NVIDIA Tesla: A Unified Graphics and Computing Architecture

To enable flexible, programmable graphics and high-performance computing, NVIDIA has developed the Tesla scalable unified graphics and parallel computing architecture. Its scalable parallel array of processors is massively multithreaded and programmable in C or via graphics APIs.

Erik Lindholm
John Nickolls
Stuart Oberman
John Montrym
NVIDIA

The modern 3D graphics processing unit (GPU) has evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. Traditional graphics pipelines consist of separate programmable stages of vertex processors executing vertex shader programs and pixel fragment processors executing pixel shader programs. (Montrym and Moreton provide additional background on the traditional graphics processor architecture.[1])

NVIDIA's Tesla architecture, introduced in November 2006 in the GeForce 8800 GPU, unifies the vertex and pixel processors and extends them, enabling high-performance parallel computing applications written in the C language using the Compute Unified Device Architecture (CUDA[2–4]) parallel programming model and development tools. The Tesla unified graphics and computing architecture is available in a scalable family of GeForce 8-series GPUs and Quadro GPUs for laptops, desktops, workstations, and servers. It also provides the processing architecture for the Tesla GPU computing platforms introduced in 2007 for high-performance computing.

In this article, we discuss the requirements that drove the unified graphics and parallel computing processor architecture, describe the Tesla architecture, and how it is enabling widespread deployment of parallel computing and graphics applications.

The road to unification

The first GPU was the GeForce 256, introduced in 1999. It contained a fixed-function 32-bit floating-point vertex transform and lighting processor and a fixed-function integer pixel-fragment pipeline, which were programmed with OpenGL and the Microsoft DX7 API.[5] In 2001, the GeForce 3 introduced the first programmable vertex processor executing vertex shaders, along with a configurable 32-bit floating-point fragment pipeline, programmed with DX8[5] and OpenGL.[6] The Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel-fragment processor programmed with DX9 and OpenGL.[7,8] The GeForce FX added 32-bit floating-point pixel-fragment processors. The XBox 360 introduced an early unified GPU in 2005, allowing vertices and pixels to execute on the same processor.[9]

0272-1732/08/$20.00 © 2008 IEEE Published by the IEEE Computer Society.

Vertex processors operate on the vertices of primitives such as points, lines, and triangles. Typical operations include transforming coordinates into screen space, which are then fed to the setup unit and the rasterizer, and setting up lighting and texture parameters to be used by the pixel-fragment processors. Pixel-fragment processors operate on rasterizer output, which fills the interior of primitives, along with the interpolated parameters.

Vertex and pixel-fragment processors have evolved at different rates: Vertex processors were designed for low-latency, high-precision math operations, whereas pixel-fragment processors were optimized for high-latency, lower-precision texture filtering. Vertex processors have traditionally supported more-complex processing, so they became programmable first. For the last six years, the two processor types have been functionally converging as the result of a need for greater programming generality. However, the increased generality also increased the design complexity, area, and cost of developing two separate processors.

Because GPUs typically must process more pixels than vertices, pixel-fragment processors traditionally outnumber vertex processors by about three to one. However, typical workloads are not well balanced, leading to inefficiency. For example, with large triangles, the vertex processors are mostly idle, while the pixel processors are fully busy. With small triangles, the opposite is true. The addition of more-complex primitive processing in DX10 makes it much harder to select a fixed processor ratio.[10] All these factors influenced the decision to design a unified architecture.

A primary design objective for Tesla was to execute vertex and pixel-fragment shader programs on the same unified processor architecture. Unification would enable dynamic load balancing of varying vertex- and pixel-processing workloads and permit the introduction of new graphics shader stages, such as geometry shaders in DX10. It also let a single team focus on designing a fast and efficient processor and allowed the sharing of expensive hardware such as the texture units. The generality required of a unified processor opened the door to a completely new GPU parallel-computing capability. The downside of this generality was the difficulty of efficient load balancing between different shader types.

Other critical hardware design requirements were architectural scalability, performance, power, and area efficiency.

The Tesla architects developed the graphics feature set in coordination with the development of the Microsoft Direct3D DirectX 10 graphics API.[10] They developed the GPU's computing feature set in coordination with the development of the CUDA C parallel programming language, compiler, and development tools.

Tesla architecture

The Tesla architecture is based on a scalable processor array. Figure 1 shows a block diagram of a GeForce 8800 GPU with 128 streaming-processor (SP) cores organized as 16 streaming multiprocessors (SMs) in eight independent processing units called texture/processor clusters (TPCs). Work flows from top to bottom, starting at the host interface with the system PCI-Express bus. Because of its unified-processor design, the physical Tesla architecture doesn't resemble the logical order of graphics pipeline stages. However, we will use the logical graphics pipeline flow to explain the architecture.

At the highest level, the GPU's scalable streaming processor array (SPA) performs all the GPU's programmable calculations. The scalable memory system consists of external DRAM control and fixed-function raster operation processors (ROPs) that perform color and depth frame buffer operations directly on memory. An interconnection network carries computed pixel-fragment colors and depth values from the SPA to the ROPs. The network also routes texture memory read requests from the SPA to DRAM and read data from DRAM through a level-2 cache back to the SPA.

The remaining blocks in Figure 1 deliver input work to the SPA. The input assembler collects vertex work as directed by the input command stream. The vertex work distri-

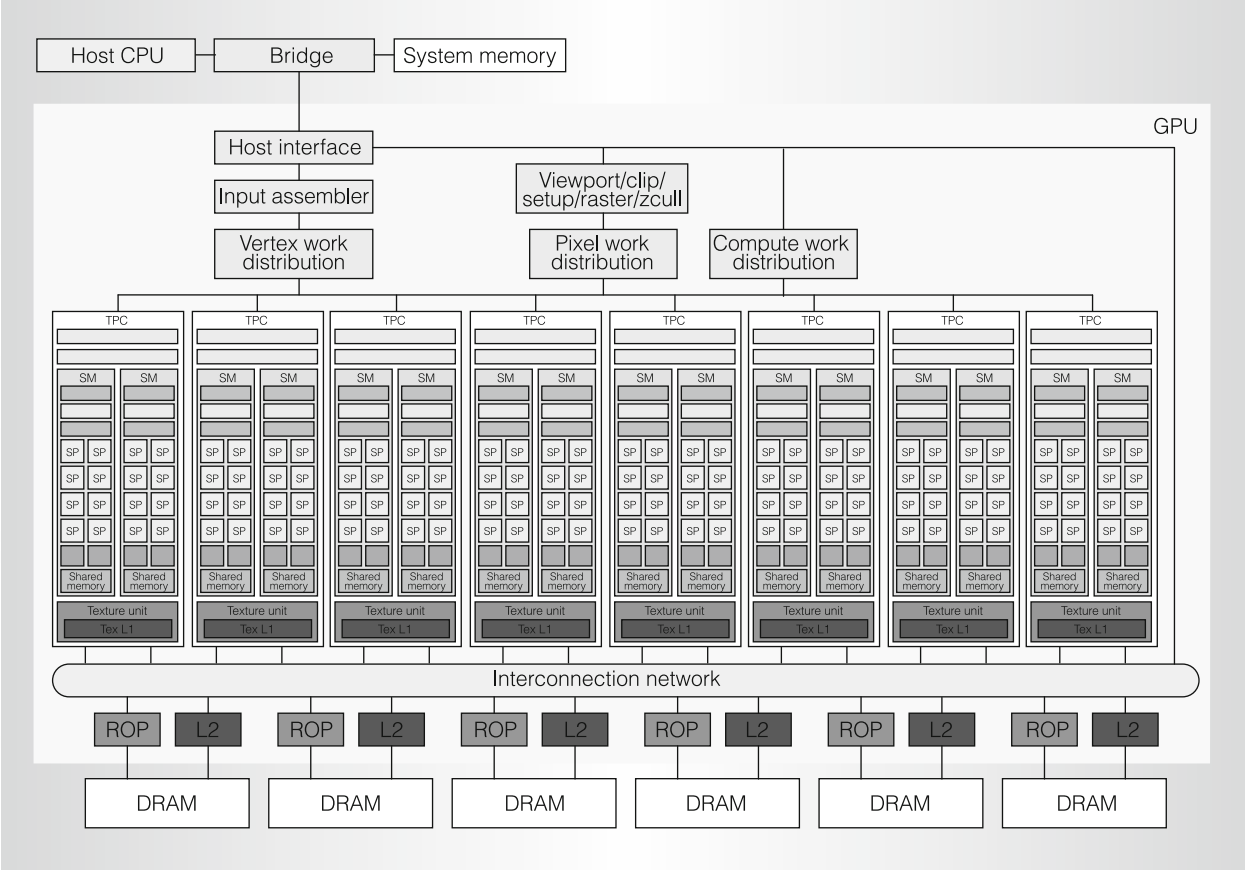

Figure 1. Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor.

bution block distributes vertex work packets to the various TPCs in the SPA. The TPCs execute vertex shader programs, and (if enabled) geometry shader programs. The resulting output data is written to on-chip buffers. These buffers then pass their results to the viewport/clip/setup/raster/zcull block to be rasterized into pixel fragments. The pixel work distribution unit distributes pixel fragments to the appropriate TPCs for pixel-fragment processing. Shaded pixel-fragments are sent across the interconnection network for processing by depth and color ROP units. The compute work distribution block dispatches compute thread arrays to the TPCs. The SPA accepts and processes work for multiple logical streams simultaneously. Multiple clock domains for GPU units, processors, DRAM, and other units allow independent power and performance optimizations.

Command processing

The GPU host interface unit communicates with the host CPU, responds to commands from the CPU, fetches data from system memory, checks command consistency, and performs context switching.

The input assembler collects geometric primitives (points, lines, triangles, line strips, and triangle strips) and fetches associated vertex input attribute data. It has peak rates of one primitive per clock and eight scalar attributes per clock at the GPU core clock, which is typically 600 MHz.

The work distribution units forward the input assembler's output stream to the array of processors, which execute vertex, geometry, and pixel shader programs, as well as computing programs. The vertex and compute work distribution units deliver work to processors in a round-robin scheme. Pixel

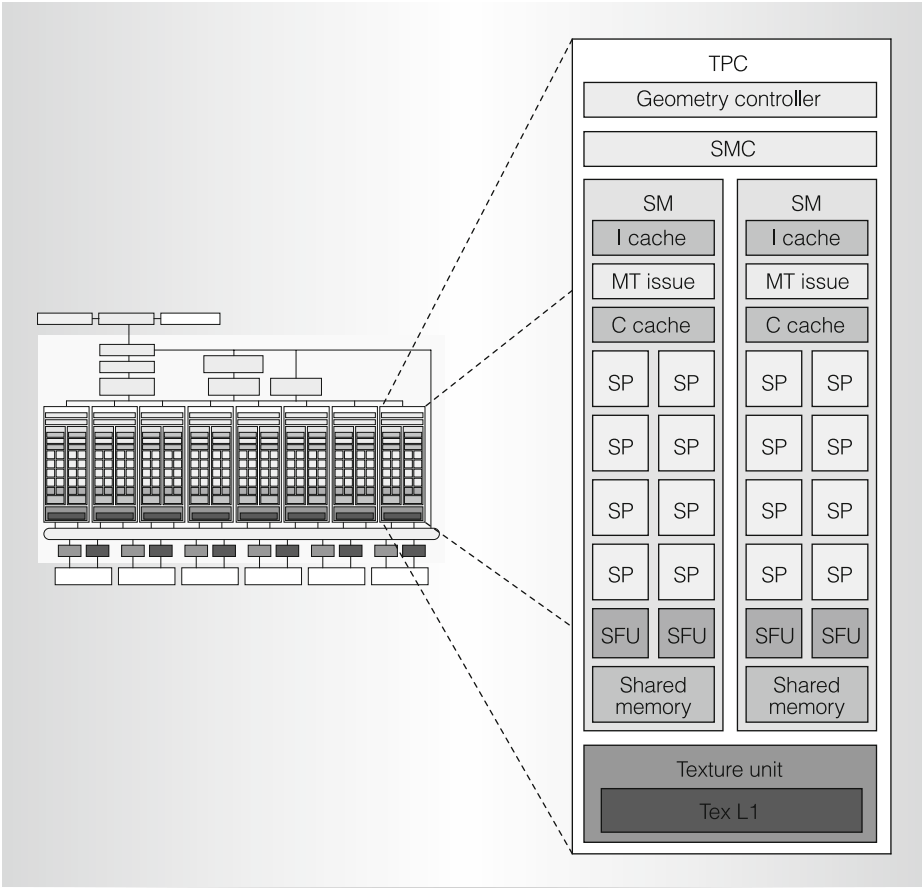

Figure 2. Texture/processor cluster (TPC).

work distribution is based on the pixel location.

Streaming processor array

The SPA executes graphics shader thread programs and GPU computing programs and provides thread control and management. Each TPC in the SPA roughly corresponds to a quad-pixel unit in previous architectures.[1] The number of TPCs determines a GPU's programmable processing performance and scales from one TPC in a small GPU to eight or more TPCs in high-performance GPUs.

Texture/processor cluster

As Figure 2 shows, each TPC contains a geometry controller, an SM controller (SMC), two streaming multiprocessors (SMs), and a texture unit. Figure 3 expands each SM to show its eight SP cores. To balance the expected ratio of math operations to texture operations, one texture unit serves two SMs. This architectural ratio can vary as needed.

Geometry controller

The geometry controller maps the logical graphics vertex pipeline into recirculation on the physical SMs by directing all primitive and vertex attribute and topology flow in the TPC. It manages dedicated on-chip input and output vertex attribute storage and forwards contents as required.

DX10 has two stages dealing with vertex and primitive processing: the vertex shader and the geometry shader. The vertex shader processes one vertex's attributes independently of other vertices. Typical operations are position space transforms and color and texture coordinate generation. The geometry shader follows the vertex shader and deals with a whole primitive and its vertices. Typical operations are edge extrusion for

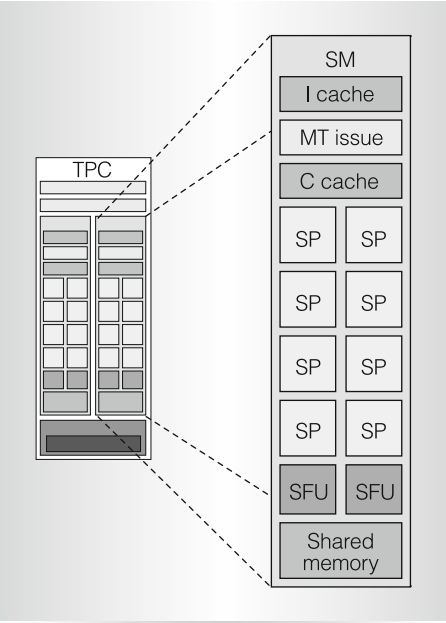

Figure 3. Streaming multiprocessor (SM).

stencil shadow generation and cube map texture generation. Geometry shader output primitives go to later stages for clipping, viewport transformation, and rasterization into pixel fragments.

Streaming multiprocessor

The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs. As Figure 3 shows, the SM consists of eight streaming processor (SP) cores, two special-function units (SFUs), a multithreaded instruction fetch and issue unit (MT Issue), an instruction cache, a read-only constant cache, and a 16-Kbyte read/write shared memory.

The shared memory holds graphics input buffers or shared data for parallel computing. To pipeline graphics workloads through the SM, vertex, geometry, and pixel threads have independent input and output buffers. Workloads can arrive and depart independently of thread execution. Geometry threads, which generate variable amounts of output per thread, use separate output buffers.

Each SP core contains a scalar multiply-add (MAD) unit, giving the SM eight MAD units. The SM uses its two SFU units for transcendental functions and attribute interpolation—the interpolation of pixel attributes from vertex attributes defining a primitive. Each SFU also contains four floating-point multipliers. The SM uses the TPC texture unit as a third execution unit and uses the SMC and ROP units to implement external memory load, store, and atomic accesses. A low-latency interconnect network between the SPs and the shared-memory banks provides shared-memory access.

The GeForce 8800 Ultra clocks the SPs and SFU units at 1.5 GHz, for a peak of 36 Gflops per SM. To optimize power and area efficiency, some SM non-data-path units operate at half the SP clock rate.

SM multithreading. A graphics vertex or pixel shader is a program for a single thread that describes how to process a vertex or a pixel. Similarly, a CUDA kernel is a C program for a single thread that describes how one thread computes a result. Graphics and computing applications instantiate many parallel threads to render complex images and compute large result arrays. To dynamically balance shifting vertex and pixel shader thread workloads, the unified SM concurrently executes different thread programs and different types of shader programs.

To efficiently execute hundreds of threads in parallel while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead.

To support the independent vertex, primitive, pixel, and thread programming model of graphics shading languages and the CUDA C/C++ language, each SM thread has its own thread execution state and can execute an independent code path. Concurrent threads of computing programs can synchronize at a barrier with a single SM instruction. Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization support very fine-grained parallelism efficiently.

Single-instruction, multiple-thread. To manage and execute hundreds of threads running

several different programs efficiently, the Tesla SM uses a new processor architecture we call single-instruction, multiple-thread (SIMT). The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. The term *warp* originates from weaving, the first parallel-thread technology. Figure 4 illustrates SIMT scheduling. The SIMT warp size of 32 parallel threads provides efficiency on plentiful fine-grained pixel threads and computing threads.

Each SM manages a pool of 24 warps, with a total of 768 threads. Individual threads composing a SIMT warp are of the same type and start together at the same program address, but they 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 and issues the next instruction to that warp's active threads. A SIMT instruction is broadcast synchronously to a warp's active parallel threads; individual threads can be inactive due to independent branching or predication.

The SM maps the warp threads to the SP cores, and each thread executes independently with its own instruction address and register state. A SIMT processor realizes full efficiency and performance when all 32 threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads reconverge to the original execution path. The SM uses a branch synchronization stack to manage independent threads that diverge and converge. Branch divergence only occurs within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. As a result, Tesla architecture GPUs are dramatically more efficient and flexible on branching code than previous generation GPUs, as their 32-thread warps are much narrower than the SIMD width of prior GPUs.[1]

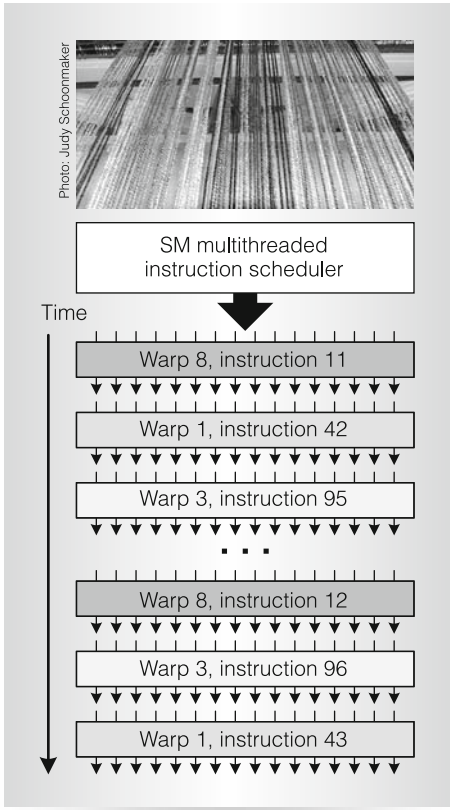

Figure 4. Single-instruction, multiple-thread (SIMT) warp scheduling.

SIMT architecture is similar to single-instruction, multiple-data (SIMD) design, which applies one instruction to multiple data lanes. The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes. A SIMD instruction controls a vector of multiple data lanes together and exposes the vector width to the software, whereas a SIMT instruction controls the execution and branching behavior of one thread.

In contrast to SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for independent threads as well as data-parallel code for coordinated threads. For program correctness, programmers can essentially ignore SIMT execution attributes such as warps; however, they can achieve substantial performance improvements by writing code that seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in

traditional codes: Programmers can safely ignore cache line size when designing for correctness but must consider it in the code structure when designing for peak performance. SIMD vector architectures, on the other hand, require the software to manually coalesce loads into vectors and to manually manage divergence.

SIMT warp scheduling. The SIMT approach of scheduling independent warps is simpler than previous GPU architectures' complex scheduling. A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The basic unit of pixel-fragment shader processing is the 2 $\times$ 2 pixel quad. The SM controller groups eight pixel quads into a warp of 32 threads. It similarly groups vertices and primitives into warps and packs 32 computing threads into a warp. The SIMT design shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency.

As a unified graphics processor, the SM schedules and executes multiple warp types concurrently—for example, concurrently executing vertex and pixel warps. The SM warp scheduler operates at half the 1.5-GHz processor clock rate. At each cycle, it selects one of the 24 warps to execute a SIMT warp instruction, as Figure 4 shows. An issued warp instruction executes as two sets of 16 threads over four processor cycles. The SP cores and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied.

Implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types was a challenging design problem. A scoreboard qualifies each warp for issue each cycle. The instruction scheduler prioritizes all ready warps and selects the one with highest priority for issue. Prioritization considers warp type, instruction type, and "fairness" to all warps executing in the SM.

SM instructions. The Tesla SM executes scalar instructions, unlike previous GPU vector instruction architectures. Shader programs are becoming longer and more scalar, and it is increasingly difficult to fully occupy even two components of the prior four-component vector architecture. Previous architectures employed vector packing—combining sub-vectors 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.

High-level graphics and computing-language compilers generate intermediate instructions, such as DX10 vector or PTX scalar instructions,[10,2] which are then optimized and translated to binary GPU instructions. The optimizer readily expands DX10 vector instructions to multiple Tesla SM scalar instructions. PTX scalar instructions optimize to Tesla SM scalar instructions about one to one. PTX provides a stable target ISA for compilers and provides compatibility over several generations of GPUs with evolving binary instruction set architectures. Because the intermediate languages use virtual registers, the optimizer analyzes data dependencies and allocates real registers. It eliminates dead code, folds instructions together when feasible, and optimizes SIMT branch divergence and convergence points.

Instruction set architecture. The Tesla SM has a register-based instruction set including floating-point, integer, bit, conversion, transcendental, flow control, memory load/store, and texture operations.

Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers. Floating-point instructions provide source operand modifiers for negation and absolute value. Transcendental function instructions include cosine, sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root. Attribute interpolation instructions provide efficient generation of pixel attributes. Bitwise operators include shift left, shift right, logic operators, and move. Control

flow includes branch, call, return, trap, and barrier synchronization.

The floating-point and integer instructions can also set per-thread status flags for zero, negative, carry, and overflow, which the thread program can use for conditional branching.

Memory access instructions. The texture instruction fetches and filters texture samples from memory via the texture unit. The ROP unit writes pixel-fragment output to memory.

To support computing and C/C++ language needs, the Tesla SM implements memory load/store instructions in addition to graphics texture fetch and pixel output. Memory load/store instructions use integer byte addressing with register-plus-offset address arithmetic to facilitate conventional compiler code optimizations.

For computing, the load/store instructions access three read/write memory spaces:

- *local* memory for per-thread, private, temporary data (implemented in external DRAM);
- *shared* memory for low-latency access to data shared by cooperating threads in the same SM; and
- *global* memory for data shared by all threads of a computing application (implemented in external DRAM).

The memory instructions load-global, store-global, load-shared, store-shared, load-local, and store-local access global, shared, and local memory. Computing programs use the fast barrier synchronization instruction to synchronize threads within the SM 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 accesses from the same warp into fewer memory block accesses. The addresses must fall in the same block and meet alignment criteria. Coalescing memory requests boosts performance significantly over separate requests. The 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 provide efficient atomic memory operations, including integer add, minimum, maximum, logic operators, swap, and compare-and-swap operations. Atomic operations facilitate parallel reductions and parallel data structure management.

Streaming processor. The SP core is the primary thread processor in the SM. It performs the fundamental floating-point operations, including add, multiply, and multiply-add. It also implements a wide variety of integer, comparison, and conversion operations. 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 unit is fully pipelined, and latency is optimized to balance delay and area.

The add and multiply operations use IEEE round-to-nearest-even as the default rounding mode. The multiply-add operation performs a multiplication with truncation, followed by an add with round-to-nearest-even. The SP flushes denormal source operands to sign-preserved zero and flushes results that underflow the target output exponent range to sign-preserved zero after rounding.

Special-function unit. The SFU supports computation of both transcendental functions and planar attribute interpolation.[11] A traditional vertex or pixel shader design contains a functional unit to compute transcendental functions. Pixels also need an attribute-interpolating unit to compute the per-pixel attribute values at the pixel's x, y location, given the attribute values at the primitive's vertices.

For functional evaluation, we use quadratic interpolation based on enhanced minimax approximations to approximate the reciprocal, reciprocal square root, $\log_2 x$, 2^x, and sin/cos functions. Table 1 shows the accuracy of the function estimates. The SFU unit generates one 32-bit floating point result per cycle.

Table 1. Function approximation statistics.

Function	Input interval	Accuracy (good bits)	ULP* error	% exactly rounded	Monotonic
$1/x$	[1, 2)	24.02	0.98	87	Yes
1/sqrt(x)	[1, 4)	23.40	1.52	78	Yes
2^x	[0, 1)	22.51	1.41	74	Yes
$\log_2 x$	[1, 2)	22.57	N/A**	N/A	Yes
sin/cos	[0, π/2)	22.47	N/A	N/A	No

* ULP: unit-in-the-last-place.
** N/A: not applicable.

The SFU also supports attribute interpolation, to enable accurate interpolation of attributes such as color, depth, and texture coordinates. The SFU must interpolate these attributes in the (x, y) screen space to determine the values of the attributes at each pixel location. We express the value of a given attribute U in an (x, y) plane in plane equations of the following form:

$$U(x, y) = (A_U \times x + B_U \times y + C_U)/(A_W \times x + B_W \times y + C_W)$$

where A, B, and C are interpolation parameters associated with each attribute U, and W is related to the distance of the pixel from the viewer for perspective projection. The attribute interpolation hardware in the SFU is fully pipelined, and it can interpolate four samples per cycle.

In a shader program, the SFU can generate perspective-corrected attributes as follows:

- Interpolate $1/W$, and invert to form W.
- Interpolate U/W.
- Multiply U/W by W to form perspective-correct U.

SM controller. The SMC controls multiple SMs, arbitrating the shared texture unit, load/store path, and I/O path. The SMC serves three graphics workloads simultaneously: vertex, geometry, and pixel. It packs each of these input types into the warp width, initiating shader processing, and unpacks the results.

Each input type has independent I/O paths, but the SMC is responsible for load balancing among them. The SMC supports static and dynamic load balancing based on driver-recommended allocations, current allocations, and relative difficulty of additional resource allocation. Load balancing of the workloads was one of the more challenging design problems due to its impact on overall SPA efficiency.

Texture unit

The texture unit processes one group of four threads (vertex, geometry, pixel, or compute) per cycle. Texture instruction sources are texture coordinates, and the outputs are filtered samples, typically a four-component (RGBA) color. Texture is a separate unit external to the SM connected via the SMC. The issuing SM thread can continue execution until a data dependency stall.

Each texture unit has four texture address generators and eight filter units, for a peak GeForce 8800 Ultra rate of 38.4 gigabilerps/s (a bilerp is a bilinear interpolation of four samples). Each unit supports full-speed 2:1 anisotropic filtering, as well as high-dynamic-range (HDR) 16-bit and 32-bit floating-point data format filtering.

The texture unit is deeply pipelined. Although it contains a cache to capture filtering locality, it streams hits mixed with misses without stalling.

Rasterization

Geometry primitives output from the SMs go in their original round-robin input order to the viewport/clip/setup/raster/zcull block. The viewport and clip units clip the primitives to the standard view frustum and to any enabled user clip planes. They transform postclipping vertices into screen (pixel) space and reject whole primitives outside the view volume as well as back-facing primitives.

Surviving primitives then go to the setup unit, which generates edge equations for the rasterizer. Attribute plane equations are also generated for linear interpolation of pixel attributes in the pixel shader. A coarse-rasterization stage generates all pixel tiles that are at least partially inside the primitive.

The zcull unit maintains a hierarchical z surface, rejecting pixel tiles if they are conservatively known to be occluded by previously drawn pixels. The rejection rate is up to 256 pixels per clock. The screen is subdivided into tiles; each TPC processes a predetermined subset. The pixel tile address therefore selects the destination TPC. Pixel tiles that survive zcull then go to a fine-rasterization stage that generates detailed coverage information and depth values for the pixels.

OpenGL and Direct3D require that a depth test be performed after the pixel shader has generated final color and depth values. When possible, for certain combinations of API state, the Tesla GPU performs the depth test and update ahead of the fragment shader, possibly saving thousands of cycles of processing time, without violating the API-mandated semantics.

The SMC assembles surviving pixels into warps to be processed by a SM running the current pixel shader. When the pixel shader has finished, the pixels are optionally depth tested if this was not done ahead of the shader. The SMC then sends surviving pixels and associated data to the ROP.

Raster operations processor

Each ROP is paired with a specific memory partition. The TPCs feed data to the ROPs via an interconnection network. ROPs handle depth and stencil testing and updates and color blending and updates. The memory controller uses lossless color (up to 8:1) and depth compression (up to 8:1) to reduce bandwidth. Each ROP has a peak rate of four pixels per clock and supports 16-bit floating-point and 32-bit floating-point HDR formats. ROPs support double-rate-depth processing when color writes are disabled.

Each memory partition is 64 bits wide and supports double-data-rate DDR2 and graphics-oriented GDDR3 protocols at up to 1 GHz, yielding a bandwidth of about 16 Gbytes/s.

Antialiasing support includes up to 16× multisampling and supersampling. HDR formats are fully supported. Both algorithms support 1, 2, 4, 8, or 16 samples per pixel and generate a weighted average of the samples to produce the final pixel color. Multisampling executes the pixel shader once to generate a color shared by all pixel samples, whereas supersampling runs the pixel shader once per sample. In both cases, depth values are correctly evaluated for each sample, as required for correct interpenetration of primitives.

Because multisampling runs the pixel shader once per pixel (rather than once per sample), multisampling has become the most popular antialiasing method. Beyond four samples, however, storage cost increases faster than image quality improves, especially with HDR formats. For example, a single 1,600 × 1,200 pixel surface, storing 16 four-component, 16-bit floating-point samples, requires 1,600 × 1,200 × 16 × (64 bits color + 32 bits depth) = 368 Mbytes.

For the vast majority of edge pixels, two colors are enough; what matters is more-detailed coverage information. The coverage-sampling antialiasing (CSAA) algorithm provides low-cost-per-coverage samples, allowing upward scaling. By computing and storing Boolean coverage at up to 16 samples and compressing redundant color and depth and stencil information into the memory footprint and bandwidth of four or eight samples, 16× antialiasing quality can be achieved at 4× antialiasing performance. CSAA is compatible with existing rendering

Table 2. Comparison of antialiasing modes.

	Antialiasing mode								
Feature	**Brute-force supersampling**			**Multisampling**			**Coverage sampling**		
Quality level	1×	4×	16×	1×	4×	16×	1×	4×	16×
Texture and shader samples	1	4	16	1	1	1	1	1	1
Stored color and *z* samples	1	4	16	1	4	16	1	4	4
Coverage samples	1	4	16	1	4	16	1	4	16

techniques including HDR and stencil algorithms. Edges defined by the intersection of interpenetrating polygons are rendered at the stored sample count quality (4× or 8×). Table 2 summarizes the storage requirements of the three algorithms.

Memory and interconnect

The DRAM memory data bus width is 384 pins, arranged in six independent partitions of 64 pins each. Each partition owns 1/6 of the physical address space. The memory partition units directly enqueue requests. They arbitrate among hundreds of in-flight requests from the parallel stages of the graphics and computation pipelines. The arbitration seeks to maximize total DRAM transfer efficiency, which favors grouping related requests by DRAM bank and read/write direction, while minimizing latency as far as possible. The memory controllers support a wide range of DRAM clock rates, protocols, device densities, and data bus widths.

Interconnection network. A single hub unit routes requests to the appropriate partition from the nonparallel requesters (PCI-Express, host and command front end, input assembler, and display). Each memory partition has its own depth and color ROP units, so ROP memory traffic originates locally. Texture and load/store requests, however, can occur between any TPC and any memory partition, so an interconnection network routes requests and responses.

Memory management unit. All processing engines generate addresses in a virtual address space. A memory management unit performs virtual to physical translation. Hardware reads the page tables from local memory to respond to misses on behalf of a hierarchy of translation look-aside buffers spread out among the rendering engines.

Parallel computing architecture

The Tesla scalable parallel computing architecture enables the GPU processor array to excel in throughput computing, executing high-performance computing applications as well as graphics applications. Throughput applications have several properties that distinguish them from CPU serial applications:

- extensive data parallelism—thousands of computations on independent data elements;
- modest task parallelism—groups of threads execute the same program, and different groups can run different programs;
- intensive floating-point arithmetic;
- latency tolerance—performance is the amount of work completed in a given time;
- streaming data flow—requires high memory bandwidth with relatively little data reuse;
- modest inter-thread synchronization and communication—graphics threads do not communicate, and parallel computing applications require limited synchronization and communication.

GPU parallel performance on throughput problems has doubled every 12 to 18 months, pulled by the insatiable demands of the 3D game market. Now, Tesla GPUs in laptops, desktops, workstations,

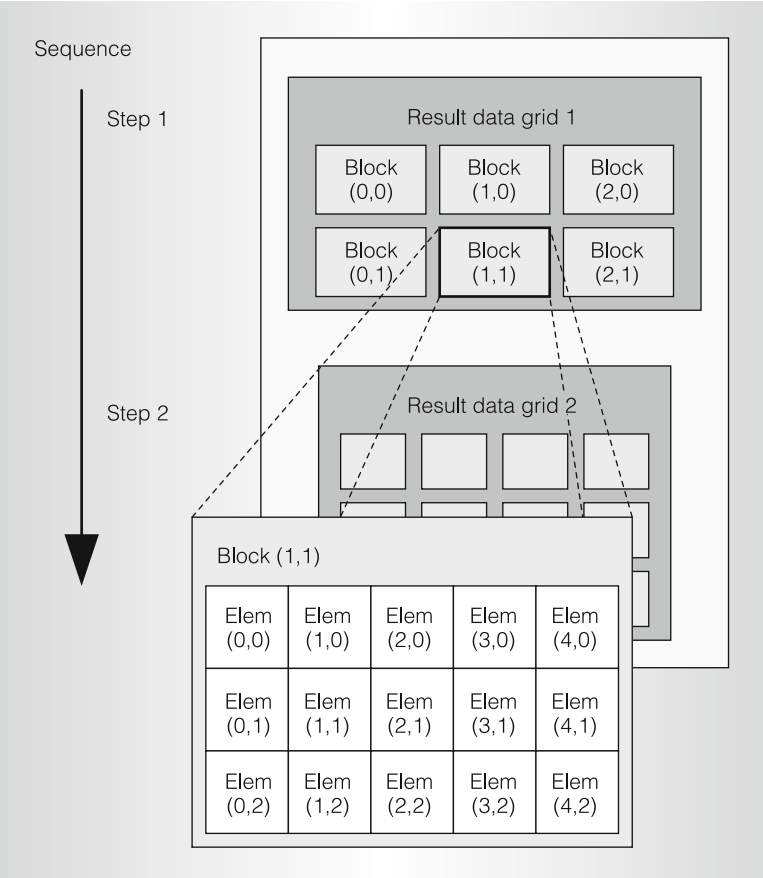

Figure 5. Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel.

and systems are programmable in C with CUDA tools, using a simple parallel programming model.

Data-parallel problem decomposition

To map a large computing problem effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, so that the result blocks can be computed independently in parallel, and the elements within each block can be computed cooperatively in parallel. Figure 5 shows the decomposition of a result data array into a 3×2 grid of blocks, in which each block is further decomposed into a 5×3 array of elements. The two-level parallel decomposition maps naturally to the Tesla architecture: Parallel SMs compute result blocks, and parallel threads compute result elements.

The programmer or compiler writes a program that computes a sequence of result grids, partitioning each result grid into coarse-grained result blocks that are computed independently in parallel. The program computes each result block with an array of fine-grained parallel threads, partitioning the work among threads that compute result elements.

Cooperative thread array or thread block

Unlike the graphics programming model, which executes parallel shader threads independently, parallel-computing programming models require that parallel threads synchronize, communicate, share data, and cooperate to efficiently compute a result. To manage large numbers of concurrent threads that can cooperate, the Tesla computing architecture introduces the *cooperative thread array* (CTA), called a *thread block* in CUDA terminology.

A CTA is an array of concurrent threads that execute the same thread program and can cooperate to compute a result. A CTA consists of 1 to 512 concurrent threads, and each thread has a unique thread ID (TID), numbered 0 through m. The programmer declares the 1D, 2D, or 3D CTA shape and dimensions in threads. The TID has one, two, or three dimension indices. Threads of a CTA can share data in global or shared memory and can synchronize with the barrier instruction. CTA thread programs use their TIDs to select work and index shared data arrays. Multidimensional TIDs can eliminate integer divide and remainder operations when indexing arrays.

Each SM executes up to eight CTAs concurrently, depending on CTA resource demands. The programmer or compiler declares the number of threads, registers, shared memory, and barriers required by the CTA program. When an SM has sufficient available resources, the SMC creates the CTA and assigns TID numbers to each thread. The SM executes the CTA threads concurrently as SIMT warps of 32 parallel threads.

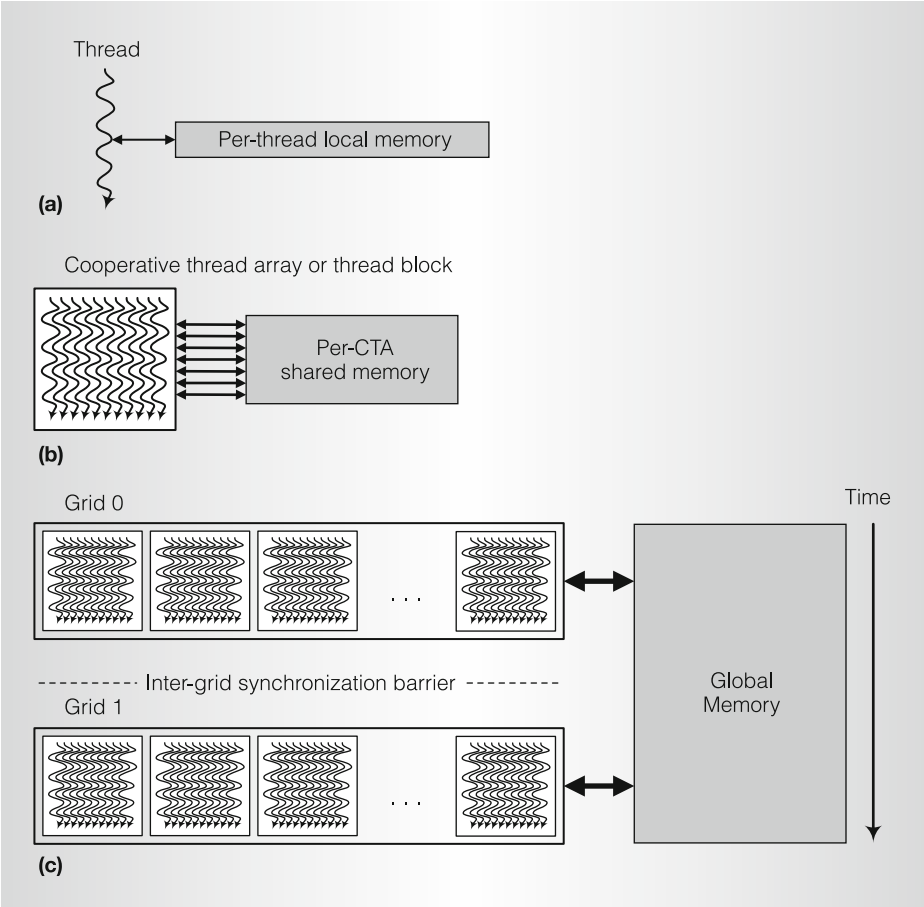

Figure 6. Nested granularity levels: thread (a), cooperative thread array (b), and grid (c). These have corresponding memory-sharing levels: local per-thread, shared per-CTA, and global per-application.

CTA grids

To implement the coarse-grained block and grid decomposition of Figure 5, the GPU creates CTAs with unique CTA ID and grid ID numbers. The compute work distributor dynamically balances the GPU workload by distributing a stream of CTA work to SMs with sufficient available resources.

To enable a compiled binary program to run unchanged on large or small GPUs with any number of parallel SM processors, CTAs execute independently and compute result blocks independently of other CTAs in the same grid. Sequentially dependent application steps map to two sequentially dependent grids. The dependent grid waits for the first grid to complete; then the CTAs of the dependent grid read the result blocks written by the first grid.

Parallel granularity

Figure 6 shows levels of parallel granularity in the GPU computing model. The three levels are

- *thread*—computes result elements selected by its TID;
- *CTA*—computes result blocks selected by its CTA ID;
- *grid*—computes many result blocks, and sequential grids compute sequentially dependent application steps.

Higher levels of parallelism use multiple GPUs per CPU and clusters of multi-GPU nodes.

Parallel memory sharing

Figure 6 also shows levels of parallel read/write memory sharing:

- *local*—each executing thread has a private per-thread local memory for register spill, stack frame, and addressable temporary variables;
- *shared*—each executing CTA has a per-CTA shared memory for access to data shared by threads in the same CTA;
- *global*—sequential grids communicate and share large data sets in global memory.

Threads communicating in a CTA use the fast barrier synchronization instruction to wait for writes to shared or global memory to complete before reading data written by other threads in the CTA. The load/store memory system uses a relaxed memory order that preserves the order of reads and writes to the same address from the same issuing thread and from the viewpoint of CTA threads coordinating with the barrier synchronization instruction. Sequentially dependent grids use a global intergrid synchronization barrier between grids to ensure global read/write ordering.

Transparent scaling of GPU computing

Parallelism varies widely over the range of GPU products developed for various market segments. A small GPU might have one SM with eight SP cores, while a large GPU might have many SMs totaling hundreds of SP cores.

The GPU computing architecture transparently scales parallel application performance with the number of SMs and SP cores. A GPU computing program executes on any size of GPU without recompiling, and is insensitive to the number of SM multiprocessors and SP cores. The program does not know or care how many processors it uses.

The key is decomposing the problem into independently computed blocks as described earlier. The GPU compute work distribution unit generates a stream of CTAs and distributes them to available SMs to compute each independent block. Scalable programs do not communicate among CTA blocks of the same grid; the same grid result is obtained if the CTAs execute in parallel on many cores, sequentially on one core, or partially in parallel on a few cores.

CUDA programming model

CUDA is a minimal extension of the C and C++ programming languages. A programmer writes a serial program that calls parallel kernels, which can be simple functions or full programs. The CUDA program executes serial code on the CPU and executes parallel kernels across a set of parallel threads on the GPU. The programmer organizes these threads into a hierarchy of thread blocks and grids as described earlier. (A CUDA thread block is a GPU CTA.)

Figure 7 shows a CUDA program executing a series of parallel kernels on a heterogeneous CPU–GPU system. **KernelA** and **KernelB** execute on the GPU as grids of **nBlkA and nBlkB** thread blocks (CTAs), which instantiate **nTidA** and **nTidB** threads per CTA.

The CUDA compiler **nvcc** compiles an integrated application C/C++ program containing serial CPU code and parallel GPU kernel code. The CUDA runtime API manages the GPU as a computing device that acts as a coprocessor to the host CPU with its own memory system.

The CUDA programming model is similar in style to a single-program multiple-data (SPMD) software model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However, CUDA is more flexible than most SPMD implementations because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step.

CUDA extends C/C++ with the declaration specifier keywords **__global__** for kernel entry functions, **__device__** for global variables, and **__shared__** for shared-memory variables. A CUDA kernel's text is simply a C function for one sequential thread. The built-in variables **threadIdx.{x, y, z}** and **blockIdx.{x, y, z}** provide the thread ID within a thread block (CTA), while **blockIdx** provides the CTA ID within a grid. The extended function call syntax **kernel<<<nBlocks,nThreads>>>(args);**

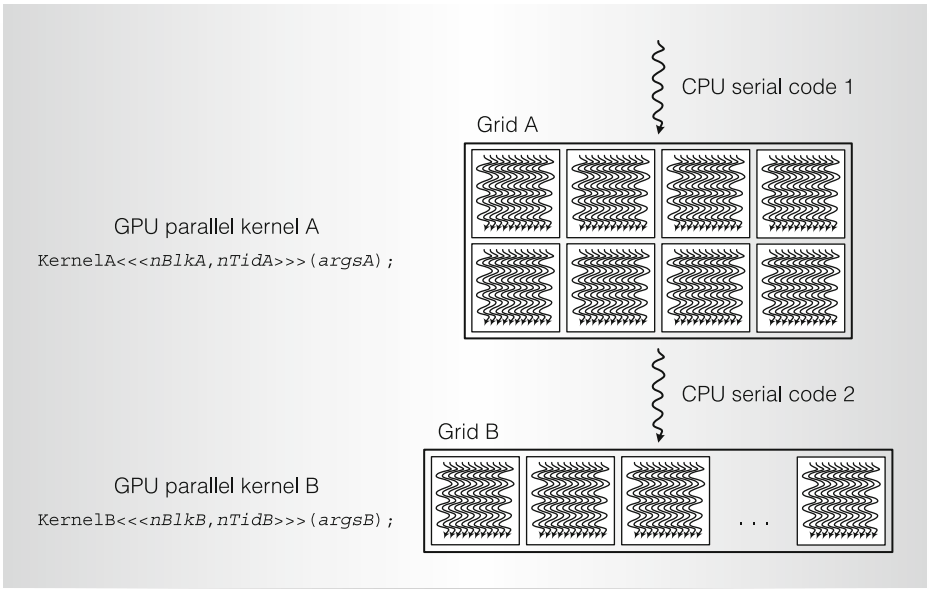

Figure 7. CUDA program sequence of kernel A followed by kernel B on a heterogeneous CPU–GPU system.

invokes a parallel kernel function on a grid of **nBlocks**, where each block instantiates **nThreads** concurrent threads, and **args** are ordinary arguments to function **kernel()**.

Figure 8 shows an example serial C program and a corresponding CUDA C program. The serial C program uses two nested loops to iterate over each array index and compute **c[idx]** = **a[idx]** + **b[idx]** each trip. The parallel CUDA C program has no loops. It uses parallel threads to compute the same array indices in parallel, and each thread computes only one sum.

Scalability and performance

The Tesla unified architecture is designed for scalability. Varying the number of SMs, TPCs, ROPs, caches, and memory partitions provides the right mix for different performance and cost targets in the value, mainstream, enthusiast, and professional

```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    . . .
    addMatrix(a, b, c, N);
}
```

(a)

```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void  main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.

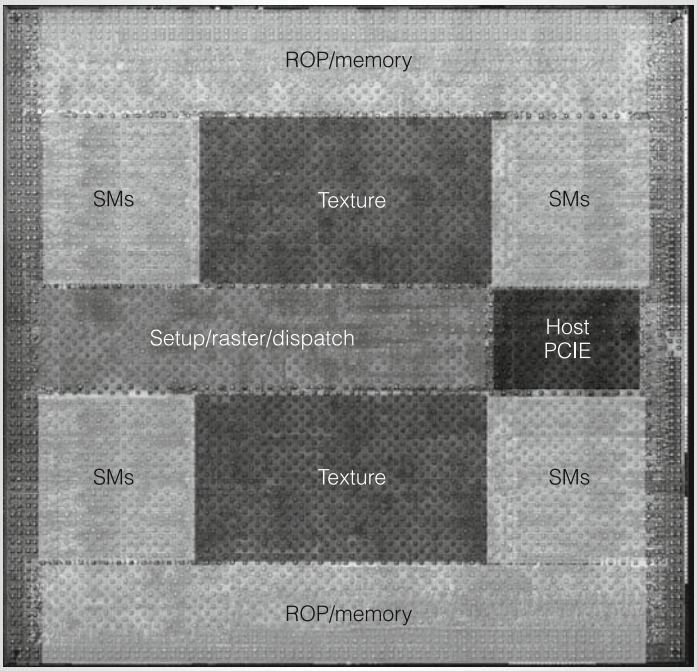

Figure 9. GeForce 8800 Ultra die layout.

market segments. NVIDIA's Scalable Link Interconnect (SLI) enables multiple GPUs to act together as one, providing further scalability.

CUDA C/C++ applications executing on Tesla computing platforms, Quadro workstations, and GeForce GPUs deliver compelling computing performance on a range of large problems, including more than 100× speedups on molecular modeling, more than 200 Gflops on *n*-body problems, and real-time 3D magnetic-resonance imaging.[12–14] For graphics, the GeForce 8800 GPU delivers high performance and image quality for the most demanding games.[15]

Figure 9 shows the GeForce 8800 Ultra physical die layout implementing the Tesla architecture shown in Figure 1. Implementation specifics include

- 681 million transistors, 470 mm^2;
- TSMC 90-nm CMOS;
- 128 SP cores in 16 SMs;
- 12,288 processor threads;
- 1.5-GHz processor clock rate;
- peak 576 Gflops in processors;
- 768-Mbyte GDDR3 DRAM;
- 384-pin DRAM interface;
- 1.08-GHz DRAM clock;
- 104-Gbyte/s peak bandwidth; and
- typical power of 150 W at 1.3 V.

The Tesla architecture is the first ubiquitous supercomputing platform. NVIDIA has shipped more than 50 million Tesla-based systems. This wide availability, coupled with C programmability and the CUDA software development environment, enables broad deployment of demanding parallel-computing and graphics applications.

With future increases in transistor density, the architecture will readily scale processor parallelism, memory partitions, and overall performance. Increased number of multiprocessors and memory partitions will support larger data sets and richer graphics and computing, without a change to the programming model.

We continue to investigate improved scheduling and load-balancing algorithms for the unified processor. Other areas of improvement are enhanced scalability for derivative products, reduced synchronization and communication overhead for compute programs, new graphics features, increased realized memory bandwidth, and improved power efficiency. MICRO

Acknowledgments

We thank the entire NVIDIA GPU development team for their extraordinary effort in bringing Tesla-based GPUs to market.

References

1. J. Montrym and H. Moreton, ''The GeForce 6800,'' *IEEE Micro*, vol. 25, no. 2, Mar./Apr. 2005, pp. 41-51.
2. *CUDA Technology*, NVIDIA, 2007, http://www.nvidia.com/CUDA.
3. *CUDA Programming Guide 1.1*, NVIDIA, 2007; http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
4. J. Nickolls, I. Buck, K. Skadron, and M. Garland, ''Scalable Parallel Programming with CUDA,'' *ACM Queue*, vol. 6, no. 2, Mar./Apr. 2008, pp. 40-53.
5. *DX Specification*, Microsoft; http://msdn.microsoft.com/directx.

6. E. Lindholm, M.J. Kilgard, and H. Moreton, ''A User-Programmable Vertex Engine,'' *Proc. 28th Ann. Conf. Computer Graphics and Interactive Techniques* (Siggraph 01), ACM Press, 2001, pp. 149-158.
7. G. Elder, ''Radeon 9700,'' Eurographics/Siggraph Workshop Graphics Hardware, Hot 3D Session, 2002, http://www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt.
8. *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.
9. J. Andrews and N. Baker, ''Xbox 360 System Architecture,'' *IEEE Micro*, vol. 26, no. 2, Mar./Apr. 2006, pp. 25-37.
10. D. Blythe, ''The Direct3D 10 System,'' *ACM Trans. Graphics*, vol. 25, no. 3, July 2006, pp. 724-734.
11. S.F. Oberman and M.Y. Siu, ''A High-Performance Area-Efficient Multifunction Interpolator,'' *Proc. 17th IEEE Symp. Computer Arithmetic* (Arith-17), IEEE Press, 2005, pp. 272-279.
12. J.E. Stone et al., ''Accelerating Molecular Modeling Applications with Graphics Processors,'' *J. Computational Chemistry*, vol. 28, no. 16, 2007, pp. 2618-2640.
13. L. Nyland, M. Harris, and J. Prins, ''Fast N-Body Simulation with CUDA,'' *GPU Gems 3*, H. Nguyen, ed., Addison-Wesley, 2007, pp. 677-695.
14. S.S. Stone et al., ''How GPUs Can Improve the Quality of Magnetic Resonance Imaging,'' *Proc. 1st Workshop on General Purpose Processing on Graphics Processing Units,* 2007; http://www.gigascale.org/pubs/1175.html.
15. A.L. Shimpi and D. Wilson, ''NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10,'' *AnandTech*, Nov. 2006; http://www.anandtech.com/video/showdoc.aspx?i=2870.

Erik Lindholm is a distinguished engineer at NVIDIA, working in the architecture group. His research interests include graphics processor design and parallel graphics architectures. Lindholm has an MS in electrical engineering from the University of British Columbia.

John Nickolls is director of GPU computing architecture at NVIDIA. His interests include parallel processing systems, languages, and architectures. Nickolls has a BS in electrical engineering and computer science from the University of Illinois and MS and PhD degrees in electrical engineering from Stanford University.

Stuart Oberman is a design manager in the GPU hardware group at NVIDIA. His research interests include computer arithmetic, processor design, and parallel architectures. Oberman has a BS in electrical engineering from the University of Iowa and MS and PhD degrees in electrical engineering from Stanford University. He is a senior member of the IEEE.

John Montrym is a chief architect at NVIDIA, where he has worked in the development of several GPU product families. His research interests include graphics processor design, parallel graphics architectures, and hardware-software interfaces. Montrym has a BS in electrical engineering from the Massachusetts Institute of Technology.

Direct questions and comments about this article to Erik Lindholm or John Nickolls, NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050; elindholm@nvidia.com or jnickolls@nvidia.com.