NVIDA GPU Instruction Set Architecture

- The instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set. PTX (Parallel Thread Execution) provides a stable instruction set for compilers as well as compatibility across generations of GPUs. The hardware instruction set is hidden from the programmer. PTX instructions describe the operations on a single CUDA thread, and usually map one-to-one with hardware instructions, but one PTX can expand to many machine instructions, and vice versa.

- PTX uses virtual registers, so the compiler figures out how many physical vector registers a SIMD thread needs, and then an optimizer divides the available register storage between the SIMD threads. This optimizer also eliminates dead code, folds instructions together, and calculates places where branches might diverge and places where diverged paths could converge.
- The format of a PTX instruction is

  opcode.type d, a, b, c;

  where d is the destination operand; a, b, and c are source operands;
  the operation type is one of the following:

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	. b8, b16, . b32, b64
Unsigned integer 8, 16, 32, and 64 bits	U8, . U16,U32, u64
Signed integer 8, 16, 32, and 64 bits	. S8, . S16, . S32, S64
Floating Point 16, 32, and 64 bits	.J16, J32, J64

- Source operands are 32-bit or 64-bit registers or a constant value. Destinations are registers, except for store instructions.

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a – b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special Function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential
Logical	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^ b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
Memory Access	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b,	atomic { d = *a; *a = cop(*a, b); }	atomic read-modify-write operation
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Control Flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

Fig: **PTX GPU Thread Instruction set.**

Conditional Branching in GPUs

- At the PTX assembler level, control flow of one CUDA thread is described by the PTX instructions branch, call, return, and exit, plus individual per-thread-lane predication of each instruction, specified by the programmer with per-thread-lane 1-bit predicate registers. The PTX assembler analyzes the PTX branch graph and optimizes it to the fastest GPU hardware instruction sequence.

- At the GPU hardware instruction level, control flow includes branch, jump, jump indexed, call, call indexed, return, exit, and special instructions that manage the branch synchronization stack. GPU hardware provides each SIMD thread with its own stack; a stack entry contains an identifier token, a target instruction address, and a target thread-active mask. There are GPU special instructions that push stack entries for a SIMD thread and special instructions and instruction markers that pop a stack entry. GPU hardware instructions also have individual per-lane predication (enable/disable), specified with a 1-bit predicate register for each lane.
- The PTX assembler typically optimizes a simple outer-level IF/THEN/ELSE statement coded with PTX branch instructions to just predicated GPU instructions, without any GPU branch instructions. The PTX assembler identifies loop branches and generates GPU branch instructions that branch to the top of the loop, along with special stack instructions to handle individual lanes breaking out of the loop and converging the SIMD Lanes when all lanes have completed the loop. GPU indexed jump and indexed call instructions push entries on the stack so that when all lanes complete the switch statement or function call the SIMD thread converges.
- A GPU set predicate instruction (setp in the figure above) evaluates the conditional part of the IF statement. The PTX branch instruction then depends on that predicate. If the PTX assembler generates predicated instructions with no GPU branch instructions, it uses a per-lane predicate register to enable or disable each SIMD Lane for each instruction.

- The SIMD instructions in the threads inside the THEN part of the IF statement broadcast operations to all the SIMD Lanes. Those lanes with the predicate set to one perform the operation and store the result, and the other SIMD Lanes don't perform an operation or store a result. For the ELSE statement, the instructions use the complement of the predicate (relative to the THEN statement), so the SIMD Lanes that were idle now perform the operation and store the result while their formerly active siblings don't. At the end of the ELSE statement, the instructions are unpredicated so the original computation can proceed. Thus, for equal length paths, an IF-THEN-ELSE operates at 50% efficiency.
- IF statements can be nested, hence the use of a stack, and the PTX assembler typically generates a mix of predicated instructions and GPU branch and special synchronization instructions for complex control flow. the PTX assembler sets a "branch synchronization" marker on appropriate conditional branch instructions that pushes the current active mask on a stack inside each SIMD thread. A branch synchronization marker pops the diverged branch entry and flips the mask bits before the ELSE portion. At the end of the IF statement, the PTX assembler adds another branch synchronization marker that pops the prior active mask off the stack into the current active mask.
- If all the mask bits are set to one, then the branch instruction at the end of the THEN skips over the instructions in the ELSE part. There is a similar optimization for the THEN part in case all the mask bits are zero, as the conditional branch jumps over the THEN instructions.
- The code for a conditional statement

```
if      (X[i]  !=0)
        X[i]= X[i]- Y[i];
else   X[i]= Z[i];
```

- This IF statement could compile to the following PTX instructions (assuming that R8 already has the scaled thread ID), with *Push, *Comp, *Pop indicating the branch synchronization markers inserted by the PTX assembler that push the old mask, complement the current mask, and pop to restore the old mask:

```
    ld.global.f64 RD0, [X+R8]       ; RD0 = X[i]
    setp.neq.s32 P1, RD0, #0        ; P1 is predicate register 1
    @!P1, bra ELSE1, *Push          ; Push old mask, set new mask bits
                                    ; if P1 false, go to ELSE1
    ld.global.f64 RD2, [Y+R8]       ; RD2 = Y[i]
    sub.f64 RD0, RD0, RD2           ; Difference in RD0
    st.global.f64 [X+R8], RD0       ; X[i] = RD0
    @P1, bra ENDIF1, *Comp          ; complement mask bits
                                    ; if P1 true, go to ENDIF1
ELSE1:  ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1: <next instruction>, *Pop    ; pop to restore old mask
```

- All instructions in the IF-THEN-ELSE statement are executed by a SIMD Processor. It's just that only some of the SIMD Lanes are enabled for the THEN instructions and some lanes for the ELSE instructions.