Graphics Processors

Why Graphics Processors?
- Graphics programs have different characteristics from general purpose programs
  - Highly parallel operations on lots of pixels
  - Graphics algorithms have common kernels for texture, edge detection, rasterization, etc.
  - Rasterization: Converting a graph description (e.g., points, lines etc.) into dots/pixels on graph
  - Graphics-intensive applications (e.g., games) need a lot of compute power
- Some graphics processors are expanding to general purpose applications
  - Called GPGPUs: General Purpose GPUs

Two Graphics Processor Types
- Vertex Processors
  - Operate on vertices of primitives (e.g., points, lines, triangles)
  - Operations: transforming coordinates to screen space, setting up lighting and texture parameters
  - Designed for low latency, high precision math operations
- Pixel Fragment Processors
  - Operate on rasterizer output
  - Fills interior of primitives and interpolated parameters
  - Designed for high latency, low precision texture filtering
- Typically need many more pixel fragment processors than vertex processors (pixels more numerous than vertices)
- Two types have been converging due to need of programming generality

Nvidia Tesla Design Goals
- Unified processor architecture to execute vertex programs & pixel fragment shader programs
  - Enables dynamic load balancing of varying workloads
  - Allows sharing of expensive hardware (e.g., texture units)
  - Facilitated parallel computing capability
  - Downside: Efficient load balancing between different shader types is more difficult
- Architectural scalability
- High performance
- Low power
- Area efficiency

Tesla Architecture
- Based on a scalable Streaming Processor Array (SPA)
- GeForce 8800 GPU (paper figure 1)
  - Eight independent processing units: Texture/Processor Clusters (TPC)
  - Each unit has 16 streaming processors (SPs) organized as 2 8-processor streaming multiprocessors (SMs)
- SPA performs all GPU’s calculations
- Memory system:
  - External DRAM control
  - Fixed-function raster operation processors (ROPs)
    - Perform color and depth frame buffer operations on memory
  - Level-2 cache
  - Interconnection network

Tesla Unit Functions
- GPU Host Interface Unit
  - Communicates with host CPU
  - Responds to CPU commands
  - Fetches data from system memory
  - Checks command consistency
  - Performs Context Switching
- Input Assembler
  - Collects geometric primitives (points, lines, triangles, strips) and fetches associated vertex input attribute data
- Work Distribution Units
  - Includes vertex, pixel, and compute work distribution
  - Forward input assembler’s output to processors
Texture/Processor Cluster (TPC)
- Number scales from 1 to 8 or more depending on required performance
- TPC Components (paper figure 2)
  - Geometry Controller
    - Maps logical graphics vertex pipeline into physical SMs
    - Manages dedicated on-chip input & output vertex attribute storage, forwards content as needed
  - SM Controller (SMC)
  - Two Streaming Multiprocessors (SMs)
  - Texture Unit
    - Processes one group of four threads per cycle
    - Texture instructions: sources are texture coordinates, outputs are filtered samples
    - One texture unit for each two SMs

Streaming Multiprocessor (SM)
- Unified graphics and computing multiprocessor
- Executes vertex, geometry, and pixel-fragment shader programs, as well as parallel computing programs
- SM components (paper figure 3)
  - Eight streaming processors (SPs)
    - Each SP contains scalar multiply-add (MAD) unit
  - Two special-function units (SFUs)
  - Multithreaded inst fetch and issue unit (MT issue)
  - Instruction cache
  - Read-only constant cache (ROM)
  - 16KB read/write shared memory
    - Holds graphics input buffers or shared data for parallel computing

SM Multithreading
- SMs are hardware multi-threaded
- Each SM manages and executes up to 768 (24x32) concurrent threads in hardware
- Parallel computing threads use CUDA (Computer Unified Device Architecture) kernels in C to specify how one thread computes a result
- Each SM thread has its own thread execution state, and can execute an independent code path
- Support for fine-grain parallelism
  - Lightweight thread creation and no overhead thread scheduling
  - Concurrent threads can synchronize at barriers quickly with a single SM instruction

Single-Instruction, Multiple-Thread (SIMT)
- Tesla SM creates, manages, schedules, executes threads in “warps”, each containing 32 threads
- Each SM manages a pool of 24 warps (768 total threads)
- All threads in a a warp are of the same type (pixel, geometry, vertex, or compute)
- Warp threads start together at the same program address, but can branch and execute independently
- Each SM thread has its own thread execution state, and can execute an independent code path
- Support for fine-grain parallelism
- Lightweight thread creation and no overhead thread scheduling
- Concurrent threads can synchronize at barriers quickly with a single SM instruction
- Very efficient execution if all warp threads have same execution path — otherwise, branch divergence causes serialization

SIMT vs. SIMD
- SIMD: Applies one instruction to multiple data lanes (e.g., vector instructions)
- SIMT: Applies one instruction to multiple independent threads in parallel
- SIMT enables programmers to write thread-level parallel code for independent threads, and data-parallel code for coordinated code
- Data-parallel code has much better performance
- Conversely, SIMD requires software to manually coalesce data into vectors, and manually manage divergence

Memory Access
- Three different memory spaces
  - Local memory: per-thread, private data, implemented in external DRAM
  - Shared memory: low-latency access to shared data in the same SM
  - Global memory: data shared by all threads, implemented in external DRAM
- Memory instructions:
  - Loads: load-global, load-shared, load-local
  - Stores: store-global, store-shared, store-local
  - Fast barrier synchronization instruction within the SM
- Atomic operations
- To improve memory bandwidth, local and global loads/stores coalesce individual parallel thread accesses from the same warp into fewer memory block accesses
Throughput Applications

- Characteristics of throughput applications
  - Extensive data parallelism: Many computes on independent data
  - Modest task parallelism: Groups of threads execute the same program, and different groups run different programs
  - Intensive FP arithmetic
  - Latency tolerance: Throughput, not latency, is the performance metric
  - Streaming data flow: Little reuse, high bandwidth
  - Modest inter-thread synchronization & communication
- Similar behavior in graphics and parallel computing applications
- Discuss: GPUs vs CPUs

Parallel Computing Architecture

- Data parallel problem decomposition
  - Need to partition problems to small problems to be done in parallel (paper figure 5)
  - Need load balancing
  - Parallel granularity based on thread, CTA, or grid (paper figure 6)
- Sharing: Local (private per thread), shared (CTA), global (grid)
- Cooperative thread array (CTA) or thread block
  - Array of concurrent threads that execute the same program, cooperate to compose a result
  - CTA consists of 1 to 512 concurrent threads, each with a unique ID
  - Could implement 1D, 2D, or 3D shapes and dimensions in threads – a thread ID could have 1, 2, or 3 dimension indices
  - Each SM executes up to 8 CTAs concurrently
- CUDA programming model (paper figure 8)

Reading Assignment

- Project Progress Reports due today