code.

Beginning with PTX ISA version 3.0, PTX files containing DWARF debug information should include the **.target debug** platform option. This forward declaration directs PTX compilation to retain mappings for source-level debugging.

Table 160. Debugging Directives: @@DWARF

@@DWARF	DWARF-format information.
Syntax	`@@DWARF` *dwarf-string* *dwarf-string* may have one of the `.byte` *byte-list* // comma-separated hexadecimal byte values `.4byte` *int32-list* // comma-separated hexadecimal integers in range [0..2^{32}-1] `.quad` *int64-list* // comma-separated hexadecimal integers in range [0..2^{64}-1] `.4byte` *label* `.quad` *label*
PTX ISA Notes	Introduced in PTX ISA version 1.2. Deprecated as of PTX ISA version 2.0, replaced by .section directive.
Target ISA Notes	Supported on all target architectures.
Examples	`@@DWARF .section .debug_pubnames, “”, @progbits` `@@DWARF .byte 0x2b, 0x00, 0x00, 0x00, 0x02, 0x00` `@@DWARF .4byte .debug_info` `@@DWARF .4byte 0x000006b5, 0x00000364, 0x61395a5f, 0x5f736f63` `@@DWARF .4byte 0x6e69616d, 0x63613031, 0x6150736f, 0x736d6172` `@@DWARF .byte 0x00, 0x00, 0x00, 0x00, 0x00`

Table 161. Debugging Directives: .section

.section	PTX section definition.
Syntax	`.section` *section_name* { *dwarf-lines* } *dwarf-lines* have the following formats: `.b8` *byte-list* // comma-separated list of integers in range [0..255] `.b32` *int32-list* // comma-separated list of integers in range [0..2^{32}-1] `.b64` *int64-list* // comma-separated list of integers in range [0..2^{64}-1] `.b32` *label* `.b64` *label*
PTX ISA Notes	Introduced in PTX ISA version 2.0, replaces @@DWARF syntax.
Target ISA Notes	Supported on all target architectures.
Examples	`.section .debug_pubnames` `{` `.b8 0x2b, 0x00, 0x00, 0x00, 0x02, 0x00` `.b32 .debug_info` `.b32 0x000006b5, 0x00000364, 0x61395a5f, 0x5f736f63` `.b32 0x6e69616d, 0x63613031, 0x6150736f, 0x736d6172` `.b8 0x00, 0x00, 0x00, 0x00, 0x00` `}`

Table 162. Debugging Directives: .file

.file	Source file name.
Syntax	`.file` *file_index "filename"*
Description	Associates a source filename with an integer index. Subsequent .loc directives reference source files by index. The .file directive is allowed only in the outermost scope, i.e., at the same level as kernel and device function declarations.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.file 1 "example.cu"` `.file 2 "kernel.cu"`

Table 163. Debugging Directives: .loc

.loc	Source file location.
Syntax	`.loc` *file_index line_number column_position*
Description	Declares the source file location (source file, line number, and column position) to be associated with lexically subsequent PTX instructions. Note that a PTX instruction may have a single associated source location, determined by the nearest lexically preceding .loc directive, or no associated source location if there is no preceding .loc directive. Labels in PTX inherit the location of the closest lexically following instruction. A label with no following PTX instruction has no associated source location.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.loc 2 4237 0` `L1: // line 4237, col 0 of file #2, inherited from mov` `mov.u32 %r1,%r2; // line 4237, col 0 of file #2` `add.u32 %r2,%r1,%r3; // line 4237, col 0 of file #2` `…` `L2: // line 4239, col 5 of file #2, inherited from sub` `.loc 2 4239 5` `sub.u32 %r2,%r1,%r3; // line 4239, col 5 of file #2`

10.6 LINKING DIRECTIVES

- .extern
- .visible
- .weak

Table 164. Linking Directives: .extern

.extern	External symbol declaration.
Syntax	`.extern` *identifier*
Description	Declares identifier to be defined external to the current module. The identifier must be declared .visible in the module where it is defined.
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.extern .global .b32 foo;  // foo is defined in another module`

Table 165. Linking Directives: .visible

.visible	Visible (externally) symbol declaration.
Syntax	`.visible` *identifier*
Description	Declares identifier to be globally visible. Unlike C, where identifiers are globally visible unless declared static, PTX identifiers are visible only within the current module unless declared .visible outside the current
PTX ISA Notes	Introduced in PTX ISA version 1.0.
Target ISA Notes	Supported on all target architectures.
Examples	`.visible .global .b32 foo;  // foo will be externally visible`

Table 166. Linking Directives: .weak

.weak	Visible (externally) symbol declaration.
Syntax	`.weak` *identifier*
Description	Declares identifier to be globally visible but “weak”. Weak symbols are similar to globally vsible symbols, except during linking, weak symbols are only chosen after global and local symbols during symbol resolution. Unlike globally visible symbols, multiple object files may declare the same weak symbol, and references to a symbol get resolved against a weak symbol only if no global or local symbols have the same name.
PTX ISA Notes	Introduced in PTX ISA version 3.1.
Target ISA Notes	Supported on all target architectures.
Examples	`.weak .func (.reg .b32 val) foo;  // foo will be externally visible`

Chapter 11. RELEASE NOTES

This section describes the history of change in the PTX ISA and implementation. The first section describes ISA and implementation changes in the current release of PTX ISA version 3.1, and the remaining sections provide a record of changes in previous releases of PTX ISA versions back to PTX ISA version 2.0.

Table 167 shows the PTX release history.

Table 167. PTX Release History

PTX ISA Version	CUDA Release	Supported Targets
PTX ISA 1.0	CUDA 1.0	sm_{10,11}
PTX ISA 1.1	CUDA 1.1	sm_{10,11}
PTX ISA 1.2	CUDA 2.0	sm_{10,11,12,13}
PTX ISA 1.3	CUDA 2.1	sm_{10,11,12,13}
PTX ISA 1.4	CUDA 2.2	sm_{10,11,12,13}
PTX ISA 1.5	driver r190	sm_{10,11,12,13}
PTX ISA 2.0	CUDA 3.0, driver r195	sm_{10,11,12,13}, sm_20
PTX ISA 2.1	CUDA 3.1, driver r256	sm_{10,11,12,13}, sm_20
PTX ISA 2.2	CUDA 3.2, driver r260	sm_{10,11,12,13}, sm_20
PTX ISA 2.3	CUDA 4.0, driver r270	sm_{10,11,12,13}, sm_20
PTX ISA 3.0	CUDA 4.1, driver r285 CUDA 4.2, driver r295	sm_{10,11,12,13}, sm_20 sm_{10,11,12,13}, sm_20, sm_30
PTX ISA 3.1	CUDA 5.0, driver r302	sm_{10,11,12,13},sm_20,sm_{30,35}

11.1 CHANGES IN PTX ISA VERSION 3.1

11.1.1 New Features

PTX ISA version 3.1 introduces the following new features:

- Support for sm_35 target architecture.
- Support for CUDA Dynamic Parallelism, which enables a kernel to create and synchronize new work.
- ld.global.nc for loading read-only global data though the non-coherent texture cache.
- A new funnel shift instruction, shf.
- Extends atomic and reduction instructions to perform 64-bit {and, or, xor} operations, and 64-bit integer {min, max} operations.
- Adds support for mipmaps.
- Adds support for indirect access to textures and surfaces.
- Extends support for generic addressing to include the .const state space, and adds a new operator, generic(), to form a generic address for .global or .const variables used in initializers.
- A new .weak directive to permit linking multiple object files containing declarations of the same symbol.

11.1.2 Semantic Changes and Clarifications

PTX 3.1 redefines the default addressing for global variables in initializers, from generic addresses to offsets in the global state space. Legacy PTX code is treated as having an implicit generic() operator for each global variable used in an initializer. PTX 3.1 code should either include explicit generic() operators in initializers, use cvta.global to form generic addresses at runtime, or load from the non-generic address using ld.global.

Instruction mad.f32 requires a rounding modifier for sm_20 and higher targets. However for PTX ISA version 3.0 and earlier, ptxas does not enforce this requirement and mad.f32 silently defaults to mad.rn.f32. For PTX ISA version 3.1, ptxas generates a warning and defaults to mad.rn.f32, and in subsequent releases ptxas will enforce the requirement for PTX ISA version 3.2 and later.

11.1.3 Features Unimplemented in PTX ISA Version 3.1

The following features remain unimplemented in PTX ISA version 3.1:

- Pointers to opaque-type variables.
- Support for variadic functions.
- Allocation of per-thread, stack-based memory using alloca.

- Indirect branches.

11.2 CHANGES IN PTX ISA VERSION 3.0

11.2.1 New Features

PTX ISA version 3.0 introduces the following new features:

- Support for sm_30 target architectures.
- SIMD video instructions.
- A new warp shuffle instruction.
- Instructions mad.cc and madc for efficient, extended-precision integer multiplication.
- Surface instructions with 3D and array geometries.
- The texture instruction supports reads from cubemap and cubemap array textures.
- Platform option .target debug to declare that a PTX module contains DWARF debug information.
- pmevent.mask, for triggering multiple performance monitor events.
- Performance monitor counter special registers %pm4..%pm7.

11.2.2 Semantic Changes and Clarifications

Special register %gridid has been extended from 32-bits to 64-bits.

PTX ISA version 3.0 deprecates module-scoped .reg and .local variables when compiling to the Application Binary Interface (ABI). When compiling without use of the ABI, module-scoped .reg and .local variables are supported as before. When compiling legacy PTX code (ISA versions prior to 3.0) containing module-scoped .reg or .local variables, the compiler silently disables use of the ABI.

The shfl instruction semantics were updated to clearly indicate that source operand **a** is read as zero for inactive and predicated-off threads within the warp.

PTX modules no longer allow duplicate .version directives. This feature was unimplemented, so there is no semantic change.

Unimplemented instructions suld.p and sust.p.{u32,s32,f32} have been removed.

11.3 CHANGES IN PTX ISA VERSION 2.3

11.3.1 New Features

PTX 2.3 adds support for texture arrays. The texture array feature supports access to an array of 1D or 2D textures, where an integer indexes into the array of textures, and then one or two single-precision floating point coordinates are used to address within the selected 1D or 2D texture.

PTX 2.3 adds a new directive, .address_size, for specifying the size of addresses.

Variables in .const and .global state spaces are initialized to zero by default.

11.3.2 Semantic Changes and Clarifications

The semantics of the .maxntid directive have been updated to match the current implementation. Specifically, .maxntid only guarantees that the total number of threads in a thread block does not exceed the maximum. Previously, the semantics indicated that the maximum was enforced separately in each dimension, which is not the case.

Bit field extract and insert instructions BFE and BFI now indicate that the **len** and **pos** operands are restricted to the value range 0..255.

Unimplemented instructions {atom,red}.f32.{min,max} have been removed.

11.4 CHANGES IN PTX ISA VERSION 2.2

11.4.1 New Features

PTX 2.2 adds a new directive for specifying kernel parameter attributes; specifically, there is a new directives for specifying that a kernel parameter is a pointer, for specifying to which state space the parameter points, and for optionally specifying the alignment of the memory to which the parameter points.

PTX 2.2 adds a new field named force_unnormalized_coords to the .samplerref opaque type. This field is used in the independent texturing mode to override the normalized_coords field in the texture header. This field is needed to support languages such as OpenCL, which represent the property of normalized/unnormalized coordinates in the sampler header rather than in the texture header.

PTX 2.2 deprecates explicit constant banks and supports a large, flat address space for the .const state space. Legacy PTX that uses explicit constant banks is still supported.

PTX 2.2 adds a new tld4 instruction for loading a component (r, g, b, or a) from the four texels compising the bilinear interpolation footprint of a given texture location. This instruction may be used to compute higher-precision bilerp results in software, or for performing higher-bandwidth texture loads.

11.4.2 Semantic Changes and Clarifications

None.

11.5 CHANGES IN PTX ISA VERSION 2.1

11.5.1 New Features

The underlying, stack-based ABI is supported in PTX ISA version 2.1 for sm_2x targets.

Support for indirect calls has been implemented for sm_2x targets.

New directives, .branchtargets and .calltargets, have been added for specifying potential targets for indirect branches and indirect function calls. A .callprototype directive has been added for declaring the type signatures for indirect function calls.

The names of .global and .const variables can now be specified in variable initializers to represent their addresses.

A set of thirty-two driver-specific execution environment special registers has been added. These are named %envreg0..%envreg31.

Textures and surfaces have new fields for channel data type and channel order, and the txq and suq instructions support queries for these fields.

Directive .minnctapersm has replaced the .maxnctapersm directive.

Directive .reqntid has been added to allow specification of exact CTA dimensions.

A new instruction, rcp.approx.ftz.f64, has been added to compute a fast, gross approximate reciprocal.

11.5.2 Semantic Changes and Clarifications

A warning is emitted if .minnctapersm is specified without also specifying .maxntid.

11.6 CHANGES IN PTX ISA VERSION 2.0

11.6.1 New Features

11.6.1.1 Floating-Point Extensions

This section describes the floating-point changes in PTX ISA version 2.0 for sm_20 targets. The goal is to achieve IEEE 754 compliance wherever possible, while maximizing backward compatibility with legacy PTX ISA version 1.x code and sm_1x targets.

The changes from PTX ISA version 1.x are as follows:

- Single-precision instructions support subnormal numbers by default for sm_20 targets. The .ftz modifier may be used to enforce backward compatibility with sm_1x.
- Single-precision add, sub, and mul now support .rm and .rp rounding modifiers for sm_20 targets.
- A single-precision fused multiply-add (fma) instruction has been added, with support for IEEE 754 compliant rounding modifiers and support for subnormal numbers. The fma.f32 instruction also supports .ftz and .sat modifiers. fma.f32 requires sm_20. The mad.f32 instruction has been extended with rounding modifiers so that it's synonymous with fma.f32 for sm_20 targets. Both fma.f32 and mad.f32 require a rounding modifier for sm_20 targets.
- The mad.f32 instruction *without rounding* is retained so that compilers can generate code for sm_1x targets. When code compiled for sm_1x is executed on sm_20 devices, mad.f32 maps to fma.rn.f32.
- Single- and double-precision div, rcp, and sqrt with IEEE 754 compliant rounding have been added. These are indicated by the use of a rounding modifier and require sm_20.
- Instructions testp and copysign have been added.

11.6.1.2 New instructions

A “load uniform” instruction, ldu, has been added.

Surface instructions support additional .clamp modifiers, .clamp and .zero.

Instruction sust now supports formatted surface stores.

A “count leading zeros” instruction, clz, has been added.

A “find leading non-sign bit” instruction, bfind, has been added.

A “bit reversal” instruction, brev, has been added.

Bit field extract and insert instructions, bfe and bfi, have been added.

A “population count” instruction, popc, has been added.

A “vote ballot” instruction, vote.ballot.b32, has been added.

Instructions {atom,red}.add.f32 have been implemented.

Instructions {atom,red}.shared have been extended to handle 64-bit data types for sm_20 targets.

A system-level membar instruction, membar.sys, has been added.

The bar instruction has been extended as follows:

- A bar.arrive instruction has been added.
- Instructions bar.red.popc.u32 and bar.red.{and,or}.pred have been added.
- bar now supports optional thread count and register operands.

Scalar video instructions (includes prmt) have been added.

Instruction isspacep for querying whether a generic address falls within a specified state space window has been added.

Instruction cvta for converting global, local, and shared addresses to generic address and vice-versa has been added.

11.6.1.3 Other new features

Instructions ld, ldu, st, prefetch, prefetchu, isspacep, cvta, atom, and red now support generic addressing.

New special registers %nwarpid, %nsmid, %clock64, %lanemask_{eq,le,lt,ge,gt} have been added.

Cache operations have been added to instructions ld, st, suld, and sust, e.g. for prefetching to specified level of memory hierarchy. Instructions prefetch and prefetchu have also been added.

The .maxnctapersm directive was deprecated and replaced with .minnctapersm to better match its behavior and usage.

A new directive, .section, has been added to replace the @@DWARF syntax for passing DWARF-format debugging information through PTX.

A new directive, .pragma "nounroll", has been added to allow users to disable loop unrolling.

11.6.2 Semantic Changes and Clarifications

The errata in cvt.ftz for PTX ISA versions 1.4 and earlier, where single-precision subnormal inputs and results were not flushed to zero if either source or destination type size was 64-bits, has been fixed. In PTX ISA version 1.5 and later, cvt.ftz (and cvt for .target sm_1x, where .ftz is implied) instructions flush single-precision subnormal inputs and results to sign-preserving zero for all combinations of floating-point instruction types. To maintain compatibility with legacy PTX code, if .version is 1.4 or earlier, single-precision subnormal inputs and results are flushed to sign-preserving zero only when neither source nor destination type size is 64-bits.

Components of special registers %tid, %ntid, %ctaid, and %nctaid have been extended from 16-bits to 32-bits. These registers now have type .v4.u32.

The number of samplers available in independent texturing mode was incorrectly listed as thirty-two in PTX ISA version 1.5; the correct number is sixteen.

APPENDIX A. DESCRIPTIONS OF .PRAGMA STRINGS

This section describes the .pragma strings defined by ptxas.

Table 168. Pragma Strings: “nounroll”

“nounroll”	Disable loop unrolling in optimizing backend compiler.
Syntax	**.pragma** “nounroll”;
Description	The “nounroll” pragma is a directive to disable loop unrolling in the optimizing backend compiler. The “nounroll” pragma is allowed at module, entry-function, and statement levels, with the following meanings: module scope — disables unrolling for all loops in module, including loops preceding the .pragma. entry-function scope — disables unrolling for all loops in the entry function body. statement-level pragma — disables unrolling of the loop for which the current block is the loop header. Note that in order to have the desired effect at statement level, the “nounroll” directive must appear before any instruction statements in the loop header basic block for the desired loop. The loop header block is defined as the block that dominates all blocks in the loop body and is the target of the loop backedge. Statement-level “nounroll” directives appearing outside of loop header blocks are silently ignored.
PTX ISA Notes	Introduced in PTX ISA version 2.0.
Target ISA Notes	Requires sm_20 or higher. Ignored for sm_1x targets.
Examples	`.entry foo (…)` `.pragma “nounroll”;  // do not unroll any loop in this function` `{` `…` `}` `.func bar (…)` `{` `…` `L1_head:` `  .pragma “nounroll”;  // do not unroll this loop` `  …` `@p  bra L1_end;` `L1_body:` `  …` `L1_continue:` `  bra L1_head;` `L1_end:` `  …` `}`

Notice

OpenCL

Trademarks

Copyright