Threading Hardware in G80
Sources

• Slides by ECE 498 AL: Programming Massively Parallel Processors: Wen-Mei Hwu
• John Nickolls, NVIDIA
3D API: OpenGL or Direct3D

CPU-GPU Boundary (AGP/PCIe)

3D Application Or Game

Assembled Primitives

Pixel Location Stream

Pixel Updates

Raster Operations

Frame Buffer

Programmable Fragment Processor

Transformed Fragments

Programmable Vertex Processor

Transformed Vertices

Rasterization and Interpolation

Pre-transformed Fragments

Pre-transformed Vertices

Transformed Vertices

Assembled Primitives

Vertex Index Stream

GPU Front End

GPU Command & Data Stream

3D API Commands

Programmable pipeline

3D Application Or Game

CPU-GPU Boundary (AGP/PCIe)
3D API:
- OpenGL or Direct3D

3D Application Or Game

Assembled Primitives

Pixel Location Stream

CPU-GPU Boundary (AGP/PCIe)

Frame Buffer

Transformed Vertices

Pre-transformed Fragments

Unified Programmable pipeline

GPU Command \& Data Stream

Verte Index Stream

GPU Front End

Primitive Assembly

Rasterization and Interpolation

Raster Operations

Frame Buffer

Pre-transformed Vertices

Pre-transformed Fragments

Transformed Vertices

Transformed Fragments

Unified Vertex, Fragment, Geometry Processor

3D API Commands

Transformed Fragments
General Diagram (6800/NV40)
TurboCache

- Uses PCI-Express bandwidth to render directly to system memory
- Card needs less memory
- Performance boost while lowering cost
- TurboCache Manager dynamically allocates from main memory
- Local memory used to cache data and to deliver peak performance when needed
An NV40 vertex processor is able to execute one vector operation (up to four FP32 components), one scalar FP32 operation, and make one access to the texture per clock cycle.
NV40 Fragment Processors

Early termination from mini z buffer and z buffer checks; resulting sets of 4 pixels (quads) passed on to fragment units
Why NV40 series was better

- Massive parallelism
- Scalability
  - Lower end products have fewer pixel pipes and fewer vertex shader units
- Computation Power
  - 222 million transistors
  - First to comply with Microsoft’s DirectX 9 spec
- Dynamic Branching in pixel shaders
Dynamic Branching

• Helps detect if pixel needs shading
• Instruction flow handled in groups of pixels
• Specify branch granularity (the number of consecutive pixels that take the same branch)
• Better distribution of blocks of pixels between the different quad engines
General Diagram (7800/G70)
<table>
<thead>
<tr>
<th></th>
<th>GeForce Go 6800 Ultra</th>
<th>GeForce Go 7800 GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shader Perf (Pixels per Clock)</strong></td>
<td>1x</td>
<td>2x</td>
</tr>
<tr>
<td><strong>Geometry</strong></td>
<td>1x</td>
<td>1.6x</td>
</tr>
<tr>
<td><strong>Transistors</strong></td>
<td>202 million</td>
<td>302 million</td>
</tr>
<tr>
<td><strong>Power Consumption</strong></td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>Identical</td>
<td>Identical</td>
</tr>
<tr>
<td><strong>Memory Interface</strong></td>
<td>256-bit</td>
<td>256-bit</td>
</tr>
<tr>
<td><strong>Clock Frequencies</strong></td>
<td>450/550</td>
<td>400/550</td>
</tr>
<tr>
<td><strong>DirectX™ 9.0</strong></td>
<td>Shader Model 3.0</td>
<td>Shader Model 3.0</td>
</tr>
<tr>
<td><strong>Transparency Antialiasing</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Programmable Video Processor</strong></td>
<td>PureVideo™</td>
<td>PureVideo™</td>
</tr>
<tr>
<td><strong>Power Management</strong></td>
<td>PowerMizer™ 5.0</td>
<td>PowerMizer™ 6.0</td>
</tr>
<tr>
<td><strong>Host Interface</strong></td>
<td>PCI Express</td>
<td>PCI Express</td>
</tr>
</tbody>
</table>
GeForce Go 7800 – Power Issues

- Power consumption and package are the same as the 6800 Ultra chip, meaning notebook designers do not have to change very much about their thermal designs
- Dynamic clock scaling can run as slow as 16 MHz
  - This is true for the engine, memory, and pixel clocks
- Heavier use of clock gating than the desktop version
- Runs at voltages lower than any other mobile performance part
- Regardless, you won’t get much battery-based runtime for a 3D game
GeForce 7800 GTX Parallelism

8 Vertex Engines

Z-Cull → Triangle Setup/Raster

Shader Instruction Dispatch

24 Pixel Shaders

Fragment Crossbar

16 Raster Operation Pipelines

Memory Partition

Memory Partition

Memory Partition

Memory Partition
G80 – Graphics Mode

• The future of GPUs is programmable processing
• So – build the architecture around the processor

- The diagram shows the architecture of the G80 GPU, with various components such as the Host, Input Assembler, Vtx Thread Issue, Geom Thread Issue, Pixel Thread Issue, Setup / Rstr / ZCull, and the Thread Processor. The diagram highlights the flow of data and processing through these components.
G80 CUDA mode – A Device Example

- Processors execute computing threads
- New operating mode/HW interface for computing
Why Use the GPU for Computing?

- The GPU has evolved into a very flexible and powerful processor:
  - It’s programmable using high-level languages
  - It supports 32-bit floating point precision
  - It offers lots of GFLOPS:

![GPU Performance Chart](chart.png)

- GPU in every PC and workstation

---

**GFLOPS Chart**

- G80 = GeForce 8800 GTX
- G71 = GeForce 7900 GTX
- G70 = GeForce 7800 GTX
- NV40 = GeForce 6800 Ultra
- NV35 = GeForce FX 5950 Ultra
- NV30 = GeForce FX 5800
What is Behind such an Evolution?

- The GPU is specialized for compute-intensive, highly data parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control

- The fast-growing video game industry exerts strong economic pressure that forces constant innovation
What is (Historical) GPGPU?

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
Previous GPGPU Constraints

• Dealing with graphics API
  – Working with the corner cases of the graphics API

• Addressing modes
  – Limited texture size/dimension

• Shader capabilities
  – Limited outputs

• Instruction sets
  – Lack of Integer & bit ops

• Communication limited
  – Between pixels
  – Scatter a[i] = p
An Example of Physical Reality Behind CUDA

CPU (host)

GPU w/ local DRAM (device)
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

```
threadID  0 1 2 3 4 5 6 7
```

...
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations and barrier synchronization**
  - Threads in different blocks cannot cooperate
Thread Batching: Grids and Blocks

• A kernel is executed as a grid of thread blocks
  – All threads share data memory space

• A thread block is a batch of threads that can cooperate with each other by:
  – Synchronizing their execution
    • For hazard-free shared memory accesses
  – Efficiently sharing data through a low latency shared memory

• Two threads from two different blocks cannot cooperate

Courtesy: NDVIA
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
CUDA Device Memory Space Overview

• Each thread can:
  – R/W per-thread registers
  – R/W per-thread local memory
  – R/W per-block shared memory
  – R/W per-grid global memory
  – Read only per-grid constant memory
  – Read only per-grid texture memory

• The host can R/W global, constant, and texture memories
Global, Constant, and Texture Memories
(Long Latency Accesses)

• Global memory
  – Main means of communicating R/W Data between host and device
  – Contents visible to all threads

• Texture and Constant Memories
  – Constants initialized by host
  – Contents visible to all threads

Courtesy: NDVIA
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
CUDA Memory Model Overview

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- **We will focus on global memory for now**
  - Constant and texture memory will come later
Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism
Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks

CPU Serial Code

GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);

CPU Serial Code

GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);
Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share global memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution using barrier
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate
CUDA Thread Block

- Programmer declares (Thread) Block:
  - Block size 1 to $512$ concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads

- All threads in a Block execute the same thread program
- Threads share data and synchronize while doing their share of the work
- Threads have thread id numbers within Block
- Thread program uses thread id to select work and address shared data

Courtesy: John Nickolls, NVIDIA
GeForce-8 Series HW Overview
CUDA Processor Terminology

- **SPA**
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)

- **TPC**
  - Texture Processor Cluster (2 SM + TEX)

- **SM**
  - Streaming Multiprocessor (8 SP)
  - Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block

- **SP**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- texture and global memory access
G80 Thread Computing Pipeline

- Processors of GPUs support threading
- So build the operating mode specifically for computing

Generates Thread grids based on kernel calls
Thread Life Cycle in HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM’s
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks
SM Executes Blocks

- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.

- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution
Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
- If one global memory access is needed for every 4 instructions
- A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency
SM Instruction Buffer – Warp Scheduling

• Fetch one warp instruction/cycle
  – from instruction L1 cache
  – into any instruction buffer slot

• Issue one “ready-to-go” warp instruction/cycle
  – from any warp - instruction buffer slot
  – operand scoreboard used to prevent hazards

• Issue selection based on round-robin/age of warp

• SM broadcasts the same instruction to 32 Threads of a Warp
Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Instruction becomes ready after the needed values are deposited
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - allows Memory/Processor ops to proceed in shadow of other waiting Memory/Processor ops

TB = Thread Block, W = Warp
Granularity Considerations

- For Matrix Multiplication, should I use 4X4, 8X8, 16X16 or 32X32 tiles?
  - For 4X4, we have 16 threads per block. Since each SM can take up to 768 threads, the thread capacity allows 48 blocks. However, each SM can only take up to 8 blocks, thus there will be only 128 threads in each SM!
    - There are 8 warps but each warp is only half full.
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it could take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
    - There are 16 warps available for scheduling in each SM
    - Each warp spans four slices in the y dimension
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
    - There are 24 warps available for scheduling in each SM
    - Each warp spans two slices in the y dimension
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!
Memory Hardware in G80
CUDA Device Memory Space: Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Parallel Memory Sharing

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication
SM Memory Architecture

- Threads in a block share data & results
  - In Memory and Shared Memory
  - Synchronize at barrier instruction
- Per-Block Shared Memory Allocation
  - Keeps data close to processor
  - Minimize trips to global Memory
  - Shared Memory is dynamically allocated to blocks, one of the limiting resources
SM Register File

• Register File (RF)
  – 32 KB (8K entries) for each SM in G80
• TEX pipe can also read/write RF
  – 2 SMs share 1 TEX
• Load/Store pipe can also read/write RF
Programmer View of Register File

• There are 8192 registers in each SM in G80
  – This is an implementation decision, not part of CUDA
  – Registers are dynamically partitioned across all blocks assigned to the SM
  – Once assigned to a block, the register is NOT accessible by threads in other blocks
  – Each thread in the same block only access registers assigned to itself
Matrix Multiplication Example

• If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  – Each block requires $10 \times 256 = 2560$ registers
  – $8192 = 3 \times 2560 + \text{change}$
  – So, three blocks can run on an SM as far as registers are concerned

• How about if each thread increases the use of registers by 1?
  – Each Block now requires $11 \times 256 = 2816$ registers
  – $8192 < 2816 \times 3$
  – Only two Blocks can run on an SM, \textbf{1/3 reduction of parallelism}!!!
More on Dynamic Partitioning

• Dynamic partitioning gives more flexibility to compilers/programmers
  – One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    • This allows for finer grain threading than traditional CPU threading models.
  – The compiler can tradeoff between instruction-level parallelism and thread level parallelism