8.7 Real stuff: The NVIDIA GeForce 8800

(Original section[1])

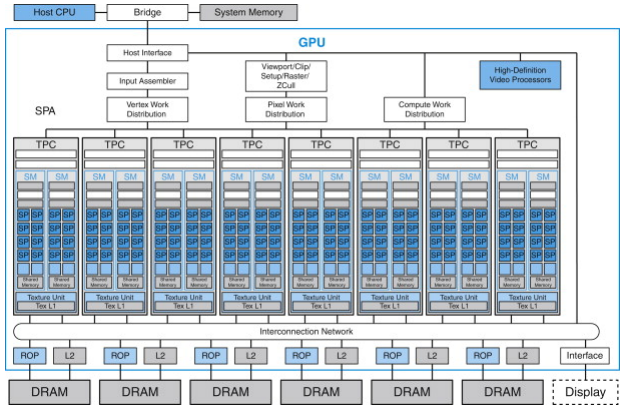
The NVIDIA GeForce 8800 GPU, introduced in November 2006, is a unified vertex and pixel processor design that also supports parallel computing applications written in C using the CUDA parallel programming model. It is the first implementation of the Tesla unified graphics and computing architecture described in COD Section B.4 (Multithreaded multiprocessor architecture) and in Lindholm et al. [2008]. A family of Tesla architecture GPUs addresses the different needs of laptops, desktops, workstations, and servers.

Streaming processor array (SPA)

The GeForce 8800 GPU shown in the figure below contains 128 *streaming processor* (SP) cores organized as 16 *streaming multiprocessors* (SMs). Two SMs share a texture unit in each *texture/processor cluster* (TPC). An array of eight TPCs makes up the *streaming processor array* (SPA), which executes all graphics shader programs and computing programs.

Figure 8.7.1: NVIDIA Tesla unified graphics and computing GPU architecture (COD Figure B.7.1).

This GeForce 8800 has 128 *streaming processor* (SP) cores in 16 *streaming multiprocessors* (SMs), arranged in eight *texture/processor clusters* (TPCs). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.



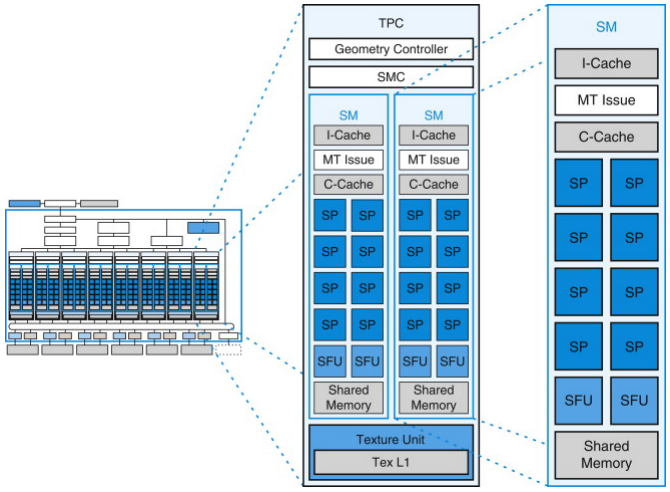
The host interface unit communicates with the host CPU via the PCI-Express bus, checks command consistency, and performs context switching. The input assembler collects geometric primitives (points, lines, triangles). The work distribution blocks dispatch vertices, pixels, and compute thread arrays to the TPCs in the SPA. The TPCs execute vertex and geometry shader programs and computing programs. Output geometric data are sent to the viewport/clip/setup/raster/zcull block to be rasterized into pixel fragments that are then redistributed back into the SPA to execute pixel shader programs. Shaded pixels are sent across the interconnection network for processing by the ROP units. The network also routes texture memory read requests from the SPA to DRAM and reads data from DRAM through a level-2 cache back to the SPA.

Texture/processor cluster (TPC)

Each TPC contains a geometry controller, an SMC, two SMs, and a texture unit as shown in the figure below.

Figure 8.7.2: Texture/processor cluster (TPC) and a streaming multiprocessor (SM) (COD Figure B.7.2).

Each SM has eight *streaming processor* (SP) cores, two SFUs, and a shared memory.



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.

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.

The texture unit processes a texture instruction for one vertex, geometry, or pixel quad, or four compute threads per cycle. Texture instruction sources are texture coordinates, and the outputs are weighted samples, typically a four-component (RGBA) floating-point color. The texture unit is deeply pipelined. Although it contains a streaming cache to capture filtering locality, it streams hits mixed with misses without stalling.

Streaming multiprocessor (SM)

The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs. The SM consists of eight SP thread processor cores, two SFUs, a multithreaded instruction fetch and issue unit (MT issue), an instruction cache, a read-only constant cache, and a 16 KB read/write shared memory. It executes scalar instructions for individual threads.

The GeForce 8800 Ultra clocks the SP cores and SFUs at 1.5 GHz, for a peak of 36 GFLOPS per SM. To optimize power and area efficiency, some SM nondatapath units operate at half the SP clock rate.

To efficiently execute hundreds of parallel threads 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. Each thread has its own thread execution state and can execute an independent code path.

A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The SIMT design, previously described in COD Section B.4 (Multithreaded multiprocessor architecture), shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency.

The SM schedules and executes multiple warp types concurrently. Each issue cycle, the scheduler selects one of the 24 warps to execute a SIMT warp instruction. An issued warp instruction executes as four sets of eight threads over four processor cycles. The SP and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied. 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.

The SM executes *cooperative thread arrays* (CTAs) as multiple concurrent warps which access a shared memory region allocated dynamically for the CTA.

Instruction set

Threads execute scalar instructions, unlike previous GPU vector instruction architectures. Scalar instructions are simpler and compiler-friendly. Texture instructions remain vector-based, taking a source coordinate vector and returning a filtered color vector.

The register-based instruction set includes all the floating-point and integer arithmetic, transcendental, logical, flow control, memory load/store, and texture instructions listed in the PTX instruction table of COD Figure B.4.3 (Basic PTX GPU thread instructions). Memory load/store instructions use integer byte addressing with register-plus-offset address arithmetic. For computing, the load/store instructions access three read-write memory spaces: local memory for per-thread, private, temporary data; shared memory for low-latency per-CTA data shared by the threads of the CTA; and global memory for data shared by all threads. Computing programs use the fast barrier synchronization `bar.sync` instruction to synchronize threads within a CTA that communicate with each other via shared and global memory. The latest Tesla architecture GPUs implement PTX atomic memory operations, which facilitate parallel reductions and parallel data structure management.

Streaming processor (SP)

The multithreaded SP core is the primary thread processor, as introduced in COD Section B.4 (Multithreaded multiprocessor architecture). Its register file provides 1024 scalar 32-bit registers for up to 96 threads (more threads than in the example SP of COD Section B.4 (Multithreaded multiprocessor architecture)). Its 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. The add and multiply operations use IEEE round-to-nearest-even as the default rounding mode. The SP core also implements all of the 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX instructions in COD Figure B.4.3 (Basic PTX GPU thread instructions). The processor is fully pipelined, and latency is optimized to balance delay and area.

Special function unit (SFU)

The SFU supports computation of both transcendental functions and planar attribute interpolation. As described in COD Section B.6 (Floating-point arithmetic), it uses quadratic interpolation based on enhanced minimax approximations to approximate the reciprocal, reciprocal square root, $\log_2 x$, 2^x, and sin/cos functions at one result per cycle. The SFU also supports pixel attribute interpolation such as color, depth, and texture coordinates at four samples per cycle.

Rasterization

Geometry primitives 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 view frustum and to any enabled user clip planes, and then transform the vertices into screen (pixel) space.

Surviving primitives then go to the setup unit, which generates edge equations for the rasterizer. 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. Pixels that survive zcull then go to a fine-rasterization stage that generates detailed coverage information and depth values.

The depth test and update can be performed ahead of the fragment shader, or after, depending on current state. The SMC assembles surviving pixels into warps to be processed by an SM running the current pixel shader. The SMC then sends surviving pixel and associated data to the ROP.

Raster operations processor (ROP) and memory system

Each ROP is paired with a specific memory partition. For each pixel fragment emitted by a pixel shader program, ROPs perform depth and stencil testing and updates, and in parallel, color blending and updates. Lossless color compression (up to 8:1) and depth compression (up to 8:1) are used to reduce DRAM 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.

Antialiasing support includes up to 16 × multisampling and supersampling. The *coverage-sampling antialiasing* (CSAA) algorithm computes and stores Boolean coverage at up to 16 samples and compresses redundant color, depth, and stencil information into the memory footprint and a bandwidth of four or eight samples for improved performance.

The DRAM memory data bus width is 384 pins, arranged in six independent partitions of 64 pins each. Each partition supports double-data-rate DDR2 and graphics-oriented GDDR3 protocols at up to 1.0 GHz, yielding a bandwidth of about 16 GB/s per partition, or 96 GB/s.

The memory controllers support a wide range of DRAM clock rates, protocols, device densities, and data bus widths. Texture and load/store requests can occur between any TPC and any memory partition, so an interconnection network routes requests and responses.

Scalability

The Tesla unified architecture is designed for scalability. Varying the number of SMs, TPCs, ROPs, caches, and memory partitions provides the right balance for different performance and cost targets in GPU market segments. *Scalable link interconnect* (SLI) connects multiple GPUs, providing further scalability.

Performance

The GeForce 8800 Ultra clocks the SP thread processor cores and SFUs at 1.5 GHz, for a theoretical operation peak of 576 GFLOPS. The GeForce 8800 GTX has a 1.35 GHz processor clock and a corresponding peak of 518 GFLOPS.

The following three sections compare the performance of a GeForce 8800 GPU with a multicore CPU on three different applications—dense linear algebra, fast Fourier transforms, and sorting. The GPU programs and libraries are compiled CUDA C code. The CPU code uses the single-precision multithreaded Intel MKL 10.0 library to leverage SSE instructions and multiple cores.

Dense linear algebra performance

Dense linear algebra computations are fundamental in many applications. Volkov and Demmel [2008] present GPU and CPU performance results for single-precision dense matrix-matrix multiplication (the SGEMM routine) and LU, QR, and Cholesky matrix factorizations. The figure below compares GFLOPS rates on SGEMM dense matrix-matrix multiplication for a GeForce 8800 GTX GPU with a quad-core CPU. COD Figure B.7.4 (Dense matrix factorization performance rates) compares GFLOPS rates on matrix factorization for a GPU with a quad-core CPU.

Figure 8.7.3: SGEMM dense matrix-matrix multiplication performance rates (COD Figure B.7.3).

The graph shows single-precision GFLOPS rates achieved in multiplying square N×N matrices (solid lines) and thin N×64 and 64×N matrices (dashed lines). Adapted from Figure 6 of Volkov and Demmel [2008]. The black lines are a 1.35 GHz GeForce 8800 GTX using Volkov's SGEMM code (now in NVIDIA CUBLAS 2.0) on matrices in GPU memory. The blue lines are a quad-core 2.4 GHz Intel Core2 Quad Q6600, 64-bit Linux, Intel MKL 10.0 on matrices in CPU memory.

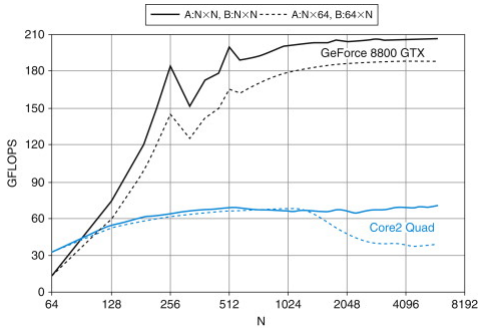
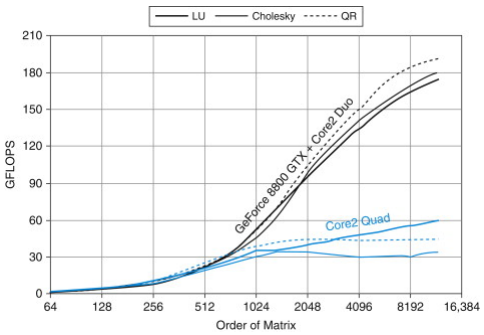


Figure 8.7.4: Dense matrix factorization performance rates (COD Figure B.7.4).

The graph shows GFLOPS rates achieved in matrix factorizations using the GPU and using the CPU alone. Adapted from Figure 7 of Volkov and Demmel [2008]. The black lines are for a 1.35 GHz NVIDIA GeForce 8800 GTX, CUDA 1.1, Windows XP attached to a 2.67 GHz Intel Core2 Duo E6700 Windows XP, including all CPU—GPU data transfer times. The blue lines are for a quad-core 2.4 GHz Intel Core2 Quad Q6600, 64-bit Linux, Intel MKL 10.0.



Because SGEMM matrix-matrix multiply and similar BLAS3 routines are the bulk of the work in matrix factorization, their performance sets an upper bound on factorization rate. As the matrix order increases beyond 200 to 400, the factorization problem becomes large enough that SGEMM can leverage the GPU parallelism and overcome the CPU-GPU system and copy overhead. Volkov's SGEMM matrix-matrix multiply achieves 206 GFLOPS, about 60% of the GeForce 8800 GTX peak multiply-add rate, while the QR factorization reached 192 GFLOPS, about 4.3 times the quad-core CPU.

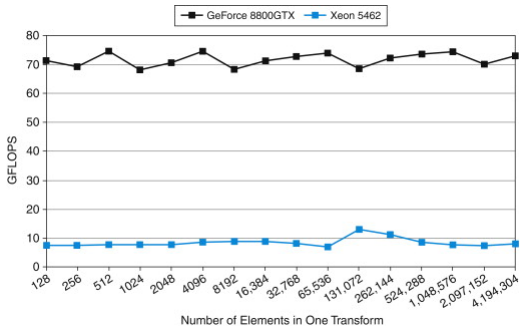
FFT performance

Fast Fourier Transforms (FFTs) are used in many applications. Large transforms and multidimensional transforms are partitioned into batches of smaller 1D transforms.

The figure below compares the in-place 1D complex single-precision FFT performance of a 1.35 GHz GeForce 8800 GTX (dating from late 2006) with a 2.8 GHz quad-Core Intel Xeon E5462 series (code named "Harpertown," dating from late 2007). CPU performance was measured using the Intel *Math Kernel Library* (MKL) 10.0 FFT with four threads. GPU performance was measured using the NVIDIA CUFFT 2.1 library and batched 1D radix-16 decimation-in-frequency FFTs. Both CPU and GPU throughput performance was measured using batched FFTs; batch size was $2^{24}/n$, where n is the transform size. Thus, the workload for every transform size was 128 MB. To determine GFLOPS rate, the number of operations per transform was taken as $5n \log_2 n$.

Figure 8.7.5: Fast Fourier Transform throughput performance (COD Figure B.7.5).

The graph compares the performance of batched one-dimensional in-place complex FFTs on a 1.35 GHz GeForce 8800 GTX with a quad-core 2.8 GHz Intel Xeon E5462 series (code named "Harpertown"), 6MB L2 Cache, 4GB Memory, 1600 FSB, Red Hat Linux, Intel MKL 10.0.



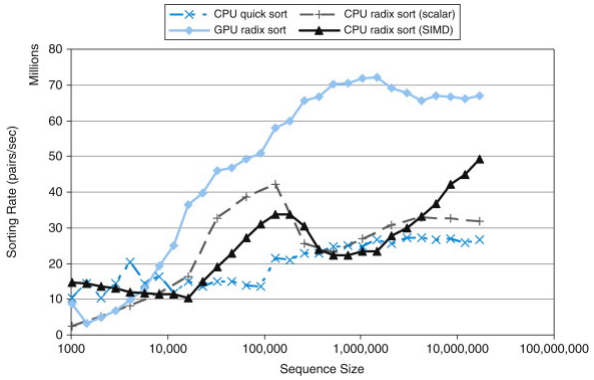
Sorting performance

In contrast to the applications just discussed, sort requires far more substantial coordination among parallel threads, and parallel scaling is correspondingly harder to obtain. Nevertheless, a variety of well-known sorting algorithms can be efficiently parallelized to run well on the GPU. Satish et al. [2008] detail the design of sorting algorithms in CUDA, and the results they report for radix sort are summarized below.

The figure below compares the parallel sorting performance of a GeForce 8800 Ultra with an 8-core Intel Clovertown system, both of which date to early 2007. The CPU cores are distributed between two physical sockets. Each socket contains a multichip module with twin Core2 chips, and each chip has a 4MB L2 cache. All sorting routines were designed to sort key-value pairs where both keys and values are 32-bit integers. The primary algorithm being studied is radix sort, although the quicksort-based `parallel_sort()` procedure provided by Intel's Threading Building Blocks is also included for comparison. Of the two CPU-based radix sort codes, one was implemented using only the scalar instruction set and the other utilizes carefully hand-tuned assembly language routines that take advantage of the SSE2 SIMD vector instructions.

Figure 8.7.6: Parallel sorting performance (COD Figure B.7.6).

This graph compares sorting rates for parallel radix sort implementations on a 1.5 GHz GeForce 8800 Ultra and an 8-core 2.33 GHz Intel Core2 Xeon E5345 system.



The graph itself shows the achieved sorting rate—defined as the number of elements sorted divided by the time to sort—for a range of sequence sizes. It is apparent from this graph that the GPU radix sort achieved the highest sorting rate for all sequences of 8K-elements and larger. In this range, it is on average 2.6 times faster than the quicksort-based routine and roughly two times faster than the radix sort routines, all of which were using the eight available CPU cores. The CPU radix sort performance varies widely, likely due to poor cache locality of its global permutations.

(*1) This section is in original form.