8.4 Multithreaded multiprocessor architecture

(Original section[1])

To address different market segments, GPUs implement scalable numbers of multiprocessors—in fact, GPUs are multiprocessors composed of multiprocessors. Furthermore, each multiprocessor is highly multithreaded to execute many fine-grained vertex and pixel shader threads efficiently. A quality basic GPU has two to four multiprocessors, while a gaming enthusiast's GPU or computing platform has dozens of them. This section looks at the architecture of one such multithreaded multiprocessor, a simplified version of the NVIDIA Tesla *streaming multiprocessor* (SM) described in COD Section B.7 (Real stuff: The NVIDIA GeForce 8800).

Why use a multiprocessor, rather than several independent processors? The parallelism within each multiprocessor provides localized high performance and supports extensive multithreading for the fine-grained parallel programming models described in COD Section B.3 (Programming GPUs). The individual threads of a thread block execute together within a multiprocessor to share data. The multithreaded multiprocessor design we describe here has eight scalar processor cores in a tightly coupled architecture, and executes up to 512 threads (the SM described in COD Section B.7 (Real stuff: The NVIDIA GeForce 8800) executes up to 768 threads). For area and power efficiency, the multiprocessor shares large complex units among the eight processor cores, including the instruction cache, the multithreaded instruction unit, and the shared memory RAM.

Massive multithreading

GPU processors are highly multithreaded to achieve several goals:

- Cover the latency of memory loads and texture fetches from DRAM
- Support fine-grained parallel graphics shader programming models
- Support fine-grained parallel computing programming models
- Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- Simplify the parallel programming model to writing a serial program for one thread

Memory and texture fetch latency can require hundreds of processor clocks, because GPUs typically have small streaming caches rather than large working-set caches like CPUs. A fetch request generally requires a full DRAM access latency plus interconnect and buffering latency. Multithreading helps cover the latency with useful computing—while one thread is waiting for a load or texture fetch to complete, the processor can execute another thread. The fine-grained parallel programming models provide literally thousands of independent threads that can keep many processors busy despite the long memory latency seen by individual threads.

A graphics vertex or pixel shader program is a program for a single thread that processes a vertex or a pixel. Similarly, a CUDA program is a C program for a single thread that computes a result. Graphics and computing programs instantiate many parallel threads to render complex images and compute large result arrays. To dynamically balance shifting vertex and pixel shader thread workloads, each multiprocessor concurrently executes multiple different thread programs and different types of shader programs.

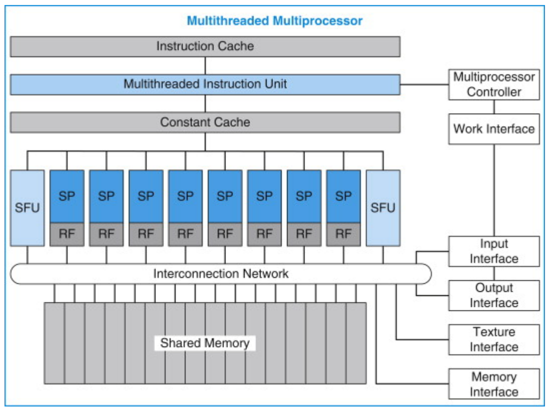
To support the independent vertex, primitive, and pixel programming model of graphics shading languages and the single-thread programming model of CUDA C/C++, each GPU thread has its own private registers, private per-thread memory, program counter, and thread execution state, and can execute an independent code path. To efficiently execute hundreds of concurrent lightweight threads, the GPU multiprocessor is hardware multithreaded—it manages and executes hundreds of concurrent threads in hardware without scheduling overhead. Concurrent threads within thread blocks can synchronize at a barrier with a single instruction. Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization efficiently support very fine-grained parallelism.

Multiprocessor architecture

A unified graphics and computing multiprocessor executes vertex, geometry, and pixel fragment shader programs, and parallel computing programs. As the figure below shows, the example multiprocessor consists of eight *scalar processor* (SP) cores each with a large multithreaded *register file* (RF), two *special function units* (SFUs), a multithreaded instruction unit, an instruction cache, a read-only constant cache, and a shared memory.

Figure 8.4.1: Multithreaded multiprocessor with eight scalar processor (SP) cores (COD Figure B.4.1).

The eight SP cores each have a large multithreaded *register file* (RF) and share an instruction cache, multithreaded instruction issue unit, constant cache, two *special function units* (SFUs), interconnection network, and a multibank shared memory.



The 16 KB shared memory holds graphics data buffers and shared computing data. CUDA variables declared as `__shared__` reside in the shared memory. To map the logical graphics pipeline workload through the multiprocessor multiple times, as shown in COD Section B.2 (GPU system architectures), vertex, geometry, and pixel threads have independent input and output buffers, and workloads arrive and depart independently of thread execution.

Each SP core contains scalar integer and floating-point arithmetic units that execute most instructions. The SP is hardware multithreaded, supporting up to 64 threads. Each pipelined SP core executes one scalar instruction per thread per clock, which ranges from 1.2 GHz to 1.6 GHz in different GPU products. Each SP core has a large RF of 1024 general-purpose 32-bit registers, partitioned among its assigned threads. Programs declare their register demand, typically 16 to 64 scalar 32-bit registers per thread. The SP can concurrently run many threads that use a few registers or fewer threads that use more registers. The compiler optimizes register allocation to balance the cost of spilling registers versus the cost of fewer threads. Pixel shader programs often use 16 or fewer registers, enabling each SP to run up to 64 pixel shader threads to cover long-latency texture fetches. Compiled CUDA programs often need 32 registers per thread, limiting each SP to 32 threads, which limits such a kernel program to 256 threads per thread block on this example multiprocessor, rather than its maximum of 512 threads.

The pipelined SFUs execute thread instructions that compute special functions and interpolate pixel attributes from primitive vertex attributes. These instructions can execute concurrently with instructions on the SPs. The SFU is described later.

The multiprocessor executes texture fetch instructions on the texture unit via the texture interface, and uses the memory interface for external memory load, store, and atomic access instructions. These instructions can execute concurrently with instructions on the SPs. Shared memory access uses a low-latency interconnection network between the SP processors and the shared memory banks.

Single-instruction multiple-thread (SIMT)

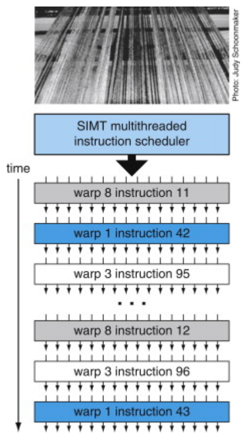
To manage and execute hundreds of threads running several different programs efficiently, the multiprocessor employs a *single-instruction multiple-thread (SIMT)* architecture. It creates, manages, schedules, and executes concurrent threads in groups of parallel threads called *warps*. The term *warp* originates from weaving, the first parallel thread technology. The photograph in the figure below shows a warp of parallel threads emerging from a loom. This example multiprocessor uses a SIMT warp size of 32 threads, executing four threads in each of the eight SP cores over four clocks. The Tesla SM multiprocessor described in COD Section B.7 (Real stuff: The NVIDIA GeForce 8800) also uses a warp size of 32 parallel threads, executing four threads per SP core for efficiency on plentiful pixel threads and computing threads. Thread blocks consist of one or more warps.

Single-instruction multiple-thread (SIMT): A processor architecture that applies one instruction to multiple independent threads in parallel.

Warp: The set of parallel threads that execute the same instruction together in a SIMT architecture.

Figure 8.4.2: SIMT multithreaded warp scheduling (COD Figure B.4.2).

The scheduler selects a ready warp and issues an instruction synchronously to the parallel threads composing the warp. Because warps are independent, the scheduler may select a different warp each time.



This example SIMT multiprocessor manages a pool of 16 warps, a total of 512 threads. Individual parallel threads composing a warp are the same type and start together at the same program address, but are otherwise free to branch and execute independently. At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute its next instruction, and then issues that instruction to the active threads of that warp. A SIMT instruction is broadcast synchronously to the active parallel threads of a warp; individual threads may be inactive due to independent branching or predication. In this multiprocessor, each SP scalar processor core executes an instruction for four individual threads of a warp using four clocks, reflecting the 4:1 ratio of warp threads to cores.

SIMT processor architecture is akin to *single-instruction multiple data* (SIMD) design, which applies one instruction to multiple data lanes, but differs in that SIMT applies one instruction to multiple independent threads in parallel, not just to multiple data lanes. An instruction for a SIMD processor controls a vector of multiple data lanes together, whereas an instruction for a SIMT processor controls an individual thread, and the SIMT instruction unit issues an instruction to a warp of independent parallel threads for efficiency. The SIMT processor finds data-level parallelism among threads at runtime, analogous to the way a superscalar processor finds instruction-level parallelism among instructions at runtime.

A SIMT processor realizes full efficiency and performance when all threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, execution serializes for each branch path taken, and when all paths complete, the threads converge to the same execution path. For equal length paths, a divergent if-else code block is 50% efficient. The multiprocessor uses a branch synchronization stack to manage independent threads that diverge and converge. Different warps execute independently at full speed regardless of whether they are executing common or disjoint code paths. As a result, SIMT GPUs are dramatically more efficient and flexible on branching code than earlier GPUs, as their warps are much narrower than the SIMD width of prior GPUs.

In contrast with SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for individual independent threads, as well as data-parallel code for many coordinated threads. For program correctness, the programmer can essentially ignore the SIMT execution attributes of warps; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional codes: cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance.

SIMT warp execution and divergence

The SIMT approach of scheduling independent warps is more flexible than the scheduling of previous GPU architectures. A warp comprises parallel threads of the same type: vertex, geometry, pixel, or compute. The basic unit of pixel fragment shader processing is the 2-by-2 pixel quad implemented as four pixel shader threads. The multiprocessor controller packs the pixel quads into a warp. It similarly groups vertices and primitives into warps, and packs computing threads into a warp. A thread block comprises one or more warps. The SIMT design shares the instruction fetch and issue unit efficiently across parallel threads of a warp, but requires a full warp of active threads to get full performance efficiency.

This unified multiprocessor schedules and executes multiple warp types concurrently, allowing it to concurrently execute vertex and pixel warps. Its warp scheduler operates at less than the processor clock rate, because there are four thread lanes per processor core. During each scheduling cycle, it selects a warp to execute a SIMT warp instruction, as shown in COD Figure B.4.2 (SIMT multithreaded warp scheduling). An issued warp-instruction executes as four sets of eight threads over four processor cycles of throughput. The processor pipeline uses several clocks of latency to complete each instruction. If the number of active warps times the clocks per warp exceeds the pipeline latency, the programmer can ignore the pipeline latency. For this multiprocessor, a round-robin schedule of eight warps has a period of 32 cycles between successive instructions for the same warp. If the program can keep 256 threads active per multiprocessor, instruction latencies up to 32 cycles can be hidden from an individual sequential thread. However, with few active warps, the processor pipeline depth becomes visible and may cause processors to stall.

A challenging design problem is implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types. The instruction scheduler must select a warp every four clocks to issue one instruction per clock per thread, equivalent to an IPC of 1.0 per processor core. Because warps are independent, the only dependences are among sequential instructions from the same warp. The scheduler uses a register dependency scoreboard to qualify warps whose active threads are ready to execute an instruction. It prioritizes all such ready warps and selects the highest priority one for issue. Prioritization must consider warp type, instruction type, and the desire to be fair to all active warps.

Managing threads and thread blocks

The multiprocessor controller and instruction unit manage threads and thread blocks. The controller accepts work requests and input data and arbitrates access to shared resources, including the texture unit, memory access path, and I/O paths. For graphics workloads, it creates and manages three types of graphics threads concurrently: vertex, geometry, and pixel. Each of the graphics work types has independent input and output paths. It accumulates and packs each of these input work types into SIMT warps of parallel threads executing the same thread program. It allocates a free warp, allocates registers for the warp threads, and starts warp execution in the multiprocessor. Every program declares its per-thread register demand; the controller starts a warp only when it can allocate the requested register count for the warp threads. When all the threads of the warp exit, the controller unpacks the results and frees the warp registers and resources.

The controller creates *cooperative thread arrays (CTAs)* which implement CUDA thread blocks as one or more warps of parallel threads. It creates a CTA when it can create all CTA warps and allocate all CTA resources. In addition to threads and registers, a CTA requires allocating shared memory and barriers. The program declares the required capacities, and the controller waits until it can allocate those amounts before launching the CTA. Then it creates CTA warps at the warp scheduling rate, so that a CTA program starts executing immediately at full multiprocessor performance. The controller monitors when all threads of a CTA have exited, and frees the CTA shared resources and its warp resources.

Cooperative thread array (CTA): A set of concurrent threads that executes the same thread program and may cooperate to compute a result. A GPU CTA implements a CUDA thread block.

Thread instructions

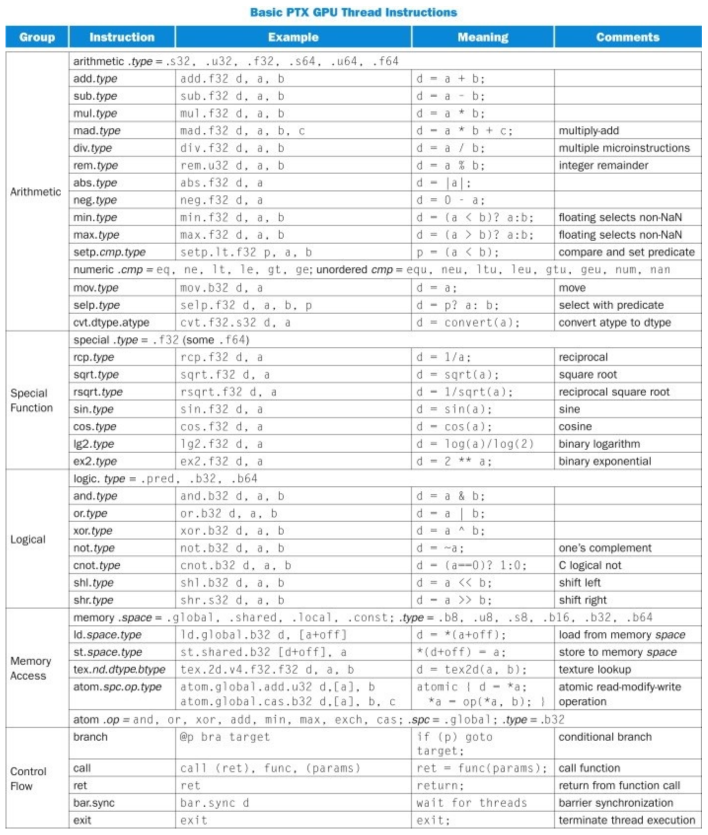
The SP thread processors execute scalar instructions for individual threads, unlike earlier GPU vector instruction architectures, which executed four-component vector instructions for each vertex or pixel shader program. Vertex programs generally compute (x, y, z, w) position vectors, while pixel shader programs compute (red, green, blue, alpha) color vectors. However, shader programs are becoming longer and more scalar, and it is increasingly difficult to fully occupy even two components of a legacy GPU four-component vector architecture. In effect, the SIMT architecture parallelizes across 32 independent pixel threads, rather than parallelizing the four vector components within a pixel. CUDA C/C++ programs have predominantly scalar code per thread. Previous GPUs employed vector packing (e.g., combining subvectors of work to gain efficiency) but that complicated the scheduling hardware as well as the compiler. Scalar instructions are simpler and compiler-friendly. Texture instructions remain vector-based, taking a source coordinate vector and returning a filtered color vector.

To support multiple GPUs with different binary microinstruction formats, high-level graphics and computing language compilers generate intermediate assembler-level instructions (e.g., Direct3D vector instructions or PTX scalar instructions), which are then optimized and translated to binary GPU microinstructions. The NVIDIA PTX (parallel thread execution) instruction set definition [2007] provides a stable target ISA for compilers, and provides compatibility over several generations of GPUs with evolving binary microinstruction-set architectures. The optimizer readily expands Direct3D vector instructions to multiple scalar binary microinstructions. PTX scalar instructions translate nearly one to one with scalar binary microinstructions, although some PTX instructions expand to multiple binary microinstructions, and multiple PTX instructions may fold into one binary microinstruction. Because the intermediate assembler-level instructions use virtual registers, the optimizer analyzes data dependencies and allocates real registers. The optimizer eliminates dead code, folds instructions together when feasible, and optimizes SIMT branch diverge and converge points.

Instruction set architecture (ISA)

The thread ISA described here is a simplified version of the Tesla architecture PTX ISA, a register-based scalar instruction set comprising floating-point, integer, logical, conversion, special functions, flow control, memory access, and texture operations. The figure below lists the basic PTX GPU thread instructions; see the NVIDIA PTX specification [2007] for details.

Figure 8.4.3: Basic PTX GPU thread instructions (COD Figure B.4.3).

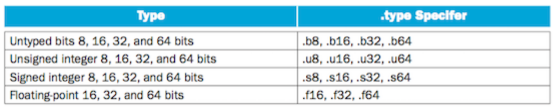


Basic PTX GPU Thread Instructions

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.dtype.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, c`	`atomic { d = *a; *a = op(*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

The instruction format is:

`opcode.type d, a, b, c;`
where `d` is the destination operand, `a`, `b`, `c` are source operands, and `.type` is one of:



Type	.type Specifer
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	.f16, .f32, .f64

Source operands are scalar 32-bit or 64-bit values in registers, an immediate value, or a constant; predicate operands are 1-bit Boolean values. Destinations are registers, except for store to memory. Instructions are predicated by prefixing them with `@p` or `@!p`, where `p` is a predicate register. Memory and texture instructions transfer scalars or vectors of two to four components, up to 128 bits in total. PTX instructions specify the behavior of one thread.

The PTX arithmetic instructions operate on 32-bit and 64-bit floating-point, signed integer, and unsigned integer types. Recent GPUs support 64-bit double-precision floating-point; see COD Section B.6 (Floating-point arithmetic). On current GPUs, PTX 64-bit integer and logical instructions are translated to two or more binary microinstructions that perform 32-bit operations. The GPU special function instructions are limited to 32-bit floating-point. The thread control flow instructions are conditional `branch`, function `call` and `return`, thread `exit`, and `bar.sync` (barrier synchronization). The conditional branch instruction `@p bra target` uses a predicate register `p` (or `!p`) previously set by a compare and set predicate `setp` instruction to determine whether the thread takes the branch or not. Other instructions can also be predicated on a predicate register being true or false.

Memory access instructions

The `tex` instruction fetches and filters texture samples from 1D, 2D, and 3D texture arrays in memory via the texture subsystem. Texture fetches generally use interpolated floating-point coordinates to address a texture. Once a graphics pixel shader thread computes its pixel fragment color, the raster operations processor blends it with the pixel color at its assigned (x, y) pixel position and writes the final color to memory.

To support computing and C/C++ language needs, the Tesla PTX ISA implements memory load/store instructions. It uses integer byte addressing with register plus offset address arithmetic to facilitate conventional compiler code optimizations. Memory load/store instructions are common in processors, but are a significant new capability in the Tesla architecture GPUs, as prior GPUs provided only the texture and pixel accesses required by the graphics APIs.

For computing, the load/store instructions access three read/write memory spaces that implement the corresponding CUDA memory spaces in COD Section B.3 (Programming GPUs):

- Local memory for per-thread private addressable temporary data (implemented in external DRAM)
- Shared memory for low-latency access to data shared by cooperating threads in the same CTA/thread block (implemented in on-chip SRAM)
- Global memory for large data sets shared by all threads of a computing application (implemented in external DRAM)

The memory load/store instructions `ld.global`, `st.global`, `ld.shared`, `st.shared`, `ld.local`, and `st.local` access the global, shared, and local memory spaces. Computing programs use the fast barrier synchronization instruction `bar.sync` to synchronize threads within a CTA/thread block that communicate with each other via shared and global memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread requests from the same SIMT warp together into a single memory block request when the addresses fall in the same block and meet alignment criteria. Coalescing memory requests provides a significant performance boost over separate requests from individual threads.

The multiprocessor's large thread count, together with support for many outstanding load requests, helps cover load-to-use latency for local and global memory implemented in external DRAM.

The latest Tesla architecture GPUs also provide efficient atomic memory operations on memory with the `atom.op.u32` instructions, including integer operations `add`, `min`, `max`, `and`, `or`, `xor`, `exchange`, and `cas` (compare-and-swap) operations, facilitating parallel reductions and parallel data structure management.

Barrier synchronization for thread communication

Fast barrier synchronization permits CUDA programs to communicate frequently via shared memory and global memory by simply calling `__syncthreads();` as part of each interthread communication step. The synchronization intrinsic function generates a single `bar.sync` instruction. However, implementing fast barrier synchronization among up to 512 threads per CUDA thread block is a challenge.

Grouping threads into SIMT warps of 32 threads reduces the synchronization difficulty by a factor of 32. Threads wait at a barrier in the SIMT thread scheduler so they do not consume any processor cycles while waiting. When a thread executes a `bar.sync` instruction, it increments the barrier's thread arrival counter and the scheduler marks the thread as waiting at the barrier. Once all the CTA threads arrive, the barrier counter matches the expected terminal count, and the scheduler releases all the threads waiting at the barrier and resumes executing threads.

Streaming processor (SP)

The multithreaded *streaming processor* (SP) core is the primary thread instruction processor in the multiprocessor. Its *register file* (RF) provides 1024 scalar 32-bit registers for up to 64 threads. It executes all the fundamental floating-point operations, including `add.f32`, `mul.f32`, `mad.f32` (floating multiply-add), `min.f32`, `max.f32`, and `setp.f32` (floating compare and set predicate). The floating-point add and multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including *not-a-number* (NaN) and infinity values. The SP core also implements all of the 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX instructions shown in COD Figure B.4.3 (Basic PTX GPU thread instructions).

The floating-point `add` and `mul` operations employ IEEE round-to-nearest-even as the default rounding mode. The `mad.f32` floating-point multiply-add operation performs a multiplication with truncation, followed by an addition with round-to-nearest-even. The SP flushes input denormal operands to sign-preserved-zero. Results that underflow the target output exponent range are flushed to sign-preserved-zero after rounding.

Special function unit (SFU)

Certain thread instructions can execute on the SFUs, concurrently with other thread instructions executing on the SPs. The SFU implements the special function instructions of COD Figure B.4.3 (Basic PTX GPU thread instructions), which compute 32-bit floating-point approximations to reciprocal, reciprocal square root, and key transcendental functions. It also implements 32-bit floating-point planar attribute interpolation for pixel shaders, providing accurate interpolation of attributes such as color, depth, and texture coordinates.

Each pipelined SFU generates one 32-bit floating-point special function result per cycle; the two SFUs per multiprocessor execute special function instructions at a quarter the simple instruction rate of the eight SPs. The SFUs also execute the `mul.f32` multiply instruction concurrently with the eight SPs, increasing the peak computation rate up to 50% for threads with a suitable instruction mixture.

For functional evaluation, the Tesla architecture SFU employs quadratic interpolation based on enhanced minimax approximations for approximating the reciprocal, reciprocal square-root, $\log_2 x$, $2x$, and sin/cos functions. The accuracy of the function estimates ranges from 22 to 24 mantissa bits. See COD Section B.6 (Floating-point arithmetic) for more details on SFU arithmetic.

Comparing with other multiprocessors

Compared with SIMD vector architectures such as x86 SSE, the SIMT multiprocessor can execute individual threads independently, rather than always executing them together in synchronous groups. SIMT hardware finds data parallelism among independent threads, whereas SIMD hardware requires the software to express data parallelism explicitly in each vector instruction. A SIMT machine executes a warp of 32 threads synchronously when the threads take the same execution path, yet can execute each thread independently when they diverge. The advantage is significant because SIMT programs and instructions simply describe the behavior of a single independent thread, rather than a SIMD data vector of four or more data lanes. Yet the SIMT multiprocessor has SIMD-like efficiency, spreading the area and cost of one instruction unit across the 32 threads of a warp and across the eight streaming processor cores. SIMT provides the performance of SIMD together with the productivity of multithreading, avoiding the need to explicitly code SIMD vectors for edge conditions and partial divergence.

The SIMT multiprocessor imposes little overhead because it is hardware multithreaded with hardware barrier synchronization. That allows graphics shaders and CUDA threads to express very fine-grained parallelism. Graphics and CUDA programs use threads to express fine-grained data parallelism in a per-thread program, rather than forcing the programmer to express it as SIMD vector instructions. It is simpler and more productive to develop scalar single-thread code than vector code, and the SIMT multiprocessor executes the code with SIMD-like efficiency.

Coupling eight streaming processor cores together closely into a multiprocessor and then implementing a scalable number of such multiprocessors makes a two-level multiprocessor composed of multiprocessors. The CUDA programming model exploits the two-level hierarchy by providing individual threads for fine-grained parallel computations, and by providing grids of thread blocks for coarse-grained parallel operations. The same thread program can provide both fine-grained and coarse-grained operations. In contrast, CPUs with SIMD vector instructions must use two different programming models to provide fine-grained and coarse-grained operations: coarse-grained parallel threads on different cores, and SIMD vector instructions for fine-grained data parallelism.

Multithreaded multiprocessor conclusion

The example GPU multiprocessor based on the Tesla architecture is highly multithreaded, executing a total of up to 512 lightweight threads concurrently to support fine-grained pixel shaders and CUDA threads. It uses a variation on SIMD architecture and multithreading called SIMT (*single-instruction multiple-thread*) to efficiently broadcast one instruction to a warp of 32 parallel threads, while permitting each thread to branch and execute independently. Each thread executes its instruction stream on one of the eight *streaming processor* (SP) cores, which are multithreaded up to 64 threads.

The PTX ISA is a register-based load/store scalar ISA that describes the execution of a single thread. Because PTX instructions are optimized and translated to binary microinstructions for a specific GPU, the hardware instructions can evolve rapidly without disrupting compilers and software tools that generate PTX instructions.

(*1) This section is in original form.