



NVIDIA Compute

PTX: Parallel Thread Execution

ISA Version 1.1

10/24/2007
SP-03483-001_v1.1

Document Change History

Version	Date	Responsible	Reason for Change
_v1.0	June 15, 2007	RJ	Preliminary release
_v1.1	October 24, 2007	RJ, TS	Release

Table of Contents

Chapter 1. Introduction.....	1
1.1. Data-Parallel Computing Using GPUs.....	1
1.2. Goals of PTX	1
1.3. The Document's Structure	2
Chapter 2. Programming Model	3
2.1. A Highly Multithreaded Coprocessor	3
2.2. Thread Batching	3
2.2.1. Cooperative Thread Arrays	3
2.2.2. Grid of Cooperative Thread Arrays.....	4
Chapter 3. Parallel Thread Execution Machine Model.....	7
3.1. A Set of SIMD Multiprocessors with On-Chip Shared Memory	7
3.2. Execution Model	9
Chapter 4. Syntax	11
4.1. Source Format	11
4.2. Comments	11
4.3. Statements.....	12
4.3.1. Directive Statements.....	12
4.3.2. Instruction Statements	12
4.4. Identifiers	13
4.5. Immediate Constants.....	14
4.5.1. Integer Immediate Constants	14
4.5.2. Floating-point Immediate Constants.....	14
4.5.3. Predicate Immediate Constants	14
4.5.4. Constant Expressions.....	14
Chapter 5. State Spaces, Types, and Variables	15
5.1. State Spaces	15
5.1.1. Register State Space.....	16
5.1.2. Special Register Space	16
5.1.3. Constant State Space.....	16
5.1.4. Global State Space.....	17
5.1.5. Local State Space.....	17

5.1.6.	Parameter State Space	17
5.1.7.	Shared State Space	17
5.1.8.	Texture State Space	18
5.1.9.	Surface State Space	18
5.2.	Types	19
5.2.1.	Fundamental Types	19
5.2.2.	Restricted Use of Sub-word Sizes	19
5.3.	Variables	20
5.3.1.	Variable Declarations	20
5.3.2.	Vectors	20
5.3.3.	Array Declarations	21
5.3.4.	Structures and Unions	21
5.3.5.	Initializers	22
5.3.6.	Alignment	22
Chapter 6.	Instruction Operands	23
6.1.	Operand Type Information	23
6.2.	Source Operands	23
6.3.	Destination Operands	23
6.4.	Using Addresses, Arrays, Vectors, Structures, and Unions	24
6.4.1.	Addresses as Operands	24
6.4.2.	Arrays as Operands	25
6.4.3.	Vectors as Operands	25
6.4.4.	Structures and Unions as Operands	26
6.4.5.	Immediate Values as Operands	26
6.5.	Type Conversion	26
6.5.1.	Scalar Conversions	26
6.5.2.	Rounding Modes	28
6.5.3.	Vector Conversions	29
6.6.	Operand Costs	30
Chapter 7.	Instruction Set	31
7.1.	Format and Semantics of Instruction Descriptions	31
7.2.	PTX Instructions	31
7.3.	Predicated Execution	32
7.3.1.	Comparisons	33
7.3.1.1.	Integer and Bit-Size Comparisons	33

7.3.1.2.	Floating-point Comparisons	33
7.3.2.	Manipulating Predicates	34
7.4.	Type Information for Instructions and Operands	35
7.5.	Divergence of Threads in Control Constructs	35
7.6.	Semantics	36
7.6.1.	Machine-specific Semantics of 16-bit Code	36
7.7.	Instructions	37
7.7.1.	Arithmetic Instructions	37
7.7.2.	Comparison and Selection Instructions	45
7.7.3.	Logic and Shift Instructions	49
7.7.4.	Data Movement and Conversion Instructions	52
7.7.5.	Texture Instruction	56
7.7.6.	Control Flow Instructions	57
7.7.7.	Parallel Synchronization and Communication Instructions	60
7.7.8.	Floating-point Instructions	62
7.7.9.	Miscellaneous Instructions	65
Chapter 8.	Special Registers	67
Chapter 9.	Directives	71
9.1.	Specifying CTAs and Functions	71
9.2.	Debugging Directives	73
9.3.	Other Directives	74
Chapter 10.	Release Notes	77
10.1.	New Features	77
10.2.	Unimplemented Features Removed From ISA	77
10.3.	Changes to Rounding Modifiers	77
10.4.	Changes to Saturation	78
10.5.	Unimplemented Features in Version 1.1	78
10.5.1.	Syntax restrictions	78
10.5.2.	State Spaces	78
10.5.3.	Variables and Operands	78
10.5.4.	Instructions	79
10.6.	Summary of Instruction Changes	79

List of Figures

Figure 1.	Thread Batching	5
Figure 2.	Machine Model A set of SIMD multiprocessors with on-chip shared memory	8
Figure 3.	Memory Model	10

List of Tables

Table 1.	PTX Directives	12
Table 2.	Reserved Instruction Keywords	13
Table 3.	Predefined identifiers	13
Table 4.	State Spaces	15
Table 5.	Properties of State Spaces	16
Table 6.	Fundamental Specifiers	19
Table 7.	CVT Instruction Precision and Format	27
Table 8.	Floating-point Rounding Modes	28
Table 9.	Integer Rounding Modes	28
Table 10.	Conversions Between Scalar Values and Vector Values	29
Table 11.	Cost estimates for accessing state-spaces	30
Table 12.	Operators for Signed Integer, Unsigned Integer, and Bit-size Types	33
Table 13.	Floating-point Comparison Operators	33
Table 14.	Floating-Point Comparison Operators Accepting NaN	34
Table 15.	Floating-Point Comparison Operators Testing for NaN	34
Table 16.	Arithmetic Instructions: ADD	38
Table 17.	Arithmetic Instructions: SUB	38
Table 18.	Arithmetic Instructions: MUL	39
Table 19.	Arithmetic Instructions: MAD	40
Table 20.	Arithmetic Instructions: MUL24	41
Table 21.	Arithmetic Instructions: MAD24	41
Table 22.	Arithmetic Instructions: SAD	42
Table 23.	Arithmetic Instructions: DIV	42
Table 24.	Arithmetic Instructions: REM	43
Table 25.	Arithmetic Instructions: ABS	43
Table 26.	Arithmetic Instructions: NEG	43
Table 27.	Arithmetic Instructions: MIN	44
Table 28.	Arithmetic Instructions: MAX	44
Table 29.	Comparison and Selection Instructions: SET	46
Table 30.	Comparison and Selection Instructions: SETP	47
Table 31.	Comparison and Selection Instructions: SELP	48

Table 32.	Comparison and Selection Instructions: SLCT	48
Table 33.	Logic and Shift Instructions: AND	49
Table 34.	Logic and Shift Instructions: OR	49
Table 35.	Logic and Shift Instructions: XOR	50
Table 36.	Logic and Shift Instructions: NOT	50
Table 37.	Logic and Shift Instructions: CNOT	50
Table 38.	Logic and Shift Instructions: SHL	51
Table 39.	Logic and Shift Instructions: SHR	51
Table 40.	Data Movement and Conversion Instructions: MOV	52
Table 41.	Data Movement and Conversion Instructions: LD	53
Table 42.	Data Movement and Conversion Instructions: ST	54
Table 43.	Data Movement and Conversion Instructions: CVT	55
Table 44.	Texture Instruction: TEX	56
Table 45.	Control Flow Instructions: { }	57
Table 46.	Control Flow Instructions: @	57
Table 47.	Control Flow Instructions: BRA	58
Table 48.	Control Flow Instructions: CALL	58
Table 49.	Control Flow Instructions: RET	59
Table 50.	Control Flow Instructions: EXIT	59
Table 51.	Parallel Synchronization and Communication Instructions: BAR	60
Table 52.	Parallel Synchronization and Communication Instructions: ATOM	61
Table 53.	Floating-point Instructions: RCP	62
Table 54.	Floating-point Instructions: SQRT	62
Table 55.	Floating-point Instructions: RSQRT	63
Table 56.	Floating-point Instructions: SIN	63
Table 57.	Floating-point Instructions: COS	63
Table 58.	Floating-point Instructions: LG2	63
Table 59.	Floating-point Instructions: EX2	64
Table 60.	Miscellaneous Instructions: TRAP	65
Table 61.	Miscellaneous Instructions: BRKPT	65
Table 62.	Special Registers: %tid	67
Table 63.	Special Registers: %ntid	68
Table 64.	Special Registers: %ctaid	68
Table 65.	Special Registers: %nctaid	69
Table 66.	Special Registers: %gridid	69

Table 67.	Special Registers: %clock	69
Table 68.	Directives: .entry	71
Table 69.	Directives: .func	72
Table 70.	Debugging Directives: .section	73
Table 71.	Debugging Directives: .file	73
Table 72.	Debugging Directives: .loc	73
Table 73.	Debugging Directives: .byte	73
Table 74.	Other Directives: .extern	74
Table 75.	Other Directives: .visible	74
Table 76.	Other Directives: .version	74
Table 77.	Other Directives: .target	75
Table 78.	Summary of Instruction Changes in Version 1.1	79

Chapter 1.

Introduction

1.1. Data-Parallel Computing Using GPUs

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

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets such as arrays can use a data-parallel programming model to speed up the computations. Data-parallel mapping is efficient for SIMD, vector, and highly multi-threaded parallel architectures. 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. Many compute-intensive applications map well to data-parallel processing. In general, all algorithms that can be formulated as parallel computations operating over datasets are good candidates for acceleration by data-parallel processing.

PTX defines a virtual machine and virtual 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 enables 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 for G80 and subsequent GPUs. High level language compilers for languages such as C and 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 virtual ISA and VM 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.
- ❑ Programmability – facilitate hand-coding of libraries, performance kernels, and architecture tests.
- ❑ Scalability – VM programming model spans GPU sizes from single unit to many parallel units.
- ❑ Provide a relatively low-level ISA and machine model that can be usefully thought of as representing the target GPU architecture.
- ❑ VM and virtual ISA will become publicly visible.
- ❑ Component of the NV Compute product.
- ❑ Compatibility – version 1 programs execute on later translators.
- ❑ Sufficient quality and usability to evolve into an industry standard.

1.3. The Document's 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 instruction set.
- ❑ Chapter 8 lists special registers.
- ❑ Chapter 9 lists the assembly directives supported in PTX.
- ❑ Chapter 10 provides release notes for Release 1.1 of PTX Version 1.1.

Chapter 2.

Programming Model

2.1. A Highly Multithreaded Coprocessor

The GPU is a compute device capable of executing a very high 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 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 Batching

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.

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 are suspended until they all reach the synchronization point.

Each thread has a unique thread id (tid) 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 tid to determine its assigned role, assign specific input and output position, compute addresses, and select work to perform. The tid is a 3-component vector, tid.0, tid.1, and tid.2, that specifies the thread's position within a 1D, 2D, or 3D CTA. Alternate component names are tid.x, tid.y, and tid.z. Each tid component ranges from 0 up to the number of thread id's in that CTA dimension.

Each CTA has a 1D, 2D, or 3D shape, specified by a 3-component vector, `ntid`, which specifies the number of threads in each CTA dimension. The `ntid` components are accessible as `ntid.0`, `ntid.1`, and `ntid.2`.

Threads within a CTA execute in SIMD 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 16 or 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 id (`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 id (`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).

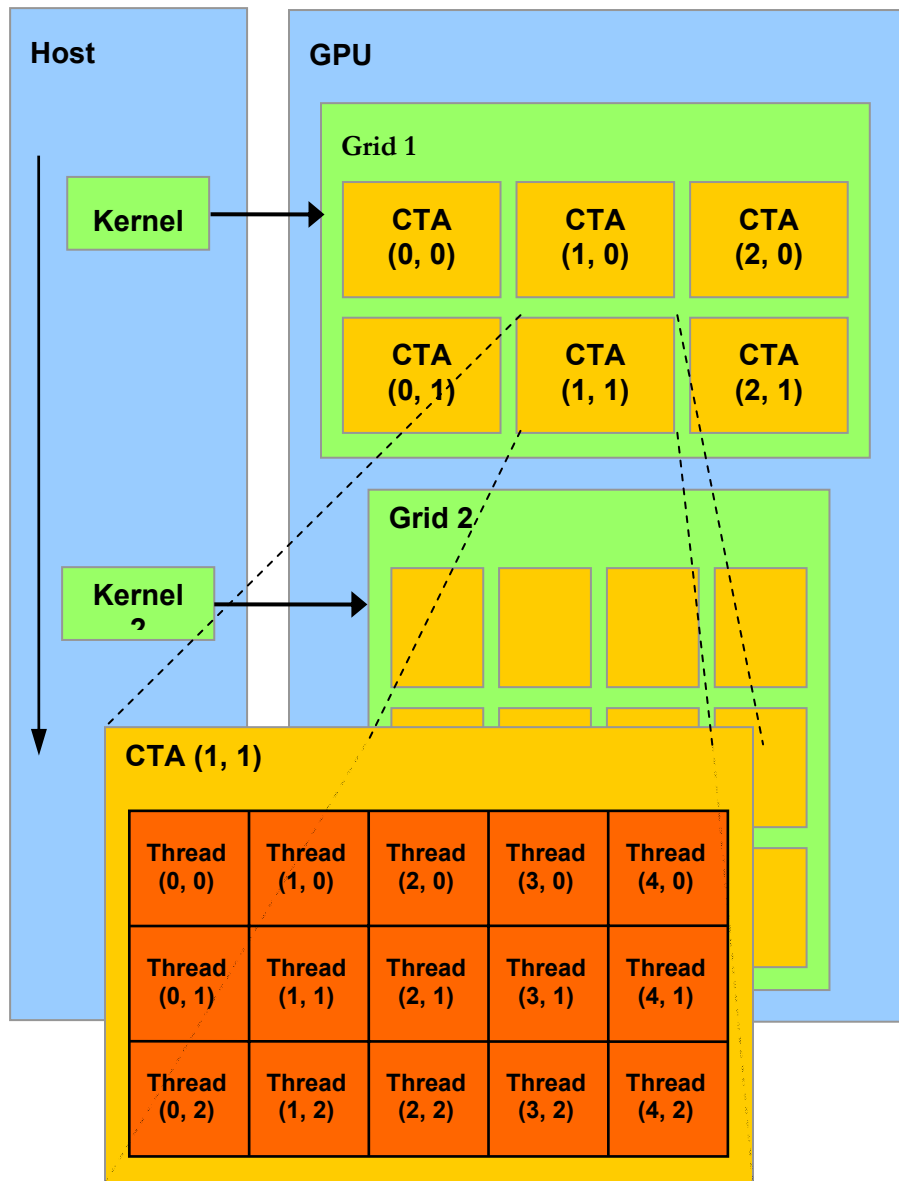


Figure 1. Thread Batching

This page is blank.

Chapter 3.

Parallel Thread Execution Machine Model

3.1. A Set of SIMD Multiprocessors with On-Chip Shared Memory

The PTX machine model is implemented as a set of multiprocessors as illustrated in Figure 2Error! Reference source not found.. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

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.

Each multiprocessor has on-chip memory of the four following types:

- ❑ One set of local 32-bit registers per processor,
- ❑ Shared memory that is shared by all the processors,
- ❑ A read-only constant cache that is shared by all the processors and speeds up reads from the constant memory, which is a read-only region of the device memory,
- ❑ A read-only texture cache that is shared by all the processors and dedicated to texture sampling.

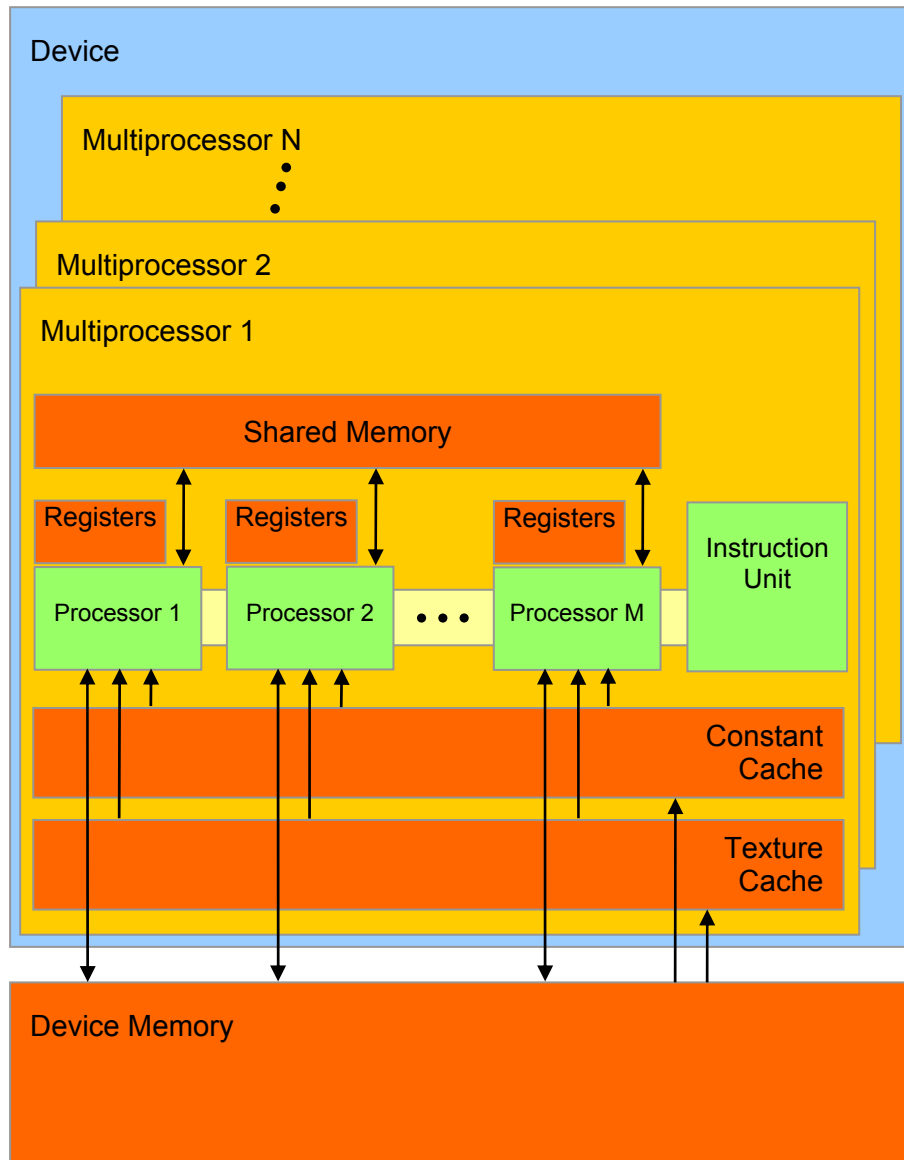


Figure 2. Machine Model
A set of SIMD multiprocessors with on-chip shared memory

3.2. Execution Model

A grid of CTAs is executed on the device by executing one or more CTAs on each multiprocessor using time slicing: Each CTA is split into SIMD groups of threads called warps; each of these warps contains the same number of threads, called the warp size, and is executed by the multiprocessor in a SIMD fashion; a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources.

The way a CTA is split into warps is always the same; each warp contains threads of consecutive, increasing thread indices with the first warp containing thread 0.

A CTA is processed by only one multiprocessor, so that threads within a CTA can use the on-chip shared memory to efficiently share data among them. More precisely, threads can perform general reads from and writes to the on-chip shared memory through a per-CTA shared memory partition and coordinate these memory accesses through synchronization mechanisms.

A multiprocessor can process several CTAs concurrently by partitioning its resources (e.g. registers and shared memory) among them.

Threads can access several other memory partitions:

- ❑ Threads can perform general cached reads from the constant memory through a per-grid constant memory partition.
- ❑ Threads can perform general non-cached reads from and writes to the device memory through two device memory partitions: a per-thread local memory partition and a per-grid global memory partition.
- ❑ At last, another way to perform general cached reads from the device memory is through texture sampling.

This memory model is illustrated in Figure 3.

The issue order of the CTAs within a grid is not defined and there is no synchronization mechanism between CTAs, so threads from two different CTAs of the same grid cannot safely communicate with each other through the device memory.

Figure 3 shows

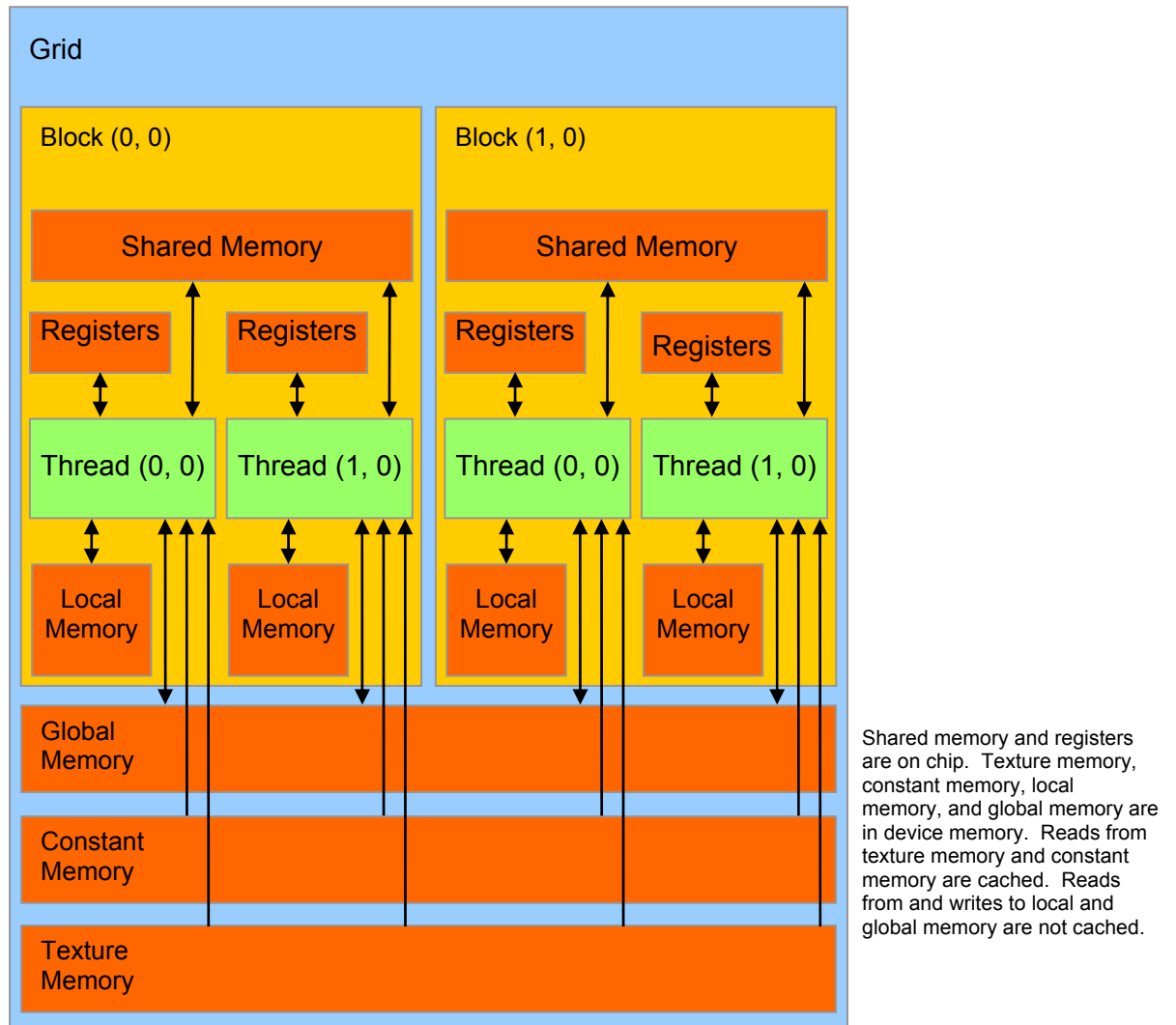


Figure 3. Memory Model

Chapter 4.

Syntax

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

4.1. Source Format

Source files 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 files. 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 file 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 to the end of the current line.

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.0;
          shl.b32    r1, r1, 2;          // shift thread id by 2 bits
          ld.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 9.

Table 1. PTX Directives

.align	.func	.section	.tex
.byte	.global	.shared	.union
.const	.local	.sreg	.version
.entry	.loc	.struct	.visible
.extern	.param	.surf	
.file	.reg	.target	

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	ex2	not	sin
add	exit	or	slct
and	ld	rcp	sqrt
atom	lg2	rem	st
bar	mad	ret	sub
bra	mad24	rsqrt	tex
brkpt	max	sad	trap
call	min	selp	vote
cnot	mov	set	xor
cos	mul	setp	
cvt	mul24	shl	
div	neg	shr	

4.4. Identifiers

User-defined identifiers follow extended C++ rules: they start with an alphabetic character, underscore, dollar sign, or percentage sign ([A-Za-z_ \$ %]) and are followed by zero or more alphanumeric, underscore, or dollar sign characters ([A-Za-z_ \$ %]).

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 a small number of special registers that begin with the percentage sign, listed in Table 3.

Table 3. Predefined identifiers

%clock	%ctaid	%ntid
%gridid	%nctaid	%tid

4.5. Immediate Constants

Immediate constants in PTX are restricted to integer and floating-point types.

4.5.1. Integer Immediate Constants

Integer immediate constants may be written in decimal, hexadecimal, octal, or binary notation.

Decimal constants begin with a nonzero digit followed by zero or more digits (0-9).

Hexadecimal constants begin with `0x` or `0X` followed by one or more hex digits (from the set [0-9a-fA-F]).

Octal constants begin with zero `0` followed by zero or more octal digits (0-7).

Binary constants begin with `0b` or `0B` followed by one or more binary digits (01).

4.5.2. Floating-point Immediate Constants

Floating-point immediate constants 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 (e.g. float or double).

PTX includes a second representation of floating-point constants, where the exact machine representation is given as a hexadecimal constant. For 64-bit floating point values, the constant begins with `0d` or `0D` followed by 16 hex digits. For 32-bit floating point values, the constant begins with `0f` or `0F` followed by 8 hex digits.

4.5.3. Predicate Immediate Constants

Predicate immediate constants for the Boolean values `TRUE` and `FALSE` are written as binary digits 1 and 0, respectively.

4.5.4. Constant Expressions

Constant expressions are evaluated at compile time to form simple values for use in immediate operands and addressing expressions. Both integer and floating-point constant expressions are supported, however, note that floating-point constant expressions may evaluate to a different value than would be computed on the target architecture, since the compiler may evaluate the expression using greater precision than the target architecture.

Constant expressions are formed from lexical constants, basic arithmetic operators (addition, subtraction, multiplication, division), and parentheses. Integer constant expressions may include remainder (`%`), shift operators (`<<` and `>>`), and logical operators (`&`, `|`, and `^`).

The meaning of operators in PTX is the same as in C or C++.

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 4, and properties of state spaces are shown in Table 5.

Table 4. State Spaces

Name	Description
.reg	Registers, fast.
.sreg	Special registers. Read-only; pre-defined; platform-specific.
.const	Per-CTA, shared, read-only memory.
.global	Global memory, shared by all threads.
.local	Local memory, private to each thread.
.param	User parameters for a program, available at CTA entry.
.shared	Addressable memory shared between threads in 1 CTA.
.surf	Global surface memory.
.tex	Global texture memory.

Table 5. Properties of State Spaces

Name	Addressible	Initializable	Access	Sharing
.reg	No	No	R/W	per-thread
.sreg	No	No	RO	per-CTA
.const	Yes	Yes	RO	per-grid
.global	Yes	Yes	R/W	Context
.local	Yes	No	R/W	per-thread
.param	Yes	No	RO	per-grid
.shared	Yes	No	R/W	per-CTA
.surf	via LD/ST, SURF instructions	Yes	R/W	Context
.tex	via TEX instruction	Yes	RO	Context

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 Error! Reference source not found. 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, registers have a width of 16-, 32-, or 64-bits.

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.

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

5.1.2. Special Register 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. The size may be limited, and there are typically many banks of constant memory, denoted by an integer index. The size and number of banks are listed in the appendix for different hardware.

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.

For any thread in a context, all addresses in global memory are shared.

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.

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.

5.1.6. Parameter State Space

The parameter (.param) state space provides addressable user parameters to CTAs. User parameters begin at address zero, and the address space is shared across CTAs within a grid.

The location of parameter space is implementation specific. For example, in some implementations, parameter space resides in global memory. No access protection is provided between parameter and global space in this case.

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

The texture (`.tex`) state space is global memory for the texture instructions. It is shared by all threads in a context.

The GPU hardware has a fixed number of texture bindings that can be accessed within a single program (typically 128). The `.tex[i]` directive will bind the named texture memory variable to the hardware texture id `i`. If no id number is given, PTX will assign texture id's sequentially, beginning with zero. Multiple names may be bound to the same physical texture id. An error is generated only if the texture id assigned is out of the physical texture id range (e.g., 0..127).

Texture memory is read-only.

Example:

```
.tex    tex_a;           // bound to physical texture 0
.tex[2] tex_b;           // bound to physical texture 2
.tex    tex_c;           // bound to physical texture 1
.tex    tex_d;           // bound to physical texture 2
.tex[42] tex_e;          // bound to physical texture 42
.tex    tex_f;           // bound to physical texture 3
```

5.1.9. Surface State Space

The surface (`.surf`) state space is similar to global memory, but is 2D in nature. It takes a 2D address (i and j components), and with respect to cache, spatial locality generally works well in a 2D neighborhood. This allows tiled decompositions to perform quite well.

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.

The following table lists the fundamental type specifiers for each basic type:

Table 6. Fundamental 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 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 and .s8 types are restricted to `ld`, `st`, and `cvt` instructions. The `ld` and `st` instructions also accept .b8 type. Byte-size integer load instructions zero- or sign-extended the value to the size of the destination register.

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.

5.3. 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 aggregate objects such as vectors, arrays, structures and unions.

5.3.1. 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.

Examples:

```
.global .u32 loc;
.reg .s32 i = 0;
.shared .f32 bias[] = {-1.0, 1.0};
.local .u8 bg[4] = {0, 0, 0, 0};
.reg .v3 .f32 accel;
.struct float4 { .f32[4] v };
.global float4 coord;
```

Note that texture and surface variables do not have an associated type and size.

5.3.2. Vectors

Limited-length vector types are supported. Vectors of length 2, 3, and 4 of any fundamental type can be declared by prefixing the type with `.v2`, `.v3`, or `.v4`. Vectors must be based on a fundamental type, and they may reside in the register space.

Examples:

```
.global .v4 .f32 V; // a length-4 vector of floats
.shared .v2 .u16 uv; // a length-2 vector of unsigned ints
.reg .v4 .pred vpred; // a vector of predicates registers
```


5.3.3. 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 the dimension is either a constant expression, or is left empty, being determined by an array initializer. Here are some examples:

```
.local .u16 kernel[19][19];
.shared .u8 mailbox[128];
.shared .s32 offset[][] = { {-1, 0}, {0, -1}, {1, 0}, {0, 1} };
```

The size of the array specifies how many elements should be reserved. For the kernel declaration above, 19*19 (361) halfwords are reserved (722 bytes).

5.3.4. Structures and Unions

A structure definition specifies a sequence of fields (consisting of a **type/size** and a **name**) as a block of memory. This is analogous to the structures in C. Once defined, the structure can be used as a type designator in subsequent variable declarations.

Example:

```
.struct somestruct { .s32 i; .s32 j; .f32 x; .f32 y; };
.global somestruct p;
.reg .b32 ptr;

...

ld.s32 r0, [p.x];
mov.b32 ptr, p;    // get address of structure p
```

Unions definitions use the same syntax as struct definitions, with the keyword **.struct** replaced by **.union**. The difference between a struct and a union is that in a struct, the fields are laid out sequentially in memory, while in a union, the fields all use the same memory. Unions provide a way to reuse memory in a relatively type-safe manner. Here is an example that provides storage for a float or an integer:

```
.union intOrFloat { .s32 i; .f32 f; };
```

Structure and union declarations may be nested. The shortcut syntax of C++ with anonymous unions is also supported.

5.3.5. Initializers

All declarations may specify an initial value for the variable being declared (including predicates). The initializers follow the conventions of C/C++, where the variable name is followed by an equals sign and the value or values for the initial values of 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). Structures take a list of values that matches the fields in a structure.

Examples:

```
.global .s32 n = 10;
.shared .f16 blur_kernel[][]
    = {{.05,.1,.05},{.1,.4,.1},{.05,.1,.05}};
.global .v3 .u8 rgb[3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
```

Initializers for thread-private memory all initialize their variables to the same value. There is no syntax for per-thread initializers.

5.3.6. 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 `space-state` specifier. The variable will be aligned to an address which is an integer multiple of `byte_count`. For arrays, structures, and unions, alignment specifies the address alignment for the starting address of the entire structure, not for individual elements.

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};
```

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 can be any declared variable, array element, structure/union member, vector or vector element.

6.4. Using Addresses, Arrays, Vectors, Structures, and Unions

Using scalar variables as operands is straightforward. The interesting capabilities begin with pointers, composite structures, and arrays.

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. 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 .f16 V;
.reg .v4 .f16 W;
.const .s32 tbl[256];
.reg .b32 p;
.reg .s32 q;

ld.u16    r0,[x];
ld.v4.f16 W, [V];
ld.s32    q, [tbl+12];
mov.b32   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.u32  s, a[0];
ld.u32  s, a[N-1];
mov.u32 s, a[1];           // move address of a[1] into s
```

6.4.3. Vectors as Operands

Vectors can be treated as a collection of elements simply by naming them. Vector variables can typically replace scalar variables in most PTX instructions, and the meaning is to perform the operation on an element-by-element basis.

```
.reg .v4 .f16 v1, v2, v3;
add.v4.s32 v3, v2, v1;
```

Vector elements can be extracted from the vector with the suffixes .0, .1, .2, and .3 or .x, .y, .z and .w suffixes, as well as the typical color fields .r, .g, .b and .a.

Vectors can be swizzled or reordered with swizzling suffixes, which are a combination of the digits or characters that represent the elements of a vector (0123, xyzw, rgba). The swizzling suffixes allow arbitrary duplication and reordering of vector elements. Swizzling is allowed only in `mov` instructions, and the source and destination must be distinct.

A brace-enclosed list is used for pattern matching to pull apart vectors. Wide loads and stores can be specified to multiple targets using vector loads, specifying multiple scalars within the brace-enclosed list. Here are some examples:

```
.reg .v3 .f32 V;
.reg .f32 a, b, c;
mov.v3.f32 {a,b,c}, 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 is an example:

```
ld.v4.f32 {a,b,c,d}, [Vmem];
```

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

```
Ra = V.0 = V.x = V.r
Rb = V.1 = V.y = V.g
Rc = V.2 = V.z = V.b
Rd = V.3 = V.w = V.a
```

6.4.4. Structures and Unions as Operands

Structures and unions can only access their members; there are no instructions that take entire structures as operands.

6.4.5. Immediate Values as Operands

Immediate values (or constants) can be used in most instructions. Only one immediate operand is permitted in an instruction. In ALU instructions, it is typically the **b** or **c** operand. In load and store instructions, an immediate offset to a register or an immediate absolute address is permitted. In instruction with only one source operand, the source operand may be an immediate. The size of the immediate value may be specified with a type suffix like `.u16`, and defaults to the size of the instruction source operand.

For directly specifying IEEE-752 single and double precision floating point numbers, a hexadecimal value may be used as an immediate operand in floating point operations. The immediate value syntax is as follows:

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

Example:

```
mov.f32 $f3, 0F3f80000; // 1.0
```

This format may also be used when initializing variables.

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 6 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 `Inf` for `f32` and `f64`, and `~131,000` for `f16`).

Table 7. CVT 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-bits, then sign-extend to 32-bits.										

6.5.2. Rounding Modes

Conversion instructions may specify a rounding modifier. In PTX, there are four integer rounding modes and four floating-point rounding modes. The following tables summarize the rounding modes.

Table 8. Floating-point Rounding Modes

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 9. Integer Rounding Modes

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

6.5.3. Vector Conversions

Conversions between scalar values and vector values are supported, allowing operations like adding the scalar value 1 to a vector. Scalar values are spread out to match the size of the vector. Short vectors are zero-extended to longer vectors, and long vectors are truncated when assigned to shorter vectors. The table below describes the conversions, where *s* is a scalar value and *v* is a vector.

Table 10. Conversions Between Scalar Values and Vector Values

Scalar-Vector Vector-Vector Conversions		Destination			
		scalar	v2	v3	v4
Source	scalar	-	[s, s]	[s, s, s]	[s, s, s, s]
	v2	v.0	-	[v.0, v.1, 0]	[v.0, v.1, 0, 0]
	v3	v.0	[v.0, v.1]	-	[v.0, v.1, v.2, 0]
	v4	v.0	[v.0, v.1]	[v.0, v.1, v.2]	-

Vector immediate values are specified similarly to aggregate initialization, but are not necessary unless the values are different (scalars are spread automatically). Some examples are shown below.

```
.global .v3 .f32 v;
add.v3.f32 v, v, 1;
```

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 11 gives estimates of the costs of using different kinds of memory.

Table 11. 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.

Instruction Set

7.1. 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.

7.2. 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 the source operands.

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

```
setp.s32.lt 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.

7.3. 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:    ...
```

7.3.1. Comparisons

7.3.1.1. 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. The following table shows the operators for signed integer, unsigned integer, and bit-size types.

Table 12. 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	
$a \leq b$	LE	LS	
$a > b$	GT	HI	
$a \geq b$	GE	HS	

7.3.1.2. Floating-point Comparisons

The ordered comparisons are **eq**, **ne**, **lt**, **le**, **gt**, **ge**. If either operand is **NaN**, the result is false.

Table 13. Floating-point Comparison Operators

Meaning	Floating-Point Operator
$a == b \ \&\& \ !\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	EQ
$a != b \ \&\& \ !\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	NE
$a < b \ \&\& \ !\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	LT
$a \leq b \ \&\& \ !\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	LE
$a > b \ \&\& \ !\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	GT
$a \geq b \ \&\& \ !\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	GE

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.

Table 14. Floating-Point Comparison Operators Accepting NaN

Meaning	Floating-Point Operator
$a == b \parallel \text{isNaN}(a) \parallel \text{isNaN}(b)$	EQU
$a != b \parallel \text{isNaN}(a) \parallel \text{isNaN}(b)$	NEU
$a < b \parallel \text{isNaN}(a) \parallel \text{isNaN}(b)$	LTU
$a <= b \parallel \text{isNaN}(a) \parallel \text{isNaN}(b)$	LEU
$a > b \parallel \text{isNaN}(a) \parallel \text{isNaN}(b)$	GTU
$a >= b \parallel \text{isNaN}(a) \parallel \text{isNaN}(b)$	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 15. Floating-Point Comparison Operators Testing for NaN

Meaning	Floating-Point Operator
$!\text{isNaN}(a) \ \&\& \ !\text{isNaN}(b)$	NUM
$\text{isNaN}(a) \parallel \text{isNaN}(b)$	NAN

7.3.2. 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
```

7.4. Type Information for Instructions and Operands

Instructions that have a type must have a type suffix, e.g. `add.u16` or `add.f32`. The operand type must agree with the instruction type suffix. The bit-size types agree with any type of the same size. 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:

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

Integer types are compatible 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.

Example:

```
.reg .u32 x;
.reg .s32 a;

neg.s32 a, x;        // signed negation of x
```

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

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

7.5. 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 reconverge 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 reconvergence. 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.

7.6. 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.

7.6.1. Machine-specific Semantics of 16-bit Code

A PTX program may execute on a GPU with either a 16-bit or a 32-bit datapath. When executing on a 32-bit datapath, 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 datapath. This approach introduces a performance penalty for 16-bit code executing on a 32-bit datapath, 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 choose 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.

7.7. Instructions

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

7.7.1. Arithmetic Instructions

Arithmetic instructions operate on the numeric types in register, vector, and constant immediate forms. The arithmetic instructions are:

- ❑ ADD
- ❑ SUB
- ❑ MUL
- ❑ MAD
- ❑ MUL24
- ❑ MAD24
- ❑ SAD
- ❑ DIV
- ❑ REM
- ❑ ABS
- ❑ NEG
- ❑ MIN
- ❑ MAX

Table 16. Arithmetic Instructions: ADD

ADD	Add two values
Syntax	<pre>add[.rnd][.sat].type d, a, b; .type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Performs addition and writes the resulting value into a destination register.
Semantics	$d = a + b$;
Integer Notes	<p>No integer rounding modes.</p> <p>Saturation modifier: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type.</p>
Floating Point Notes	<p>Rounding modes (default is .rn): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero</p> <p>Saturation modifier: .sat limits result to (0.0, 1.0). Applies only to .f32 type.</p>
Examples	<pre>@p add.u32 x,y,z; add.sat.s32 c,c,1; add.rz.f32 f1,f2,f3;</pre>

Table 17. Arithmetic Instructions: SUB

SUB	Subtract one value from another
Syntax	<pre>sub[.rnd][.sat].type d, a, b; .type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Performs subtraction and writes the resulting value into a destination register.
Semantics	$d = a - b$;
Integer Notes	<p>No integer rounding modes.</p> <p>Saturation modifier: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type.</p>
Floating Point Notes	<p>Rounding modes (default is .rn): .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero</p> <p>Saturation modifier: .sat limits result to (0.0, 1.0). Applies only to .f32 type.</p>
Examples	<pre>sub.s32 c,a,b;</pre>

Table 18. Arithmetic Instructions: MUL

MUL	Multiply two values
Syntax	<pre>mul[.hi,.lo,.wide][.rnd][.sat].type d, a, b;</pre> <pre>.type = { .u16, .u32, .u64,</pre> <pre> .s16, .s32, .s64,</pre> <pre> .f32, .f64 };</pre>
Description	Compute the product of two values.
Semantics	<pre>t = a * b;</pre> <pre>n = bitwidth of type;</pre> <pre>d = t; // for floating-point and .wide</pre> <pre>d = t<2n-1..n>; // for .hi variant</pre> <pre>d = t<n-1..0>; // for .lo variant</pre>
Integer Notes	<p>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.</p> <p>The .wide suffix is supported only for 16- and 32-bit integer types.</p> <p>No integer rounding modes.</p> <p>No integer saturation.</p>
Floating Point Notes	<p>For floating-point multiplication, all operands must be the same size.</p> <p>Rounding modes (default is .rn):</p> <pre>.rn mantissa LSB rounds to nearest even</pre> <pre>.rz mantissa LSB rounds towards zero</pre> <p>Saturation modifier:</p> <pre>.sat limits result to (0.0, 1.0).</pre> <p>Applies only to .f32 type.</p>
Examples	<pre>mul.wide.s16 fa,fxs,fys; // 16*16 bits yields 32 bits</pre> <pre>mul.lo.s16 fa,fxs,fys; // 16*16 bits, save only the low 16 bits</pre> <pre>mul.wide.s32 z,x,y; // 32*32 bits, creates 64 bit result</pre> <pre>mul.f32 circumf,radius,pi // a single-precision multiply</pre>

Table 19. Arithmetic Instructions: MAD

MAD	Multiply two values and add a third value
Syntax	<pre>mad[.hi,.lo,.wide][.sat].type d, a, b, c;</pre> <pre>.type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Multiplies two values and adds a third, and then writes the resulting value into a destination register.
Semantics	<pre>t = a * b;</pre> <pre>n = bitwidth of type;</pre> <pre>d = t + c; // for floating-point and .wide</pre> <pre>d = t<2n-1..n> + c; // for .hi variant</pre> <pre>d = t<n-1..0> + c; // for .lo variant</pre>
Integer Notes	<p>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.</p> <p>The .wide suffix is supported only for 16- and 32-bit integer types.</p> <p>No integer rounding modes.</p> <p>Saturation modifier:</p> <p>.sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type in .hi mode.</p>
Floating Point Notes	<p>Saturation modifier:</p> <p>.sat limits result to (0.0, 1.0). Applies only to .f32 type.</p>
Examples	<pre>mad.lo.s32 d,a,b,c;</pre> <pre>mad.lo.s32 r,p,q,r;</pre> <pre>@p mad.f32 d,a,b,c;</pre>

Table 20. Arithmetic Instructions: MUL24

MUL24	Multiply two 24-bit integer values
Syntax	<pre>mul24[.hi,.lo].type d, a, b; .type = { .u32, .s32 };</pre>
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	<pre>t = a * b; d = t<47..16>; // for .hi variant d = t<31..0>; // for .lo variant</pre>
Notes	<p>Integer multiplication yields a result that is twice the size of the input operands, i.e. 48-bits.</p> <p>mul24.hi performs a 24x24-bit multiply and returns the high 32 bits of the 48-bit result.</p> <p>mul24.lo performs a 24x24-bit multiply and returns the low 32 bits of the 48-bit result.</p> <p>All operands are of the same type and size.</p> <p>No saturation.</p> <p>mul24.hi may be less efficient on machines without hardware support for 24-bit multiply.</p>
Examples	<pre>mul24.lo.s32 d,a,b; // low 32-bits of 24x24-bit signed multiply.</pre>

Table 21. Arithmetic Instructions: MAD24

MAD24	Multiply two 24-bit integer values and add a third value.
Syntax	<pre>mad24[.hi,.lo][.sat].type d, a, b, c; .type = { .u32, .s32 };</pre>
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	<pre>t = a * b; d = t<47..16> + c; // for .hi variant d = t<31..0> + c; // for .lo variant</pre>
Notes	<p>Integer multiplication yields a result that is twice the size of the input operands, i.e. 48-bits.</p> <p>mad24.hi performs a 24x24-bit multiply and adds the high 32 bits of the 48-bit result to a third value.</p> <p>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.</p> <p>Saturation modifier:</p> <p>.sat limits result of 32-bit signed addition to MININT..MAXINT (no overflow). Applies only to .s32 type in .hi mode.</p> <p>mad24.hi may be less efficient on machines without hardware support for 24-bit multiply.</p>
Examples	<pre>mad24.lo.s32 d,a,b,c; // low 32-bits of 24x24-bit signed multiply.</pre>

Table 22. Arithmetic Instructions: SAD

SAD	Sum of absolute differences.
Syntax	<pre>sad.type d, a, b, c; .type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Adds the absolute value of a-b to c and writes the resulting value into a destination register.
Semantics	$d = c + ((a < b) ? (b - a) : (a - b));$
Examples	<pre>sad.s32 d,a,b,c; sad.u32 d,a,b,d; // running sum sad.f32 w,x,y,z;</pre>

Table 23. Arithmetic Instructions: DIV

DIV	Divide one value by another.
Syntax	<pre>div[.wide][.sat].type d, a, b; .type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Divides a by b , stores result in d .
Semantics	$d = a / b;$
Integer Notes	<p>The .wide suffix specifies that a is twice the size of b and d. Otherwise, all three operands are the same size.</p> <p>The .wide suffix is supported only for 16- and 32-bit integer types.</p> <p>Division by zero yields an unspecified, machine-specific value.</p> <p>No integer saturation.</p>
Floating Point Notes	<p>Division by zero creates a value of infinity (with same sign as a).</p> <p>Division rounds to nearest even.</p> <p>Saturation modifier:</p> <p>.sat limits result to (0.0, 1.0).</p> <p>Applies only to .f32 type.</p>
Release Notes	div.wide and div.{u64,s64} are unimplemented.
Examples	<pre>div.s32 b,n,i; div.wide.s32 d,an_s64_var,b; div.f32 diam,circum,3.14159;</pre>

Table 24. Arithmetic Instructions: REM

REM	The remainder of integer division.
Syntax	<pre>rem[.wide].type d, a, b;</pre> <pre>.type = { .u16, .u32, .u64, .s16, .s32, .s64 };</pre>
Description	Divides a by b , store the remainder in d .
Semantics	$d = a \% b;$
Integer Notes	<p>The .wide suffix specifies that a is twice the size of b and d. Otherwise, all three operands are the same size.</p> <p>The .wide suffix is supported only for 16- and 32-bit integer types.</p> <p>The behavior for negative numbers is machine-dependent and depends on whether divide rounds towards zero or negative infinity.</p>
Floating Point Notes	No floating-point support.
Release Notes	rem.wide and rem.{u64,s64} are unimplemented.
Examples	<pre>rem.s32 x,x,8; // x = x%8;</pre>

Table 25. Arithmetic Instructions: ABS

ABS	Absolute value.
Syntax	<pre>abs.type d, a;</pre> <pre>.type = { .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Take the absolute value of a and store it in d .
Semantics	$d = a ;$
Examples	<pre>abs.s32 r0,a;</pre> <pre>abs.f32 x,f0;</pre>

Table 26. Arithmetic Instructions: NEG

NEG	Arithmetic negate.
Syntax	<pre>neg.type d, a;</pre> <pre>.type = { .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Subtract a from zero and store the result in d .
Semantics	$d = 0 - a;$
Notes	Only for signed integers and floating-point numbers.
Examples	<pre>neg.s32 r0,a;</pre> <pre>neg.f32 x,f0;</pre>

Table 27. Arithmetic Instructions: MIN

MIN	Find the minimum of two values.
Syntax	<pre>max.type d, a, b; .type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Store the minimum of a and b in d .
Semantics	<pre>d = (a < b) ? a : b; // Integer (signed and unsigned) d = isNaN(a) ? b : isNaN(b) ? a : (a < b) ? a : b; // FP</pre>
Integer Notes	Signed and unsigned differ.
Floating Point Notes	If either source operand is NaN, then the result is the other operand.
Examples	<pre>min.s32 r0,a,b; @p min.u16 h,i,j; min.f32 z,z,x;</pre>

Table 28. Arithmetic Instructions: MAX

MAX	Find the maximum of two values.
Syntax	<pre>min.type d, a, b; .type = { .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Store the maximum of a and b in d .
Semantics	<pre>d = (a > b) ? a : b; // Integer (signed and unsigned) d = isNaN(a) ? b : isNaN(b) ? a : (a > b) a : b; // FP</pre>
Integer Notes	Signed and unsigned differ.
Floating Point Notes	If either source operand is NaN, then the result is the other operand.
Examples	<pre>max.f32 f0,f1,f2; max.u32 d,a,b; max.s32 q,q,0;</pre>

7.7.2. Comparison and Selection Instructions

The comparison select instructions are:

- ❑ SET
- ❑ SETP
- ❑ SELP
- ❑ SLCT

Table 29. 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	<pre>set.CmpOp.dtype.type d, a, b; set.CmpOp.BoolOp.dtype.type d, a, b, [!]c; .dtype = { .u32, .s32, .f32 }; .type = { .b16, .b32, .b64, .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	<p>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.</p> <p>The comparison operator is a suffix on the instruction, and can be one of: eq, ne, lt, le, gt, ge lo, ls, hi, hs equ, neu, ltu, leu, gtu, geu num, nan</p> <p>The Boolean operator BoolOp(A,B) is one of: and, or, xor</p>
Semantics	<pre>t = (a CmpOp b) ? 1 : 0; if (isFloat(dtype)) d = BoolOp(t, c) ? 1.0f : 0x00000000; else d = BoolOp(t, c) ? 0xFFFFFFFF : 0x00000000;</pre>
Integer Notes	<p>The signed and unsigned comparison operators are eq, ne, lt, le, gt, ge.</p> <p>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.</p> <p>The untyped, bit-size comparisons are eq and ne.</p>
Floating Point Notes	<p>The ordered comparisons are eq, ne, lt, le, gt, ge. If either operand is NaN, the result is false.</p> <p>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.</p> <p>num returns true if both operands are numeric values (not NaN), and nan returns true if either operand is NaN.</p>
Examples	<pre>set.lt.and.f32.s32 d,a,b,r; set.eq.u32.u32 d,i,n;</pre>

Table 30. 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	<pre>setp.CmpOp.type p[q], a, b; setp.CmpOp.BoolOp.type p[q], a, b, [!]c;</pre> <p>.type = { .b16, .b32, .b64, .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</p>
Description	<p>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.</p> <p>Applies to all numeric types. The destinations p and q must be .pred variables.</p> <p>The comparison operator is a suffix on the instruction, and can be one of:</p> <pre>eq, ne, lt, le, gt, ge lo, ls, hi, hs equ, neu, ltu, leu, gtu, geu num, nan</pre> <p>The Boolean operator BoolOp(A,B) is one of: and, or, xor</p>
Semantics	<pre>t = (a CmpOp b) ? 1 : 0; p = BoolOp(t, c); q = BoolOp(!t, c);</pre>
Integer Notes	<p>The signed and unsigned comparison operators are <code>eq</code>, <code>ne</code>, <code>lt</code>, <code>le</code>, <code>gt</code>, <code>ge</code>.</p> <p>For unsigned values, the comparison operators <code>lo</code>, <code>ls</code>, <code>hi</code>, and <code>hs</code> for lower, lower-or-same, higher, and higher-or-same may be used instead of <code>lt</code>, <code>le</code>, <code>gt</code>, <code>ge</code>, respectively.</p> <p>The untyped, bit-size comparisons are <code>eq</code> and <code>ne</code>.</p>
Floating Point Notes	<p>The ordered comparisons are <code>eq</code>, <code>ne</code>, <code>lt</code>, <code>le</code>, <code>gt</code>, <code>ge</code>. If either operand is NaN, the result is false.</p> <p>To aid comparison operations in the presence of NaN values, unordered versions are included: <code>equ</code>, <code>neu</code>, <code>ltu</code>, <code>leu</code>, <code>gtu</code>, <code>geu</code>. 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.</p> <p><code>num</code> returns true if both operands are numeric values (not NaN), and <code>nan</code> returns true if either operand is NaN.</p>
Examples	<pre>setp.lt.and.s32 p q,a,b,r; setp.eq.u32 p,i,n;</pre>

Table 31. Comparison and Selection Instructions: SELP

SELP	Select between source operands, based on the value of the predicate source operand.
Syntax	<pre>selp.type d, a, b, c;</pre> <pre>.type = { .b16, .b32, .b64, .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
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	$d = (c == 1) ? a : b;$
Examples	<pre>selp.s32 r0,r,g,p;</pre> <pre>selp.f32 f0,t,x,xp;</pre>

Table 32. Comparison and Selection Instructions: SLCT

SLCT	Select one source operand, based on the sign of the third operand.
Syntax	<pre>slct.dtype.ctype d, a, b, c;</pre> <pre>.dtype = { .b16, .b32, .b64, .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre> <pre>.ctype = { .s32, .f32 };</pre>
Description	Conditional selection. If c >= 0, a is stored in d , b otherwise. 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.
Semantics	$d = (c \geq 0) ? a : b;$ For .f32 comparisons, if operand c is a denorm, it is flushed to zero, resulting in selection of operand a . If operand c is NaN, the comparison is unordered and operand b is selected.
Floating Point Notes	For .f32 data selections, denorm results are flushed to zero.
Examples	<pre>slct.u32.s32 x, y, z, val;</pre> <pre>slct.u64.f32 A, B, C, fval;</pre>

7.7.3. 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
- ❑ SHL
- ❑ SHR

Table 33. Logic and Shift Instructions: AND

AND	Bitwise AND .
Syntax	<code>and.type d, a, b;</code> <code>.type = { .pred, .b16, .b32, .b64 };</code>
Description	Compute the bit-wise and operation for the bits in a and b .
Semantics	<code>d = a & b;</code>
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
Examples	<code>and.b32 x,q,r;</code> <code>and.b32 sign,fpvalue,0x80000000;</code>

Table 34. Logic and Shift Instructions: OR

OR	Bitwise OR .
Syntax	<code>or.type d, a, b;</code> <code>.type = { .pred, .b16, .b32, .b64 };</code>
Description	Compute the bit-wise or operation for the bits in a and b .
Semantics	<code>d = a b;</code>
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
Examples	<code>or.b32 mask mask,0x00010001</code> <code>or.pred p,q,r;</code>

Table 35. Logic and Shift Instructions: XOR

XOR	Bitwise exclusive-OR (inequality).
Syntax	<code>xor.type d, a, b;</code> <code>.type = { .pred, .b16, .b32, .b64 };</code>
Description	Compute the bit-wise exclusive-or operation for the bits in a and b .
Semantics	$d = a \wedge b$;
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
Examples	<code>xor.b32 d,q,r;</code> <code>xor.b16 d,x,0x0001;</code>

Table 36. Logic and Shift Instructions: NOT

NOT	Bitwise negation; one's complement.
Syntax	<code>not.type d, a;</code> <code>.type = { .pred, .b16, .b32, .b64 };</code>
Description	Invert the bits in a .
Semantics	$d = \sim a$;
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicates.
Examples	<code>not.b32 mask,mask;</code> <code>not.pred p,q;</code>

Table 37. Logic and Shift Instructions: CNOT

CNOT	C/C++ style logical negation.
Syntax	<code>xor.type d, a, b;</code> <code>.type = { .pred, .b16, .b32, .b64 };</code>
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.
Examples	<code>cnot.b32 d,a;</code>

Table 38. Logic and Shift Instructions: SHL

SHL	Shift bits left, zero-fill on right.
Syntax	<code>shl.type d, a, b;</code> <code>.type = { .b16, .b32, .b64 };</code>
Description	Shift a left by the amount specified by b .
Semantics	<code>d = a << b;</code>
Notes	Shift amounts greater than the register width N are clamped to N . The size of the operands must match, but not necessarily the type.
Examples	<code>shl.b32 q,a,2;</code>

Table 39. Logic and Shift Instructions: SHR

SHR	Shift bits right, sign or zero fill on left.
Syntax	<code>shr.type d, a, b;</code> <code>.type = { .b16, .b32, .b64,</code> <code>.u16, .u32, .u64,</code> <code>.s16, .s32, .s64 };</code>
Description	Shift a right by the amount specified by b . Signed shifts fill with the sign bit, unsigned and untyped shifts fill with 0.
Semantics	<code>d = a >> b;</code>
Notes	Shift amounts greater than the register width N are clamped to N . Bit-size types are included for symmetry with SHL.
Examples	<code>shr.u16 c,a,2;</code> <code>shr.s32 i,i,1;</code> <code>shr.b16 k,i,j;</code>

7.7.4. 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.

The Data Movement and Conversion Instructions are:

- ❑ MOV
- ❑ LD
- ❑ ST
- ❑ CVT

Table 40. Data Movement and Conversion Instructions: MOV

MOV	Set a register variable with the value of a register variable or an immediate value.
Syntax	<pre> mov.type d, a; mov.type d, sreg; // sizes must match mov.type d, avar; // move address of variable into destination reg .type = { .pred, .b16, .b32, .b64, .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 }; </pre>
Description	Write register d with the value of a . Operand a may be a register, special register, immediate, or addressable variable.
Semantics	$d = a;$
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.
Examples	<pre> 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 </pre>

Table 41. Data Movement and Conversion Instructions: LD

LD	Load a register variable from an addressable state space variable.
Syntax	<pre>ld.space.type d,[a]; // load from address ld.space.vec.type d,[a]; // vector load from address ld.volatile.space.type d,[a]; // load from address ld.volatile.space.vec.type d,[a]; // vector load from address .space = { .const, .global, .local, .param, .shared }; .vec = { .v2, .v3, .v4 }; .type = { .b8, .b16, .b32, .b64, .u8, .u16, .u32, .u64, .s8, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	<p>Load register variable d from the location specified by the source address operand a.</p> <p>The addressable operand a is one of:</p> <ul style="list-style-type: none"> [avar] the name of an addressable variable var, [areg] a register reg containing a byte address, [areg+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). <p>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.</p> <p>The instruction must carry a .space suffix. A register containing an address may be declared as a bit-size type or integer type.</p> <p>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.</p>
Semantics	<pre>d = a; // named variable a d = *a; // register d = *(a+immOff); // register-plus-offset d = *(immAddr); // immediate address</pre>
Notes	<p>Destination d must be in the .reg state space.</p> <p>For integer loads, if the destination register is wider than the specified type, the value loaded is extended to the destination register width. The type of extension (sign or zero) is determined by the .type field.</p> <p>.f16 data may be loaded using ld.b16, and then converted to .f32 or .f64 using cvt.</p>
Examples	<pre>ld.global.f32 d,[a]; ld.shared.b32 d,[p]; ld.const.s32 d,[p+4]; ld.global.v4.f32 Q,[p]; ld.local.b64 x,[240];</pre>

Table 42. Data Movement and Conversion Instructions: ST

ST	Store a register variable to an addressable state space variable.
Syntax	<pre> st.space.type [d],a; // store to address st.space.vec.type [d],a; // vector store to address st.volatile.space.type [d],a; // store to address st.volatile.space.vec.type [d],a; // vector store to address .space = { .global, .local, .shared }; .vec = { .v2, .v3, .v4 }; .type = { .b8, .b16, .b32, .b64, .u8, .u16, .u32, .u64, .s8, .s16, .s32, .s64, .f32, .f64 }; </pre>
Description	<p>Store the value of register variable a in the location specified by the destination address operand d.</p> <p>The addressable operand d is one of:</p> <ul style="list-style-type: none"> [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). <p>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.</p> <p>The instruction must carry a .space suffix. A register containing an address may be declared as a bit-size type or integer type.</p> <p>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.</p>
Semantics	<pre> d = a; // named variable d *d = a; // register *(d+immOffset) = a; // register-plus-offset *(immAddr) = a; // immediate address </pre>
Notes	<p>Operand a must be in the .reg state space.</p> <p>.f16 data resulting from a cvt instruction may be stored using st.b16.</p>
Examples	<pre> st.global.f32 [d],a; st.local.b32 [q+4],a; st.global.v4.s32 [p],Q; st.shared.s32 [100],r7; </pre>

Table 43. Data Movement and Conversion Instructions: CVT

CVT	Convert a value from one type to another.
Syntax	<pre>cvt[.rnd][.sat].dtype.atype d, a;</pre> <p>.dtype = .atype = { .u8, .u16, .u32, .u64, .s8, .s16, .s32, .s64, .f16, .f32, .f64 };</p>
Description	Convert between different types and sizes. See the Integer and Floating-point Notes below for details of rounding modes.
Semantics	<code>d = convert(a);</code>
Integer Notes	<p>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.</p> <p>Integer rounding modes:</p> <ul style="list-style-type: none"> .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 <p>Saturation modifier:</p> <ul style="list-style-type: none"> .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies to both signed and unsigned integer types.
Floating Point Notes	<p>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.</p> <p>Floating-point rounding modes:</p> <ul style="list-style-type: none"> .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 <p>A floating-point value may be rounded to an integral value using the integer rounding modes (see Integer Notes). The operands must be of the same size. The result is an integral value, stored in floating-point format.</p> <p>Saturation modifier:</p> <ul style="list-style-type: none"> .sat limits result to (0.0, 1.0). Applies to .f16, .f32, and .f64 types. <p>NaN is preserved, except for .f16 (no NaN available).</p>
Examples	<pre>cvt.f32.s32 f,i; cvt.sat.s32.f64 j,r; cvt.rni.f32.f32 x,y; // round fp val to nearest int, result is fp</pre>

7.7.5. Texture Instruction

Table 44. Texture Instruction: TEX

TEX	Perform a texture memory lookup.
Syntax	<pre>tex.geom.dtype.btype d, a, b; .geom = { .1d, .2d, .3d }; .dtype = .btype = { .b16, .b32, .b64, .u16, .u32, .u64, .s16, .s32, .s64, .f32, .f64 };</pre>
Description	Texture lookup using a texture coordinate vector.
Examples	<pre>tex.3d.v4.s32.f32 {r1,r2,r3,r4},tex_a,{f1,f2,f3}; tex.1d.v4.s32.f32 {r1,r2,r3,r4},tex_a,{f1};</pre>

7.7.6. Control Flow Instructions

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

- {}
- @
- BRA
- CALL
- RET
- EXIT

Table 45. 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.
Examples	{ add.s32 a,b,c; mov.s32 d,a; }

Table 46. 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
Examples	<pre> setp.eq.f32 p,y,0; // is y zero? @!p div.f32 ratio,x,y // avoid division by zero @g bra L23; // conditional branch </pre>

Table 47. Control Flow Instructions: BRA

BRA	Branch to a target and continue execution there.
Syntax	<code>bra[.uni] target;</code>
Description	Continue execution at the target. Conditional branches are specified with the '@' prefix.
Semantics	<code>pc = target;</code>
Notes	A bra is assumed to be divergent unless the .uni suffix is present, indicating that the branch is guaranteed to be non-divergent.
Release Notes	Indirect branch through a register is unimplemented.
Examples	<pre> setp.eq.f32 p,y,0; // is y zero? @!p div.f32 ratio,x,y // avoid division by zero @q bra L23; // conditional branch </pre>

Table 48. Control Flow Instructions: CALL

CALL	Call a function, recording the return location.
Syntax	<code>call[.uni] fname;</code> <code>call[.uni] fname, (param-list);</code> <code>call[.uni] (ret-param), fname, (param-list);</code>
Description	Call a function, storing current execution information for subsequent return.
Notes	<p>The call instruction stores the address of the next instruction, so execution can resume at that point after executing a RET instruction.</p> <p>The called location can be either a symbolic function name or an address held in a register.</p> <p>A call is assumed to be divergent unless the .uni suffix is present, indicating that the call is guaranteed to be non-divergent.</p> <p>Input and return parameters are optional. Parameters must be of register type, and parameters are pass-by-value. In the current ptx release, parameters are passed through statically allocated ptx registers; i.e., there is no support for recursive calls..</p>
Release Notes	Indirect call through a register is unimplemented.
Examples	<pre> 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 </pre>

Table 49. Control Flow Instructions: RET

RET	Return from function to instruction after call.
Syntax	<code>ret[.uni];</code>
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.
Notes	<p>A ret is assumed to be divergent unless the .uni suffix is present, indicating that the return is guaranteed to be non-divergent.</p> <p>Any values returned from a function should be moved into the return parameter register variables prior to executing the RET instruction.</p> <p>A return instruction executed in a top-level entry routine will terminate thread execution.</p>
Examples	<pre>ret; @p ret;</pre>

Table 50. Control Flow Instructions: EXIT

EXIT	Terminate a thread.
Syntax	<code>exit;</code>
Description	Ends execution of a thread.
Examples	<pre>exit; @p exit;</pre>

7.7.7. Parallel Synchronization and Communication Instructions

These instructions are:

- BAR
- ATOM

Table 51. Parallel Synchronization and Communication Instructions: BAR

BAR	Signal arrival at a barrier and wait.
Syntax	<code>bar.sync d;</code>
Description	<p>Mark7s the arrival of threads at a barrier and waits for all other threads to arrive.</p> <p>The barrier resource is named via a small integer, typically in the range 0..15. The barrier number may be given as an immediate.</p>
Notes	<p>The hardware has a limited, implementation-specific number of barrier resources, typically sixteen or fewer. Since a CTA will not launch until all allocated resources are available, a program should minimize the number of distinct barrier variables allocated. Ideally, a program uses a single, global barrier that is re-used throughout the program.</p>
Examples	<code>bar.sync 0;</code>

Table 52. Parallel Synchronization and Communication
Instructions: ATOM

ATOM	Atomic reduction operations for thread-to-thread communication.
Syntax	<pre>atom.space.operation.type d, a, b[, c]; .space = { .global }; .operation = { .and, .or, .xor, // .b32 .cas, .exch, // .b32 .add, // .u32, .s32, .inc, .dec, // .u32 only .min, .max }; // .u32, .s32, .type = { .b32, .u32, .s32 };</pre>
Description	<p>Atomically loads the original value at location <i>a</i> into destination register <i>d</i>, and stores the result of the specified operation at location <i>a</i>, overwriting the original value. The a operand specifies a location in the specified state space.</p> <p>The addressable operand <i>a</i> is one of:</p> <ul style="list-style-type: none"> [avar] the name of an addressable variable <i>avar</i>, [areg] a de-referenced register <i>areg</i> containing a byte address, [areg+immOff] a de-referenced sum of register <i>areg</i> containing a byte address plus a constant integer byte offset, or [immAddr] an immediate absolute byte address. <p>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.</p> <p>The instruction must carry a <i>.space</i> suffix. A register containing an address may be declared as a bit-size type or integer type.</p> <p>The bit-size operations are and, or, xor, cas (compare-and-swap), and exch (exchange).</p> <p>The integer operations are add, inc, dec, min, max. The inc and dec operations return a result in the range [0..b].</p>
Semantics	<pre>atomic { d = *a; a = (operation == cas) ? operation(*a, b, c) : operation(*a, b); }</pre> <p>where</p> <pre>inc(r, s) = (r >= s) ? 0 : r+1; dec(r, s) = (r > s) ? s : r-1; exch(r, s) = s; cas(r,s,t) = (r == s) ? t : r;</pre>
Notes	<p>Operand a must reside the global state space.</p> <p>Simple reductions may be specified by using the “bit bucket” destination operand ‘_’.</p>
Target ISA Notes	atom.global requires sm_11 or later.
Examples	<pre>atom.global.add.s32 d,[a],1; @p atom.global.cas.b32 d,[p],my_val,my_new_val;</pre>

7.7.8. Floating-point Instructions

These instructions are for floating-point types in register, vector, and constant immediate forms. These instructions are:

- ❑ RCP
- ❑ SQRT
- ❑ RSQRT
- ❑ SIN
- ❑ COS
- ❑ LG2
- ❑ EX2

Table 53. Floating-point Instructions: RCP

RCP	Take the reciprocal of a value.
Syntax	<code>rcp.type d, a;</code> <code>.type = { .f32, .f64 };</code>
Description	Compute $1/a$.
Semantics	$d = 1/a;$
Examples	<code>rcp.f32 ri,r;</code>

Table 54. Floating-point Instructions: SQRT

SQRT	Take the square root of a value.
Syntax	<code>sqrt.type d, a;</code> <code>.type = { .f32, .f64 };</code>
Description	Compute sqrt(a) ; store in d .
Semantics	$d = \text{sqrt}(a);$
Floating Point Notes	If $a < 0$; $d = \text{NaN}$; The sqrt instruction always yields the positive root of a number, except for <code>sqrt(-0.0)</code> which yields <code>-0.0</code> .
Examples	<code>sqrt.f32 r,x;</code>

Table 55. Floating-point Instructions: RSQRT

RSQRT	Take the reciprocal of the square root of a value.
Syntax	<code>rsqrt.type d, a;</code> <code>.type = { .f32, .f64 };</code>
Description	Compute $1/\sqrt{a}$; store the result in d
Semantics	<code>d = 1/sqrt(a);</code>
Floating Point Notes	if <code>a < 0</code> ; <code>d = NaN</code> ; if <code>a == 0</code> , <code>d = Inf</code> ; The rsqrt instruction always yields a positive value, except for <code>rsqrt(-0.0)</code> which yields -0.0.
Examples	<code>rsqrt.f32 isr,x;</code>

Table 56. Floating-point Instructions: SIN

SIN	Find the sine of a value.
Syntax	<code>sin.type d, a;</code> <code>.type = { .f32 };</code>
Description	Find the sine of the angle a (in radians).
Semantics	<code>d = sin(a);</code>
Notes	Applies only to .f32.
Examples	<code>sin.f32 sa,a;</code>

Table 57. Floating-point Instructions: COS

COS	Find the cosine of a value.
Syntax	<code>cos.type d, a;</code> <code>.type = { .f32 };</code>
Description	Find the cosine of the angle a (in radians).
Semantics	<code>d = cos(a);</code>
Notes	Applies only to .f32.
Examples	<code>cos.f32 cb,b;</code>

Table 58. Floating-point Instructions: LG2

LG2	Find the log, base 2, of a value.
Syntax	<code>lg2.type d, a;</code> <code>.type = { .f32 };</code>
Description	Determine the \log_2 of a .
Semantics	<code>d = log(a)/log(2);</code>
Notes	Applies only to .f32.
Examples	<code>@p lg2.f32 q,a;</code>

Table 59. Floating-point Instructions: EX2

EX2	Exponentiate a value, base 2.
Syntax	<code>ex2.type d, a;</code> <code>.type = { .f32 };</code>
Description	Raise 2 to the power a .
Semantics	$d = 2 ^ a;$
Notes	Applies only to .f32.
Examples	<code>ex2.f32 q,r;</code>

7.7.9. Miscellaneous Instructions

The Miscellaneous instructions are:

- ❑ TRAP
- ❑ BRKPT

Table 60. Miscellaneous Instructions: TRAP

TRAP	Perform trap operation.
Syntax	<code>trap</code>
Description	Abort execution and generate an interrupt to the host CPU.
Examples	<pre>trap; @p trap;</pre>

Table 61. Miscellaneous Instructions: BRKPT

BRKPT	Breakpoint – suspend execution.
Syntax	<code>brkpt</code>
Description	Suspends execution
Target ISA Notes	Requires <code>sm_11</code> or later.
Examples	<pre>brkpt; @p brkpt;</pre>

This page is blank.

Chapter 8. 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
- ❑ %ctaid
- ❑ %nctaid
- ❑ %gridid
- ❑ %clock

Table 62. Special Registers: %tid

%tid	Thread ID within a CTA.
Syntax	<pre>.sreg .v3 .u16 %tid; // thread id vector .sreg .u16 %tid.0, %tid.1, %tid.2; // individual thread id components .sreg .u16 %tid.x, %tid.y, %tid.z; // alternate component names</pre>
Description	<p>A predefined, read-only, per-thread special register initialized with the thread ID within the CTA. The %tid special register is a 1D, 2D, or 3D vector to match the CTA shape; the %tid value in unused dimensions is 0. The number of threads in each dimension are specified by the predefined special register %ntid.</p> <p>Every thread in the CTA has a unique %tid.</p> <p>%tid component values range from 0 through %ntid-1 in each CTA dimension. %tid.1 == %tid.2 == 0 in 1D CTAs. %tid.2 == 0 in 2D CTAs.</p> <p>It is guaranteed that:</p> <pre>0 <= %tid.0 < %ntid.0 0 <= %tid.1 < %ntid.1 0 <= %tid.2 < %ntid.2</pre>
Notes	<p>3D CTA initialization code separates hardware %tid R0 bit fields [15:0, 25:16, 31:26] into three .u16 components in R0L, R0H, and R1L, and %tid maps to [R0L, R0H, R1L] in half words. 2D and 1D CTAs require no %tid initialization code.</p> <p>Preserve %tid for debugging.</p>
Examples	<pre>mov.b16 r0,%tid.0; // zero-extends tid.0 to r0 cvt.u32.u16 r2,%tid.2; // zero-extends tid.2 to r2</pre>

Table 63. Special Registers: %ntid

%ntid	Number of thread IDs per CTA.
Syntax	<pre>.sreg .v3 .u16 %ntid; // CTA shape vector .sreg .u16 %ntid.0, %ntid.1, %ntid.2; // CTA dimensions .sreg .u16 %ntid.x, %ntid.y, %ntid.z; // alternate component names</pre>
Description	<p>A predefined, read-only special register initialized with the number of thread ids in each CTA dimension. CTA dimensions are non-zero. The total number of threads in a CTA is (%ntid.0 * %ntid.1 * %ntid.2).</p> <p>The CTA dimensions are initialized in the predefined variable %ntid. The value of each element of the vector is at least 1.</p> <p>%ntid.1 == %ntid.2 == 1 in 1D CTAs. %ntid.2 == 1 in 2D CTAs.</p>
Notes	
Examples	<pre>mov.b16 r0,%tid.0; mov.b16 h1,%tid.1; mov.u16 h2,%ntid.0; mad.u16 r0,h1,h2,r0; // r0 = unified tid for 2D CTA</pre>

Table 64. Special Registers: %ctaid

%ctaid	CTA id within a grid.
Syntax	<pre>.sreg .v3 .u16 %ctaid; // CTA id vector .sreg .u16 %ctaid.0, %ctaid.1, %ctaid.2; // CTA id components .sreg .u16 %ctaid.x, %ctaid.y, %ctaid.z; // alternate component names</pre>
Description	<p>A predefined, read-only special register initialized with the CTA id within the CTA grid. %ctaid is a 1D, 2D, or 3D vector, depending on the shape and rank of the CTA grid. The value of each element of the vector is >= 0 and < 65535.</p> <p>It is guaranteed that:</p> <pre>0 <= %ctaid.0 < %nctaid.0 0 <= %ctaid.1 < %nctaid.1 0 <= %ctaid.2 < %nctaid.2</pre>
Notes	The G80 translator maps ctaid.0 to grid parameters g[6].u16 , ctaid.1 to g[7].u16 , and ctaid.2 to user parameter g[8].u16 .
Examples	<pre>mov.u32 %r1,%ctaid.1;</pre>

Table 65. Special Registers: %nctaid

%nctaid	Number of CTA ids per grid.
Syntax	<pre>.sreg .v3 .u16 %nctaid // Grid shape vector .sreg .u16 %nctaid.0, %nctaid.1, %nctaid.2; // Grid dimensions .sreg .u16 %nctaid.x, %nctaid.y, %nctaid.z; // alternate component names</pre>
Description	<p>A predefined, read-only special register initialized with the number of CTAs in each grid dimension. %nctaid is a 1D, 2D, or 3D vector, depending on the shape and rank of the CTA grid.</p> <p>The size of the grid of CTAs is stored in the predefined special register %nctaid. It is a 3D vector, and each member has a value of at least 1.</p> <p>It is guaranteed that:</p> $1 \leq \text{nctaid}.* < 65,536$
Notes	The G80 translator maps nctaid.0 to grid parameters g[4].u16, nctaid.1 to g[5].u16, and nctaid.2 to user parameter g[9].u16
Examples	<pre>mov.u32 r1,%nctaid;</pre>

Table 66. Special Registers: %gridid

%gridid	Grid ID.
Syntax	<pre>.sreg .u16 %gridid; // initialized when the grid is launched</pre>
Description	<p>A predefined, read-only special register initialized with the per-grid temporal grid ID number. This is used by debuggers to distinguish CTAs within concurrent (small) CTA grids.</p> <p>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.</p>
Notes	<p>The driver assigns a counting sequential gridid to each grid launched.</p> <p>The G80 translator maps gridid to grid parameter g[0].u16, “flags”.</p>
Examples	<pre>mov.u32 r1,%gridid;</pre>

Table 67. Special Registers: %clock

%clock	A predefined, read-only 32-bit unsigned cycle counter.
Syntax	
Description	Special register %clock is an unsigned 32-bit read-only cycle counter that wraps silently.
Notes	
Examples	<pre>mov.u32 r1,%clock;</pre>

This page is blank.

Chapter 9.

Directives

9.1. Specifying CTAs and Functions

Directives exist for specifying CTA entry points, the default number of threads in a CTA, functions, and other things. Some can be overridden later on.

Table 68. Directives: `.entry`

.entry	Defines a CTA entry point name, CTA rank, and optional CTA dimensions.
Syntax	<pre>.entry name [NTID.2][NTID.1][NTID.0]; // 3D CTA: RANKTID = 3. .entry name [NTID.1][NTID.0]; // 2D CTA: RANKTID = 2. .entry name [NTID.0]; // 1D CTA: RANKTID = 1. .entry name [NTID.2][NTID.1][NTID.0] { <i>per-CTA naming scope</i> }</pre>
Description	<p>Specifies a CTA entry point and name. The number of bracket pairs specifies the CTA rank constant RANKTID to be 1, 2, or 3. Constant expressions within the bracket pairs define the CTA dimension constants NTID.0, NTID.1, and NTID.2. Omitted dimension values define their constant dimension as 0. Omitted bracket pairs define their constant dimension as 1.</p> <p>Empty bracket pairs have unspecified dimensions that vary at run time, as specified by <code>ntid.0</code>, <code>ntid.1</code>, or <code>ntid.2</code>.</p> <p>Optionally specify a per-CTA naming scope enclosed in <code>{ }</code> braces, for <code>.shared</code> and <code>.param</code> variable declarations. PTX appends an exit instruction following the code in the braces.</p> <p>PTX defines an anonymous 3D CTA entry point at the first instruction encountered outside of a <code>.entry</code> or <code>.func</code> block. PTX appends an exit instruction after the last instruction of an anonymous entry point.</p>
Semantics	<p>Specify the entry point for a CTA program. Defines the constants RANKTID, NTID.0, NTID.1, and NTID.2.</p> <p>At run time, the CTA parameters <code>ntid.0</code>, <code>ntid.1</code>, and <code>ntid.2</code> are initialized with the actual CTA dimensions. The programmer may use the constant dimensions rather than the runtime dimensions if the CTA is always invoked with the constant dimensions.</p>
Notes	CTA dimensions are positive integers; zero means a dimension is unknown until runtime. G80 limits the product of dimensions to 512.
Examples	<pre>.entry cta_fft[256]; // 1D CTA with max 256 threads. .entry filter[16][16] { code; ... } // 2D CTA</pre>

Table 69. Directives: .func

.func	Function definition.
Syntax	<pre>.func fname <i>function-body</i> .func fname (<i>param-list</i>) <i>function-body</i> .func (<i>ret-param</i>) fname (<i>param-list</i>) <i>function-body</i></pre>
Description	Defines a function, including input and return parameters and function body.
Semantics	<p>Specifies the entry point and parameter names for a function. The parameter lists bind register names in the caller's namespace to register names in the callee namespace.</p> <p>The implementation of parameter passing is left to the optimizing translator, which may use a combination of registers and stack locations to pass parameters. In the current ptx release, parameters are passed through statically allocated ptx registers; i.e., there is no support for recursive calls.</p>
Notes	<p>The input and return parameters are enclosed in parentheses. Parameters must be base types in the register space. Parameter passing is call-by-value.</p> <p>A .func directive with no body may be used to declare a function prototype.</p>
Examples	<pre>.func (.reg .b32 rval) foo (.reg .b32 arg0, .reg .f64 arg1) { .reg .b32 localVar; ... use arg0; other code; mov.b32 rval,result; ret; } ... call (fooval), foo, (val0, val1); // return value in fooval ...</pre>

9.2. Debugging Directives

The following directives are needed to communicate Dwarf-format debug information. Details TBD.

Table 70. Debugging Directives: `.section`

.section	PTX section definition
Syntax	<code>.section <i>section_type</i>, <i>section_name</i></code>
Description	
Semantics	
Notes	
Examples	<code>.section .debug_info, "",@progbits</code>

Table 71. Debugging Directives: `.file`

.file	Source file information
Syntax	<code>.file <i>filename</i></code>
Description	
Semantics	
Notes	
Examples	

Table 72. Debugging Directives: `.loc`

.loc	Source file location
Syntax	<code>.loc <i>line_number</i></code>
Description	
Semantics	
Notes	
Examples	

Table 73. Debugging Directives: `.byte`

.byte	Byte data
Syntax	<code>.byte <i>data-list</i></code>
Description	Defines a sequence of data bytes.
Semantics	
Notes	
Examples	<code>.byte 0x7d, 0x01, 0x00, 0x00, 0x02, 0x00</code>

9.3. Other Directives

Table 74. Other Directives: `.extern`

.extern	External symbol declaration
Syntax	.extern <i>identifier</i>
Description	Declares identifier to be defined externally.
Semantics	
Notes	
Examples	<pre>.extern foo // variable foo is declared in another file .b32 foo;</pre>

Table 75. Other Directives: `.visible`

.visible	Visible (externally) symbol declaration
Syntax	.visible <i>identifier</i>
Description	Declares identifier to be externally visible.
Semantics	
Notes	
Examples	<pre>.visible foo // variable foo will be externally visible .b32 foo;</pre>

Table 76. Other Directives: `.version`

.version	PTX version number
Syntax	.version <i>major.minor</i> // <i>major, minor are integers</i>
Description	Specifies the PTX language version number. Increments to the major number indicate incompatible changes to PTX.
Semantics	<p>Indicates that this file must be compiled with tools having the same major version number and an equal or greater minor version number.</p> <p>Each ptx file must begin with a <code>.version</code> directive. Duplicate <code>.version</code> directives are allowed provided they match the original <code>.version</code> directive.</p>
Notes	Cuda Release 1.1 supports PTX ISA Versions 1.0 and 1.1.
Examples	<code>.version 1.1</code>

Table 77. Other Directives: `.target`

.target	Architecture and Platform target
Syntax	<p>.target <i>stringlist</i> <i>// comma separated list of target specifiers</i></p> <p><i>string</i> = { compute_10, compute_11, <i>// virtual arch targets</i> sm_10, sm_11, <i>// gpu target architectures</i> map_f64_to_f32 <i>// platform option</i> };</p>
Description	<p>Specifies the target architecture for which the current ptx code was generated.</p> <p>The target identifier strings are platform-specific.</p>
Semantics	<p>PTX features are checked against the specified target architecture, and an error is generated if an unsupported feature is used.</p> <p>The map_f64_to_f32 specifier indicates that all double-precision instructions will be mapped to single-precision regardless of the target architecture. This feature enables compilers for high-level languages such as Cuda to compile programs containing type double.</p> <p>Each PTX file must begin with a .version directive, immediately followed by a .target directive. Duplicate .target directives are allowed provided they match the original .target directive.</p>
Notes	
Examples	<pre>.target sm_11 // baseline target architecture // allow .f64 ops, but map them to .f32 in the translator .target sm_11, map_f64_to_f32</pre>

This page is blank.

Chapter 10.

Release Notes

This section describes ISA and implementation changes between PTX ISA Version 1.0 and Version 1.1. The changes may be summarized as (1) addition of new features, (2) removal of unimplemented features and instructions from the ISA, (3) better specification of rounding modifiers, and (4) better specification of saturation,

10.1. New Features

Instructions LD and ST now support a `.volatile` modifier. See the instruction descriptions in Chapter 7 for details.

10.2. Unimplemented Features Removed From ISA

PTX ISA version 1.0 contained a number of instructions and features that were unimplemented in the CUDA tools in release 1.0. Since these features were not implemented, their removal from PTX ISA version 1.1 does not create an incompatibility with any valid PTX version 1.0 code.

The vector instructions CROSS, DOT, MAG, and VRED have been removed from PTX. These instructions were unimplemented in version 1.0.

Instructions EXTRACT, INSERT, MEMBAR, and NOP were removed from the list of reserved PTX keywords shown in Table 2. The description of MEMBAR was removed from Chapter 7. These instructions were unimplemented in version 1.0.

Support for `.f64` type in SIN, COS, LG2, and EX2 has been removed from the ISA. These were unimplemented in version 1.0.

ATOM.`{cas,exch}` operations have been restricted to bitsize types. ATOM was unimplemented in PTX version 1.0.

10.3. Changes to Rounding Modifiers

PTX 1.0 did not fully specify rounding behavior for all instructions, nor did it define a default round behavior in cases where such defaults exist.

Rounding behavior not fully specified in PTX version 1.0 has been defined in version 1.1, with the following changes noted as errata for version 1.0:

- Instructions ADD, SUB, and MUL have round-to-nearest documented as their default rounding behavior.
- Instructions MAD no longer supports a rounding modifier.
- SAD, and DIV no longer support a rounding modifier. For double-precision, DIV implements round-to-nearest-even by default.
- Rounding modifiers are now required in some cases and illegal in other cases for the CVT instruction (see description). Hand-written version 1.0 PTX code may exist that violates these new restrictions.

10.4. Changes to Saturation

Saturation support has been removed from a number of instructions. None of these cases were used by the CUDA 1.0 compiler, and many were not implemented. These restrictions are compatible with PTX 1.0 code generated by the CUDA compiler tools.

- Integer saturation has been removed from instructions MUL, MUL24, MAD.wide, MAD.lo, MAD24.lo, SAD, DIV, and REM no longer support saturation.
- The CVT instruction supports saturation for both signed and unsigned integer types.

10.5. Unimplemented Features in Version 1.1

In Release 1.1 of the PTX ISA Version 1,1, a number of features are not supported. This section summarizes the unsupported features.

10.5.1. Syntax restrictions

Predicate constant immediates are not supported.

Constant expressions are not supported.

10.5.2. State Spaces

Declarations and instructions using `.surf` space are not supported.

The constant space is restricted to a single bank. This may be written as `.const` or `.const[0]`.

10.5.3. Variables and Operands

Vector declarations, initialization, and conversions are not supported.

Vector operands are not generally supported. The LD, ST, and TEX instructions do support limited use of vector operands written using the tuple notation.

10.5.4. Instructions

The FRC instruction is deprecated and will be removed in the next release.

See individual instruction descriptions in Section 7.7 for restrictions of the current release.

10.6. Summary of Instruction Changes

The following table summarizes changes to instructions in this release.

Table 78. Summary of Instruction Changes in Version 1.1

Instruction	Implementation Change
ADD	Default rounding of .rn documented.
SUB	Default rounding of .rn documented.
MUL	Integer saturation removed from parser. Default rounding of .rn documented.
MUL24	Integer saturation removed from parser.
MAD	Integer saturation removed from .wide and .lo modes. Rounding removed.
MAD24	Integer saturation removed from .lo mode.
SAD	Saturation removed (both int and float); rounding removed.
DIV	Integer saturation removed; rounding modifier removed. Document that DIV rounds to nearest even.
CVT	Rounding modes required when not illegal. See instruction description for details. Saturation extended to unsigned integer types.
LD, ST	Added .volatile modifier.
SET, SETP	Allow lt, le, ge, gt comparison operators to be used with unsigned integers.
CROSS, DOT, MAG, VRED	Removed. These were unimplemented in PTX 1.0.
SIN, COS, LG2, EX2	Remove .f64. This was unimplemented in PTX 1.0.
FRC	FRC is deprecated and will not be supported in the next PTX release.
ATOM	ATOM.{cas,exch} restricted to bitsize types. ATOM was not implemented in PTX 1.0.
EXTRACT, INSERT, MEMBAR, NOP	Removed keywords and descriptions for unimplemented instructions.



Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

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

Copyright

© 2007 NVIDIA Corporation. All rights reserved.



NVIDIA.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com