GPUs and Modern Parallel Processor Architectures

Jaakko Lehtinen
Aalto Dept. Of Computer Science / NVIDIA Research
Today

• The Graphics Pipeline: Evolution of the GPU

• Bigger picture: Parallel processor designs
  – Throughput-optimized (GPU-like)
  – Latency-optimized (Multicore CPU-like)

• A look at NVIDIA’s Fermi GPU architecture

• Musings on Future

• Assumed: basic computer architecture
The Graphics Pipeline
The Graphics Pipeline
The Graphics Pipeline

- Vertex Transform & Lighting
- Triangle Setup & Rasterization
- Texturing & Pixel Shading
- Depth Test & Blending
- Framebuffer
The Graphics Pipeline

- Remains a useful abstraction
- Hardware *used to* look like this
The Graphics Pipeline

- Hardware used to look like this
  - Vertex, pixel processing became programmable

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```
The Graphics Pipeline

- Hardware **used to** look like this
  - Vertex, pixel processing became programmable
  - New stages added

```c
// Each thread performs one pair-wise addition
__global__ void * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```
The Graphics Pipeline

- Hardware used to look like this
  - Vertex, pixel processing became programmable
  - New stages added

GPU architecture increasingly centers around shader execution

```c
// Each thread performs one pair-wise addition
__global__ void * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```
Modern GPUs: Unified Design

Vertex shaders, pixel shaders, etc. become threads running different programs on a flexible core
GPU Architecture: GT200 (Tesla)

- Host
  - Input Assembler
    - Vertex Thread Issue
      - Setup & Rasterize
        - Geom Thread Issue
          - Pixel Thread Issue

- Thread Scheduler
  - Input Assembler
  - Vertex Thread Issue
  - Geom Thread Issue
  - Pixel Thread Issue
What makes it fast?

• Massive number of independent work items (pixels)
  – Allows parallelism
  – Usually, coherent control
  – Except synchronization points
    • (FB writes need to appear ordered)
What makes it fast?

• Massive number of independent work items (pixels)

• High degree of data locality
  – Maintain much data on-chip (vertices, attributes, etc.) during processing of a triangle
  – Main sources of off-chip accesses: textures and FB
    • Very coherent: neighboring pixels are spatially adjacent, likely read the same texture regions etc.
    • Spatial adjacency also helps with FB writes
What makes it fast?

• Massive number of independent work items (pixels)

• High degree of data locality

• Custom scheduling and resource allocation
  – No need for software arbitration, thread launching, sync..
  – Efficient on-chip (L2$) allocation with ring buffers
  – Extremely lightweight thread launch/retire
  – When launching work, can always know ahead of time that result can be written out (no deadlocks)
What makes it fast?

• Massive number of independent work items (pixels)

• High degree of data locality

• Custom scheduling and resource allocation

• Fixed function units for common, expensive ops
  – E.g. anisotropic texture filtering (form of anti-aliasing for textured surfaces) reads tons of memory, performs highly nontrivial arithmetic
  – Much more power efficient than doing the same in SW
A Step Back

• The Graphics Workload...
  – Large number of independent but similar work items
  – Heavy on arithmetic (lots of math/memory op)
  – Coherent control, little data-dependent branching
  – Coherent memory accesses
A Step Back

• The Graphics Workload...
  – Large number of independent but similar work items
  – Heavy on arithmetic (lots of math/memory op)
  – Coherent control, little data-dependent branching
  – Coherent memory accesses

• Contrast this to
  – Long programs with serial dependencies
  – Complex data-dependent control and memory access patterns
  – Few independent work items
    • Not 2 million pixels
    • Think of an OS running {insert favorite productivity tool}
Digression: Computer Arch 101

• **Throughput**
  – Number of instructions completed per clock

• **Latency**
  – Number of clocks it takes to complete an instruction
    • All modern processors employ pipelines (latency > 1clk)
    • Also, need to wait for memory (even caches have latencies)

• **Not the same thing!**
  – Even if a FP MUL takes 4 cycles to complete, BUT you can issue one of them per clock, throughput is 1 MUL/clk
Different Workloads

• **Throughput computing**
  – Large number of independent but similar work items
  – Heavy on arithmetic (lots of math/memory op)
  – Coherent control, little data-dependent branching
  – Coherent memory accesses

• **Latency-sensitive computing**
  – Long programs with serial dependencies
  – Complex data-dependent control and memory access patterns
  – Few independent work items
    • *Not* 2 million pixels

• What do you do to maximize performance?
Physical Realities Today

• Clock speeds are not going up by much...
• ...and power consumption is superlinear in Hz
Physical Realities Today

• Clock speeds are not going up by much...
• ...and power consumption is superlinear in Hz

Unavoidable corollary:
Processors must be parallel

• Cheaper to do N/2 concurrent ops/clk in two units next to each other than N ops/clk in one unit
  – Not entirely free of problems either, though
Physical Realities Today, cont’d

• DRAM is slow
  – Latency is 100s of cycles
  – More speed is exponentially more expensive

• DRAM is bad with random access
  – Memory atom is large (32 bytes), need coalesced R/W
    • Strong pressure towards 64 byte atom

• DRAM is power hungry
  – An off-chip access may burn 1000x more power than reading off the register file (which is not free either!)
  – Especially bad for battery-powered systems
Physical Realities Today, cont’d

• DRAM is slow

• DRAM is bad with random access

• DRAM is power hungry

→ Need to minimize DRAM use, otherwise execution units are sitting idle waiting for data!
Dealing with DRAM, Approach 1

1. Get **locality** by large, fast on-chip caches ($)
2. Reorder instructions to hide latency
3. Use a few threads/core to further hide latency
   - Simultaneous Multithreading (SMT) e.g. HyperThreading(tm)
Dealing with DRAM, Approach 1

1. Get **locality** by large, fast on-chip caches ($)
2. Reorder instructions to hide latency
3. Use a few threads/core to further hide latency
   - Simultaneous Multithreading (SMT) e.g. HyperThreading(tm)

   - Good for workloads that reuse data
     - When cache is large enough to accommodate working set
     - Even with non-coherent access patterns
     - When this doesn’t hold, you wait
   - Tolerates unpredictable control by branch prediction

   - What about scaling in #threads?
Dealing with DRAM, Approach 2

1. Bite the bullet and always wait for it
   – But switch in other threads that have all the data they need for next instruction ("Simultaneous multithreading", SMT)
   – With enough threads, DRAM and pipeline latency is hidden
     • What is Enough? Need many times more threads than execution units (remember, latency is 100s of cycles)

2. Exploit locality by having individual threads explicitly co-operate through fast on-chip memory
Dealing with DRAM, Approach 2

1. Bite the bullet and wait for it
   - But switch in other threads that have all the data they need for next instruction ("Simultaneous multithreading", SMT)
   - With enough threads, DRAM and pipeline latency is hidden
     • What is Enough? Need many times more threads than execution units (remember, latency is 100s of cycles)

2. Exploit locality by having individual threads explicitly co-operate through fast on-chip memory

   ➡️ Allows execution units to be much simpler
   - No need for branch prediction, instruction reordering logic, register renaming, etc.
Multicore CPU:
Run ~10 Threads Fast

- Few processors, each supporting 1–2 hardware threads
- Large on-chip cache near processors for hiding latency
- Each thread gets to run instructions close to back-to-back
Manycore GPU: Run ~10,000 Threads Fast

- Hundreds of processors, each supporting hundreds of hardware threads

- On-chip memory near processors
  - Use as explicit local storage, allow thread co-operation

- Hide latency by switching between many threads
Different Philosophies

• Different goals produce different designs
  – GPU assumes workload is highly parallel
  – CPU must be good at everything, parallel or not

• CPU: minimize latency experienced by 1 thread
  – Lots of big on-chip caches
  – Extremely sophisticated control

• GPU: maximize throughput of all threads
  – Lots of big ALUs
  – SMT can hide latency, so skip the big caches
  – Also, exploit convergent program flow by sharing control (scheduling, instr. issue) over multiple threads (SIMD)
The Two Design Points

MC = Multi-Core            MT = Many-Threads

How Do The Designs Differ?

• Many-threads (GPU) approach
  – Many, many threads that exploit execution coherence => can keep many more ALUs hot
  – Doesn’t deal with data/control irregularity as well

• Multicore CPU approach
  – Tolerates irregularity
  – Worse for computation that doesn’t reuse data as much
    • I.e., loop over the data ~once
  – Doesn’t scale to large numbers of threads
    • Each thread needs to cache its working set
    • When too little $/thread, starts to deteriorate
Well Over 4 TFLOP/s Today
Questions?
Fermi Architecture (2010-12)
Fermi Architecture (2010-12)
Streaming Multiprocessor (SM)
Streaming Multiprocessor (SM)

- 16 SMs per Fermi chip
  - 32 “cores” per SM (512 total)
  - 64KB of configurable L1$ / shared memory

- Unified L2$ for all SMs (768 KB)
  - Fast, coherent data sharing across all cores in the GPU

<table>
<thead>
<tr>
<th></th>
<th>FP32</th>
<th>FP64</th>
<th>INT</th>
<th>SFU</th>
<th>LD/ST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ops / clk / SM</strong></td>
<td>32</td>
<td>16</td>
<td>32</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
SM Microarchitecture

- New IEEE 754-2008 arithmetic standard
- Fused Multiply-Add (FMA) for SP & DP
- New integer ALU optimized for 64-bit and extended precision ops
- Note: contains little else than functional units
SM Local Memory (L1 data$)

Bring the data closer to the ALU

- Minimize trips to external memory
- Share values between threads to minimize overfetch and computation
- Increases arithmetic intensity by keeping data close to the processors

- User managed generic memory, threads read/write arbitrarily
  - Challenging to implement, tens of concurrent independent reads and writes!
Fast Memory Interface

- GDDR5 memory interface
  - 2x improvement in peak speed over GDDR3
  - Practical max. throughput ~150GiB/s

- Allows up to 1 Terabyte of memory attached to GPU
  - Operate on large data sets
Programming Model

- Each SM is a set of **SIMT cores**
  - Single Instruction Multiple Thread

- Each core has
  - Program counter (PC), register file (RF), etc.
  - Scalar data path
  - Read/write memory access
Programming Model, cont’d

- Each SM is a set of **SIMT cores**
  - Single Instruction Multiple Thread

- Each core has
  - Program counter (PC), register file (RF), etc.
  - Scalar data path
  - Read/write memory access

- Unit of SIMT execution: **WARP**
  - Warp is a group of threads that execute the same instruction/clock
  - Each clock, scheduler picks one warp to be executed and dispatches it to all cores
  - __syncthreads() from last time will make sure all warps within a block wait before proceeding
SIMT Multithreaded Execution

- **Warp**: a set of \( N \) (currently 32) parallel threads that execute a SIMD instruction
- Warp is the basic scheduling unit
- SM hardware implements zero-overhead warp and thread scheduling
- Threads can execute independently
  - If half the threads take a branch and the others don’t, the latter get masked off during the execution of the former
- SIMD warp *automatically* diverges and reconverges when threads branch
- Best efficiency and performance when threads of a warp execute together
  - Happens often in practice
Programmer’s View of SM

<table>
<thead>
<tr>
<th>Warp 0</th>
<th>PC</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>...</th>
<th>thr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 2</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 3</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 4</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 5</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 6</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp $n$</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
</tbody>
</table>
Each thread has otherwise independent state, but it shares PC with other threads of warp.
Programmer’s View of SM: Execution

<table>
<thead>
<tr>
<th>SM</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>...</th>
<th>core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 0</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 1</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 2</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 3</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 4</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 5</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 6</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp n</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
</tbody>
</table>
Programmer’s View of SM: Execution

<table>
<thead>
<tr>
<th>SM</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>core</th>
<th>...</th>
<th>core</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = r2 * r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Warp 0</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 1</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 2</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 3</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 4</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 5</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 6</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Warp n</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
</tbody>
</table>
## Programmer’s View of SM: Execution

### SM

| Warp 0 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
|--------|----|---|----|----|----|----|----|----|----|----|     |    |
| Warp 1 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
| Warp 2 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
| Warp 3 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
| Warp 4 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
| Warp 5 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
| Warp 6 | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|
| ...    |    |   |    |    |    |    |    |    |    |    |     |    |
| Warp n | PC |thr| thr| thr| thr| thr| thr| thr| thr| thr| ... |thr|

**r1 = r2 * r3**

**read r2 and r3**
Programmer’s View of SM: Execution

SM

\[ r_1 = r_2 \times r_3 \]

\begin{array}{cccccccccc}
\text{Warp 0} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\text{Warp 1} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\text{Warp 2} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\text{Warp 3} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\text{Warp 4} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\text{Warp 5} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\text{Warp 6} & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\text{Warp } n & \text{PC} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \text{thr} & \ldots & \text{thr} \\
\end{array}
Programmer’s View of SM: Execution

<table>
<thead>
<tr>
<th>Warp 0</th>
<th>PC</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>thr</th>
<th>...</th>
<th>thr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 2</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 3</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 4</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 5</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>Warp 6</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp n</td>
<td>PC</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>thr</td>
<td>...</td>
<td>thr</td>
</tr>
</tbody>
</table>

r1 = r2 * r3

write to r1

core core core core core core core core core ... core
Programmer’s View of SM: Execution
# Programmer’s View of SM: Execution

<table>
<thead>
<tr>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
</tr>
</tbody>
</table>

| Warp 0 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| Warp 1 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| Warp 2 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| Warp 3 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| Warp 4 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| Warp 5 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| Warp 6 | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... |
| ...   | ...| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Warp $n$ | PC | thr | thr | thr | thr | thr | thr | thr | thr | ... | thr |
Programmer’s View of SM: Blocks

Note: Blocks are formed on the fly from the available warps (don’t need to consecutive)
Programmer’s View of SM: Blocks

Note: Blocks are formed on the fly from the available warps (don’t need to consecutive)
Incoherence Example

```c
if (a < 10)
    foo();
else
    bar();
```

```assembly
/*0048*/     ISETP.GT.AND P0, pt, R4, 0x9, pt;
/*0050*/     @P0 BRA 0x70;
/*0058*/     ...;       if branch
/*0060*/     ...;
/*0068*/     BRA 0x80;
/*0070*/     ...;       else branch
/*0078*/     ...;
/*0080*/     continue here after the if-block
```
Incoherence Example

**Case 1:** All threads take the *if* branch

```c
if (a < 10)
    foo();
else
    bar();
```

```assembly
/*0048*/     ISETP.GT.AND P0, pt, R4, 0x9, pt;
/*0050*/     @P0 BRA 0x70;  // no thread wants to jump
/*0058*/     ...;        if branch
/*0060*/     ...;
/*0068*/     BRA 0x80;
/*0070*/     ...;        else branch
/*0078*/     ...;
/*0080*/     continue here after the if-block
```
Incoherence Example

Case 2: All threads take the else branch

```c
if (a < 10)
    foo();
else
    bar();
```

/*0048*/     ISETP.GT.AND P0, pt, R4, 0x9, pt;
/*0050*/     @P0 BRA 0x70;  // all threads want to jump
/*0058*/     ...;    if branch
/*0060*/     ...;
/*0068*/     BRA 0x80;
/*0070*/     ...;    else branch
/*0078*/     ...;
/*0080*/     continue here after the if-block
Incoherence Example

Case 3: Some threads take the if branch, some take the else branch

```c
if (a < 10)
    foo();
else
    bar();
```

```asm
/*0048*/     ISETP.GT.AND P0, pt, R4, 0x9, pt;
/*0050*/     @P0 BRA 0x70;  // some threads want to jump: push
/*0058*/     ...;       // if branch
/*0060*/     ...;
/*0068*/     BRA 0x80;  // restore active thread mask
/*0070*/     ...;       // else branch
/*0078*/     ...;      // pop
/*0080*/     continue here after the if-block
```

continue here after the if-block
Memory in CUDA, part 1

• Global memory
  – Accessible from everywhere, including CPU (memcpy)
  – Requests go through L1, L2, DRAM

• Shared memory
  – Either 16 or 48 KB per SM in Fermi
  – Pieces allocated to thread blocks when launched
  – Accessible from threads in the same block
  – Requests served directly, very fast

• Thread Local memory
  – Actually a thread-local portion of global memory
  – Used for register spilling and indexed arrays
Memory in CUDA, part 2

• **Textures**
  – Data can also be fetched from DRAM through texture units
  – Separate texture caches
  – High latency, extreme pipelining capability
  – Read-only

• **Surfaces**
  – Read / write access with pixel format conversions
  – Useful for integrating with graphics

• **Constants**
  – Coherent and frequent access of same data
Simplified

• Global memory
  – Almost all data access goes here, you will need this

• Shared memory
  – Use to share data between threads

• Textures
  – Use to accelerate data fetching

• Local memory, constants, surfaces
  – Let’s ignore for now, details can be found in manuals
Memory Access Coherence

• GPU memory buses are wide
  – Both external and internal

• When warp executes a memory instruction, the addresses matter a lot
  – Those that land on the same cache line are served together
  – Different cache lines are served sequentially

• This can have a huge impact on performance
  – Easy to accidentally burden the memory system
  – Incoherent access also easily overflows caches
Improving Memory Coherence

• Try to access nearby addresses from nearby threads

• If each thread processes just one element, choose wisely which one

• If each thread processes multiple elements, preferably use striding
Striding Example

• We want each thread to process 10 elements of an array
  – 64 threads per block

Time

No striding
Thread 0: 0 1 2 3 4 5 6 7 8 9
Thread 1: 10 11 12 13 14 15 16 17 18 19
.. Thread 63: 630 631 632 633 634 635 636 637 638 639

With stride of 64
Thread 0: 0 64 128 192 256 320 384 448 512 576
Thread 1: 1 65 129 193 257 321 385 449 513 577
.. Thread 63: 63 127 191 255 319 383 447 511 575 639

Bad access pattern
Optimal access pattern
Questions?
Benefits of SIMT

• Supports all structured C++ constructs
  – If/else, switch/case, loops, function calls, exceptions
  – \texttt{goto} is a different beast – supported, but best to avoid

• Multi-level constructs handled efficiently
  – Break/continue from inside multiple levels of conditionals
  – Function return from inside loops and conditionals
  – Retreating to exception handler from anywhere

• You only need to care about SIMT when tuning for performance
Some Consequences of SIMT

• An if statement takes the same number of cycles for any number of threads > 0
  – If nobody participates it’s cheap
  – Also, masked-out threads don’t do memory accesses

• A loop is iterated until all active threads in the warp are done

• A warp stays alive until every thread in it has terminated
  – Terminated threads cause “empty slots” in warps
  – Thread utilization = percentage of active threads
Coherent Execution Is Great

• An `if` statement is perfectly efficient if either everyone takes it or nobody does
  – All threads stay active

• A loop is perfectly efficient if everyone does the same number of iterations

• Note: These are required for traditional SIMD
Incoherent Execution Is Okay

• Conditionals are efficient as long as threads usually agree

• Loops are efficient if threads usually take roughly the same number of iterations

• Much easier to program than explicit SIMD
  – SIMT: Incoherence is supported, performance degrades gracefully if control diverges
  – SIMD: performance is fixed, incoherence not supported
Striving for Execution Coherence

• Learn to spot low-hanging fruit for improving execution coherence

• Process input in coherent order
  – E.g., process nearby pixels of an image together

• Fold branches together as much as possible
  – Only put the differing part in a conditional

• Simple low-level fixes
  – Favor $f_{\text{min}} / f_{\text{max}}$ over conditionals
  – Bitwise operators sometimes help
Questions?
Why warps?

• Scheduling is easier
  – Much simpler to choose from 48 (say) active warps than >1000 threads

• Resource management is easier
  – Can allocate/deallocate register file, shared memory in chunks

• When convergent, it’s much more efficient to cache, decode and issue the instruction once instead of 32 times
SIMT vs. SIMD

- SIMD means one executes the same instruction on multiple pieces of data at a time

- In SIMT, the SM executes one warp per clock, meaning all cores in the SM execute the same instruction
  - Why is it not SIMD?
SIMT vs. SIMD

- SIMD means one executes the same instruction on multiple pieces of data at a time.

- In SIMT, the SM executes one warp per clock, meaning all cores in the SM execute the same instruction.
  - Why is it not SIMD?

- Crucial difference: in SIMT, each thread is scalar.
  - Much easier writing scalar threads that communicate through shared memory than explicitly managing wide SIMD.
  - Threads within warp can diverge and reconverge to utilize as much of control coherence as possible.
SIMT vs. SIMD, illustrated

Traditional SIMD thread
(think SSE/AVX/Altivec)

Scalar op  SIMD op
SIMT vs. SIMD, illustrated

Traditional SIMD thread (think SSE/AVX/Altivec)

It’s challenging to write programs that keep the SIMD lanes hot (active research done in this area)
SIMT vs. SIMD, illustrated

Traditional SIMD thread (think SSE/AVX/Altivec)

SIMT threads are scalar, but they work on many concurrent instances at once
=> better utilization
SIMT vs. SIMD, illustrated

Traditional SIMD thread (think SSE/AVX/Altivec)

SIMT threads are scalar, but they work on many concurrent instances at once => better utilization
Questions?
Physical Realities Today, cont’d

• We are already out of power
  – The socket gives ~200W
  – Even worse in mobile
Physical Realities Today, cont’d

• We are already out of power
  – The socket gives ~200W
  – Even worse in mobile

• Even if caches help to use less DRAM BW, moving data around is expensive even on-chip
  – Double-precision fused multiply-add (DFMA) takes ~45pJ
  – Reading and writing operands from register file ~50pJ
  – Moving operands along the wires to the ALU is not free
    • Moving 3x64 DP FP over 1mm ~10pJ

• It’s easy to burn more power schlepping data around than doing useful work!
  – Numbers are a few years old, but the general idea applies!
Consequences

• Strong push towards greater locality on all scales
  – On-chip memories keep growing..
  – ..and **local register files** keep data even closer to ALUs than main RF (see James Balfour, Billy Dally & co’s research)
Consequences

• Strong push towards greater locality on all scales
  – On-chip memories keep growing..
  – ..and local register files keep data even closer to ALUs than main RF (see James Balfour, Billy Dally & co’s research)

• Fixed function HW is ideal in terms of power
  – Data moves only over short distances
  – Precision can be adapted for particular need
    • ”Cheaper to do 13.7-bit fixed-point MUL than 32-bit FP”
  – Examples: texture fetch, video encode/decode
  – Tradeoff: greater rigidity
  – Crystal ball: Tons and tons of FF units, most of which are powered off almost all the time?
Related Observations

• Complex logic (instruction reordering, register renaming etc.) and caches burn power
  – Push towards lots of simple cores from other players too
  – E.g. Intel’s Knights Ferry

• But for the foreseeable future, we will have tasks that are not parallel and require low latencies
Related Observations

• Complex logic (instruction reordering, register renaming etc.) and caches burn power
  – Push towards lots of simple cores from other players too
  – E.g. Intel’s Knights Ferry

• But for the foreseeable future, we will have tasks that are not parallel and require low latencies =>

• Heterogeneous systems are seemingly the way to go
  – Ubiquitous CPU + GPU combo is one option
  – Ideally: Both latency and throughput cores on same chip
    • NVIDIA Tegra, Intel Sandy Bridge, AMD Fusion, Apple A5 etc.
Takeaways

• The world is parallel and needs to get even more so

• The GPU is a massively parallel, throughput-optimized processor
  – Latency hiding with many more threads than FUs
  – Design exploits coherent control and memory accesses
  – When problem is data parallel, runs really fast

• Multicore CPUs are parallel, but optimized for the latency of a single thread
  – Peak performance for tasks that cannot be parallelized
Thank You!