Graphics Processing Units (GPUs)

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.

Why Study GPUs?

- Most successful commodity accelerator
- GPUs combine two useful strategies to increase efficiency
  - Massive parallelism
  - Specialization
- Illustrates tension between performance and programmability in accelerators

Graphics Processors Timeline

- Till mid-90s
  - VGA controllers used to accelerate some display functions
- Mid-90s to mid-2000s
  - Fixed-function accelerators for the OpenGL and DirectX APIs
  - 3D graphics: triangle setup & rasterization, texture mapping & shading
- Modern GPUs
  - Programmable multiprocessors optimized for data-parallelism
    - OpenGL/DirectX and general purpose languages (CUDA, OpenCL, …)
  - Some fixed-function hardware (texture, raster ops, …)

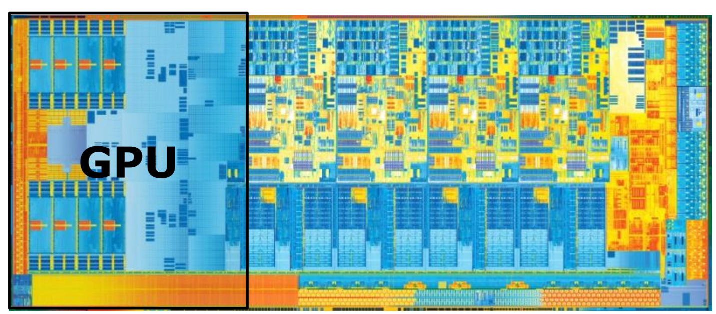
GPUs in Modern Systems

- Discrete GPUs
  - PCIe-based accelerator
  - Separate GPU memory

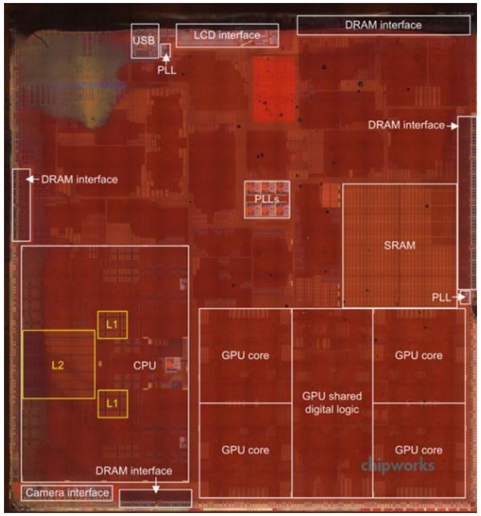
Nvidia Kepler

- Integrated GPUs
  - CPU and GPU on same die
  - Shared main memory and last-level cache

- Pros/cons?



Intel Ivy Bridge, 22nm 160mm^2

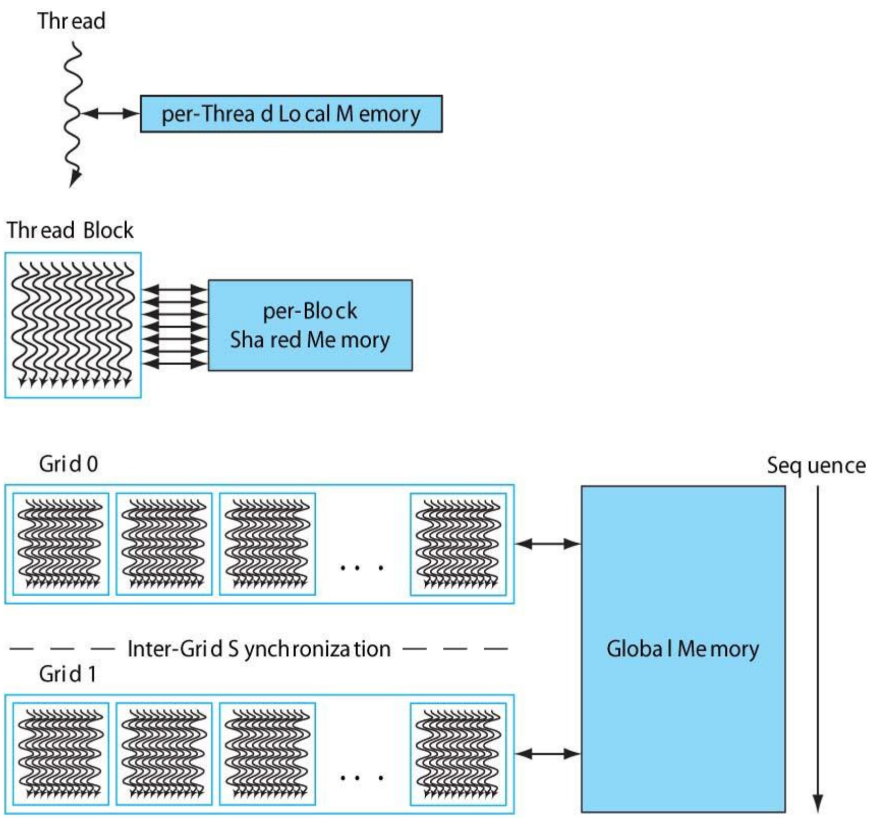


Apple A7, 28nm TSMC, 102mm^2

Our Focus

- GPUs as programmable multicores
  - Software model
  - Hardware architecture
- Good high-level mental model
  - GPU = Multicore chip with highly-threaded vector cores
  - Not 100% accurate, but helpful as a SW developer
- Will use Nvidia programming model (CUDA) and terminology (like Hennessy & Patterson)
  - If interested, ask me about pointers for AMD/ATI equivalents

CUDA GPU Thread Model



- Single-program multiple data (SPMD) model
- Each thread has local memory
- Parallel threads packed in blocks
  - Access to per-block shared memory
  - Can synchronize with barrier
- Grids include independent blocks
  - May execute concurrently

Code Example: DAXPY

C Code

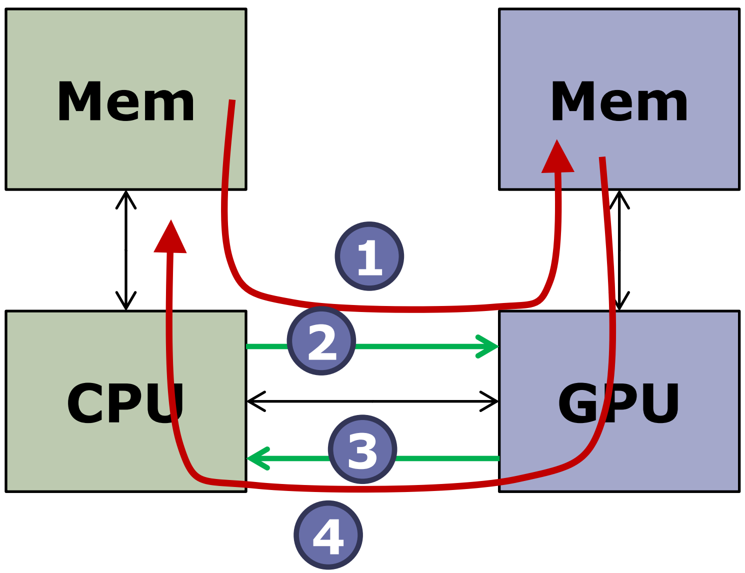
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
        for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

CUDA Code

```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
- CUDA vs vector terminology:
  - Thread = 1 iteration of scalar loop (1 element in vector loop)
  - Block = Body of vectorized loop (with VL=256 in this example)
  - Grid = Vectorizable loop

GPU Kernel Execution



1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space → no copies

GPU ISA and Compilation

- GPU microarchitecture and instruction set change very frequently

- To achieve compatibility:
  - Compiler produces intermediate pseudo-assembler language (e.g., Nvidia PTX)
  - GPU driver JITs kernel, tailoring it to specific microarchitecture

- In practice, little performance portability
  - Code is often tuned to specific GPU architecture

GPU Architecture Overview

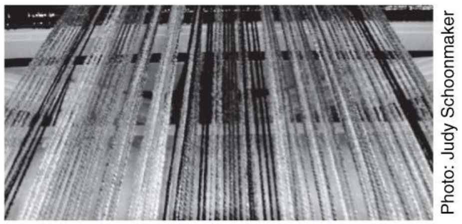
- A highly multithreaded multicore chip
- Example: Nvidia Kepler GK110

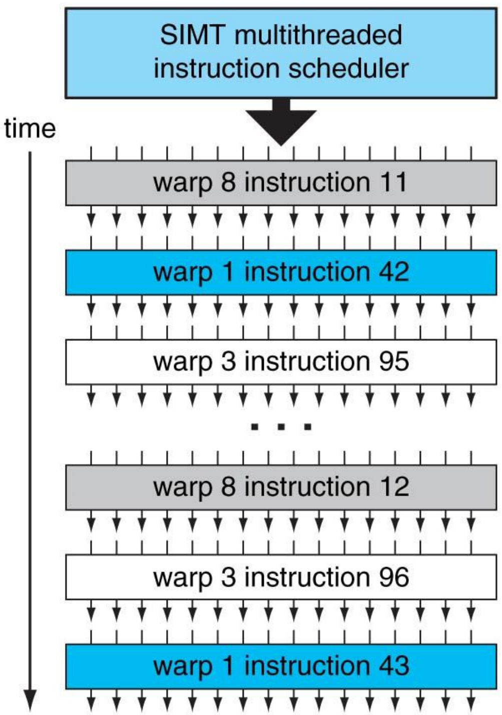
- 15 cores or streaming multiprocessors (SMX)
- 1.5MB Shared L2 cache
- 6 memory channels
- Fixed-function logic for graphics (texture units, raster ops, …)

- Scalability → change number of cores and memory channels

- Scheduling mostly controlled by hardware

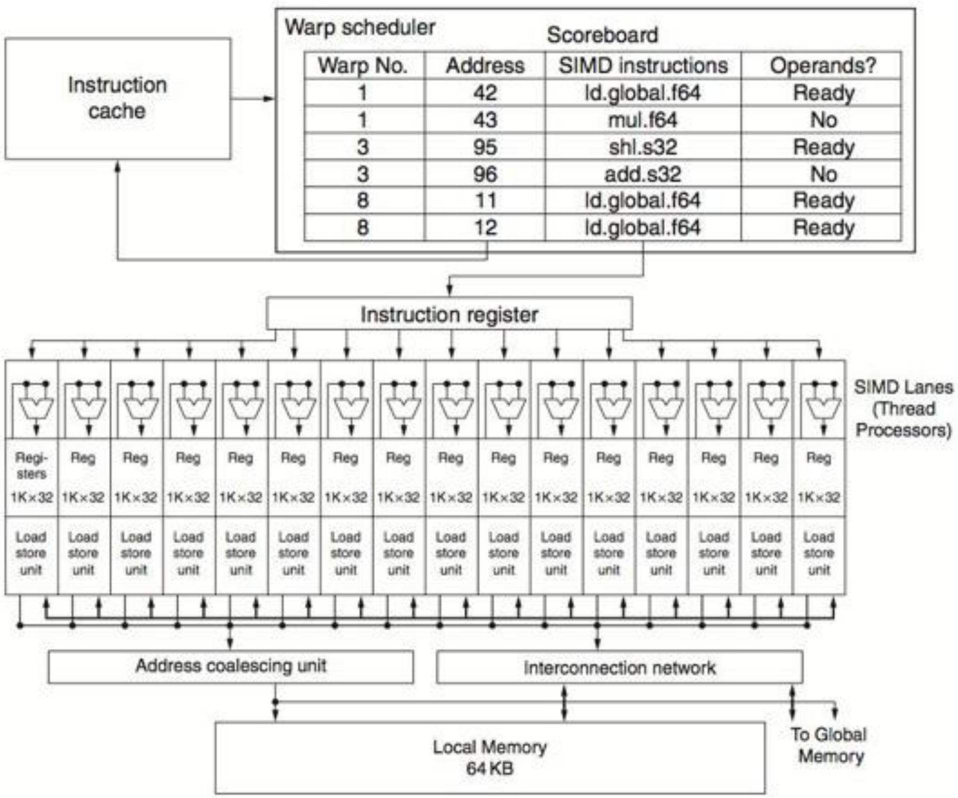
Instruction & Thread Scheduling: Thread + Data Parallelism





- In theory, all threads can be independent
- For efficiency, 32 threads packed in warps
  - Warp: set of parallel threads that execute the same instruction
    - Warp = a thread of vector instructions
    - Warps introduce data parallelism
  - 1 warp instruction keeps cores busy for multiple cycles
- Individual threads may be inactive
  - Because they branched differently
  - This is the equivalent of conditional execution (but implicit)
  - Loss of efficiency if not data parallel
- Software thread blocks mapped to warps
  - When HW resources are available

Streaming Multiprocessor Execution Overview



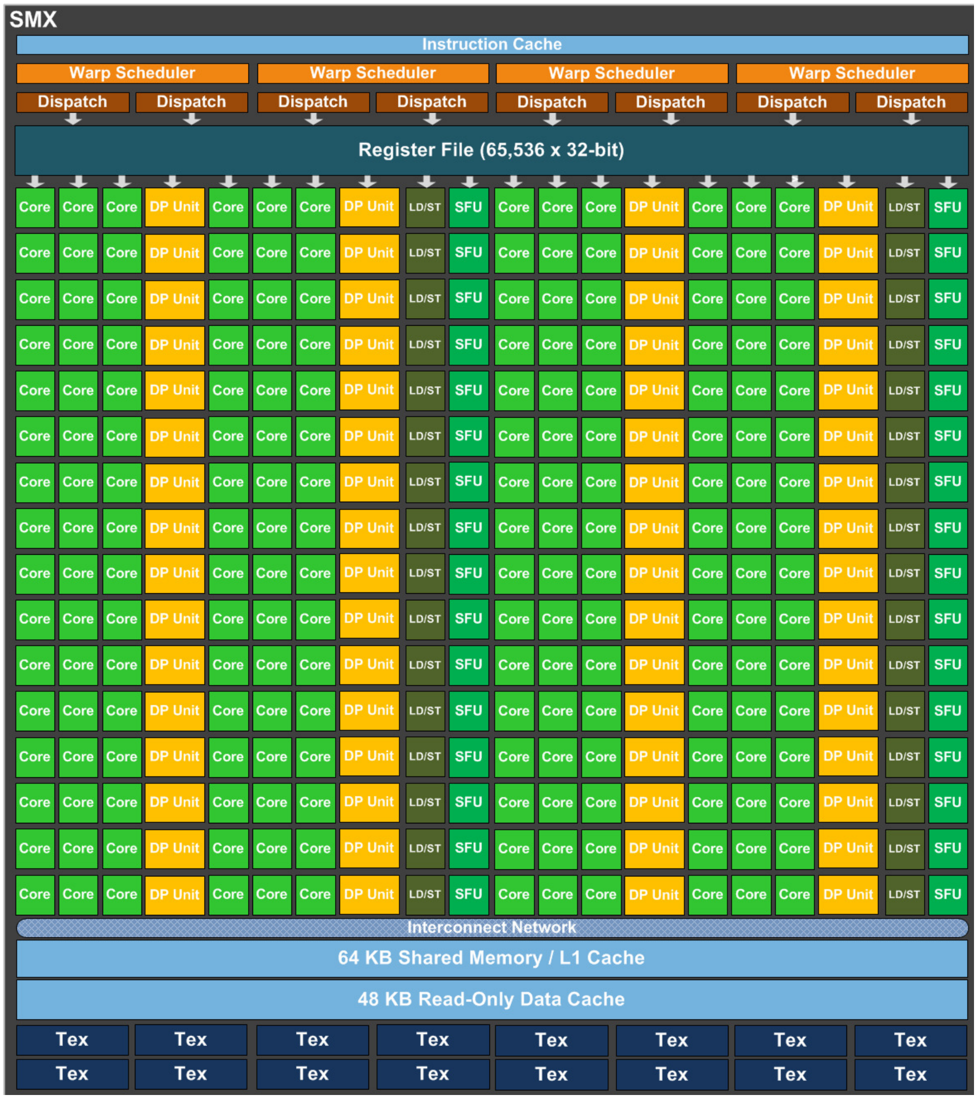
- Each SM supports 10s of warps (e.g., 64 in Kepler)
- Fetch 1 instr/cycle
- Issue 1 ready instr/cycle
  - Simple scoreboarding: all warp elements must be ready
- Instruction broadcast to all lanes
- Multithreading is the main latency-hiding mechanism

Context Size vs Number of Contexts

- SMs support a variable number of contexts based on required registers and shared memory
  - Few large contexts → Fewer register spills
  - Many small contexts → More latency tolerance
  - Choice left to the compiler
  - Constraint: All warps of a thread block must be scheduled on same SM

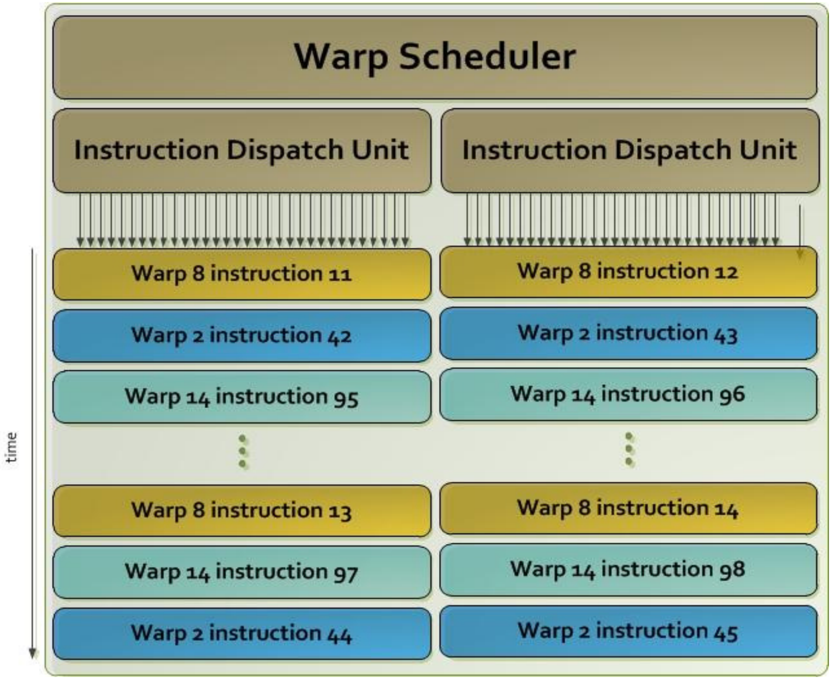
- Example: Kepler supports up to 64 warps
  - Max: 64 warps @ <=32 registers/thread
  - Min: 8 warps @ 255 registers/thread

Example: Kepler Streaming Multiprocessor



- Execution units
  - 192 simple FUs (int and single-precision FP)
  - 64 double-precision FUs
  - 32 load-store FUs
  - 32 special-function FUs (e.g., sqrt, sin, cos, …)
- Memory structures
  - 64K 32-bit registers
  - 64KB data memory, split between shared memory (scratchpad) and L1
  - 48KB read-only data/texture cache

Kepler Warp Scheduler & Instruction Dispatch



- Up to 64 warps per SM
- 32 threads per warp
  - 64K registers/SMX
  - Up to 255 registers per thread (if 8 warps)
- Scheduling
  - 4 schedulers select 1 warp/cycle
  - 2 independent instructions issued per warp
  - Total throughput = 4 * 2 * 32 = 256 ops per cycle
- Register scoreboarding
  - To track ready instructions
  - Simplified using static latencies from compiler

Handling Branch Divergence

- Similar to vector processors, but masks are handled internally
  - Per-warp stack stores PCs and masks of non-taken paths
- On a conditional branch
  - Push the current mask onto the stack
  - Push the mask and PC for the non-taken path
  - Set the mask for the taken path
- At the end of the taken path
  - Pop mask and PC for the non-taken path and execute
- At the end of the non-taken path
  - Pop the original mask before the branch instruction
- If a mask is all zeros, skip the block

Example: Branch Divergence

Assume 4 threads/warp,
initial mask 1111

```
if (m[i] != 0) {              1
  if (a[i] > b[i]) {          2
    y[i] = a[i] - b[i];
  } else {                    3
    y[i] = b[i] - a[i];
  }
                              4
} else {
  y[i] = 0;
                              5
}
```

1. Push mask 1111
   Push mask 0011
   Set mask 1100
2. Push mask 1100
   Push mask 0100
   Set mask 1000
3. Pop mask 0100
4. Pop mask 0011
5. Pop mask 1111

Memory Access Divergence

- All loads are gathers, all stores are scatters
- SM address coalescing unit detects sequential and strided patterns, coalesces memory requests
- Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!

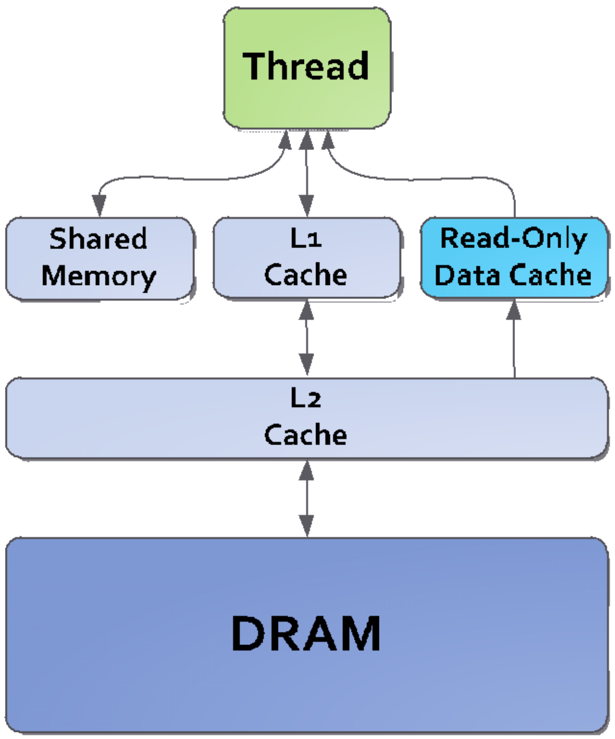
Memory System

- Per-SM caches and memories
  - Instruction and constant data caches
  - Multi-banked shared memory (scratchpad)
  - No inter-SM coherence
- Bandwidth-optimized main memory
  - Interleaved addresses
  - Aggressive access scheduling
  - Lossless and lossy compression (e.g., for textures)
- Per-thread private and global memories mapped to DRAM
  - Rely on multithreading to hide long latencies
- Recent GPUs feature a small shared L2
  - Reduce energy, amplify bandwidth
  - Faster atomic operations

Synchronization

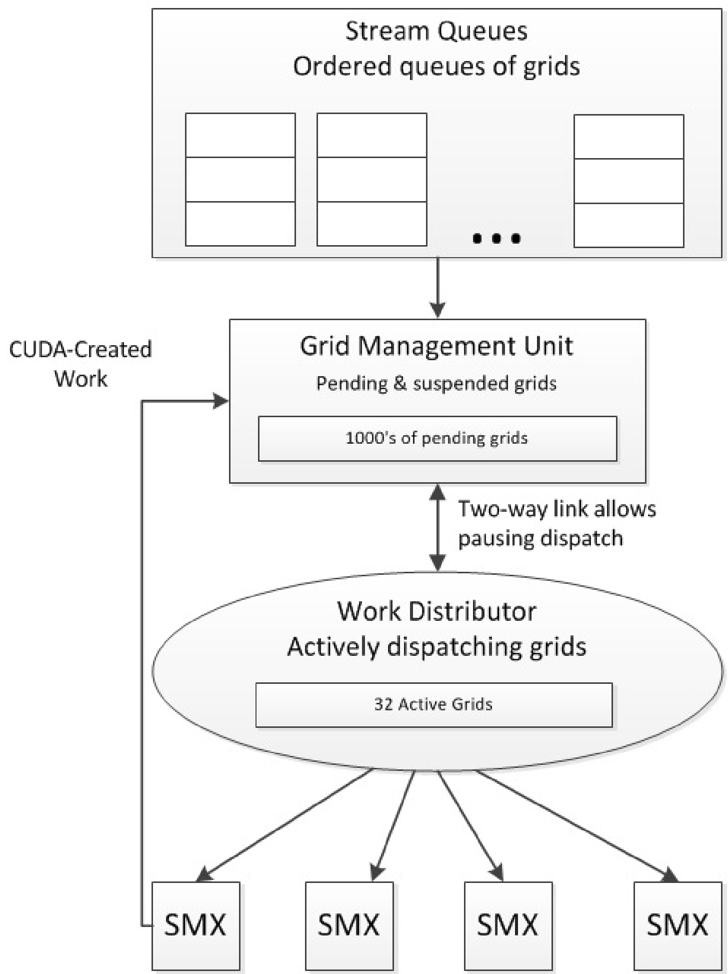
- Barrier synchronization within a thread block (__syncthreads())
  - Tracking simplified by grouping threads into warps
  - Counter tracks number of warps that have arrived to barrier
- Atomic operations to global memory
  - Read-modify-write operations (add, exchange, compare-and-swap, …)
  - More on these in Lecture 22
  - Performed at the memory controller or at the L2
- Limited inter-block synchronization!
  - Can't wait for other blocks to finish

Example: Kepler Memory Hierarchy



- Each SM has 64KB of memory
  - Split between shared mem and L1 cache
    - 16/48, 32/32, 48/16
  - 256B per access
- 48KB read-only data cache
- 1.5MB shared L2
  - Supports synchronization operations (atomicCAS, atomicADD, …)
  - How many bytes/thread?
- GDDR5 main memory
  - 384-bit interface (6x 64-bit channels) @ 1.75 GHz (x4 T/cycle)
  - 336 GB/s peak bandwidth

Hardware Scheduling



- HW unit schedules grids on SMX
  - Priority-based scheduling
- 32 active grids
  - More queued/paused
- Grids can be launched by CPU or GPU
  - Work from multiple CPU threads and processes

System-Level Issues

- Memory management
  - First GPUs had no virtual memory
  - Recent support for basic virtual memory (protection among grids, no paging)
  - Host-to-device copies with separate memories (discrete GPUs)

- Scheduling
  - Each kernel is non-preemptive (but can be aborted)
  - Resource management and scheduling left to GPU driver, opaque to OS

Vector vs GPU Terminology

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

[H&P5, Fig 4.25]