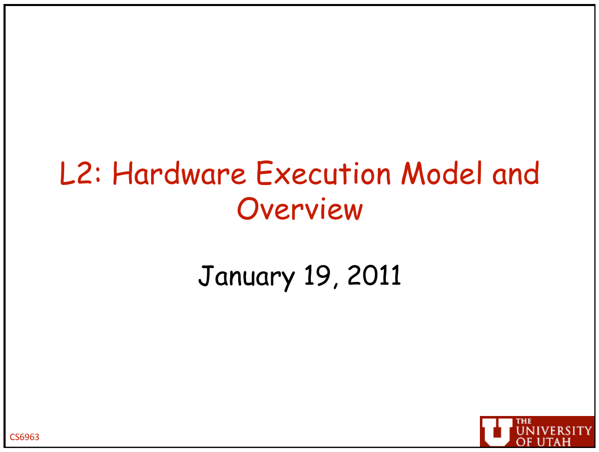
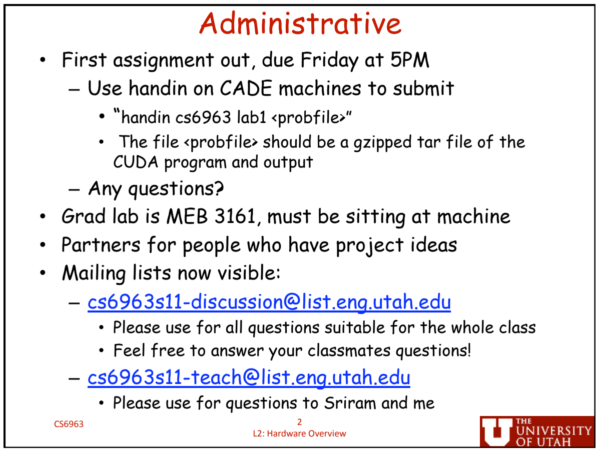


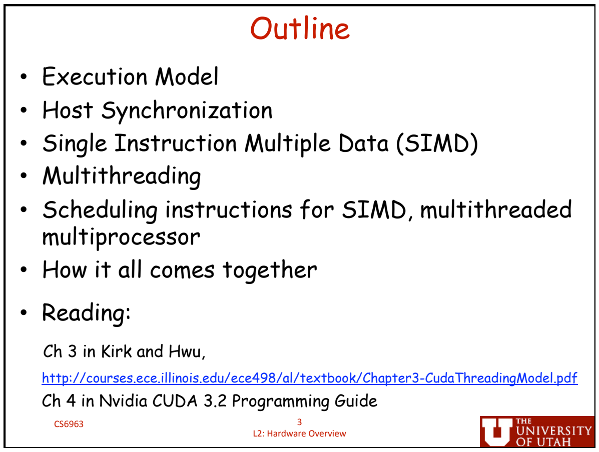
L2: Hardware Execution Model and Overview

January 19, 2011



Administrative

- First assignment out, due Friday at 5PM
  - Use handin on CADE machines to submit
    - "handin cs6963 lab1 <probfile>"
    - The file <probfile> should be a gzipped tar file of the CUDA program and output
  - Any questions?
- Grad lab is MEB 3161, must be sitting at machine
- Partners for people who have project ideas
- Mailing lists now visible:
  - cs6963s11-discussion@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!
  - cs6963s11-teach@list.eng.utah.edu
    - Please use for questions to Sriram and me



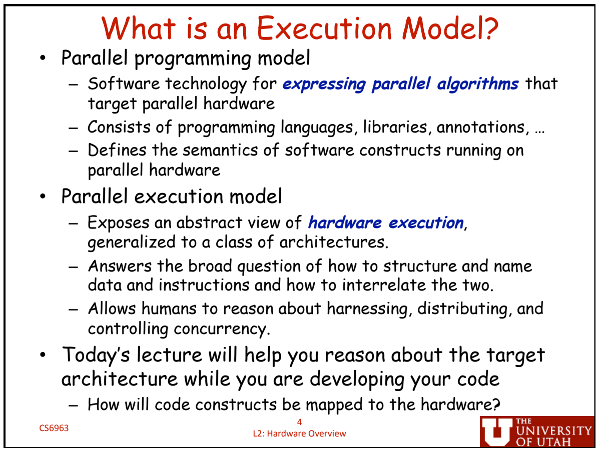
Outline

- Execution Model
- Host Synchronization
- Single Instruction Multiple Data (SIMD)
- Multithreading
- Scheduling instructions for SIMD, multithreaded multiprocessor
- How it all comes together
- Reading:

Ch 3 in Kirk and Hwu,

http://courses.ece.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

Ch 4 in Nvidia CUDA 3.2 Programming Guide



What is an Execution Model?

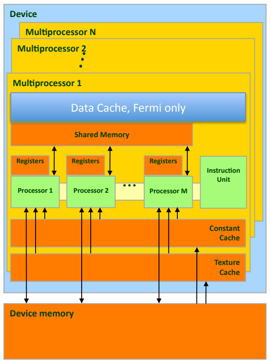
- Parallel programming model
  - Software technology for ***expressing parallel algorithms*** that target parallel hardware
  - Consists of programming languages, libraries, annotations, …
  - Defines the semantics of software constructs running on parallel hardware
- Parallel execution model
  - Exposes an abstract view of ***hardware execution***, generalized to a class of architectures.
  - Answers the broad question of how to structure and name data and instructions and how to interrelate the two.
  - Allows humans to reason about harnessing, distributing, and controlling concurrency.
- Today's lecture will help you reason about the target architecture while you are developing your code
  - How will code constructs be mapped to the hardware?

NVIDIA GPU Execution Model

I. SIMD Execution of warpsize=M threads (from single block)
   - Result is a set of instruction streams roughly equal to # blocks in thread divided by warpsize

II. Multithreaded Execution across different instruction streams within block
   - Also possibly across different blocks if there are more blocks than SMs

III. Each block mapped to single SM
   - No direct interaction across SMs



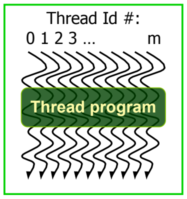
SIMT = Single-Instruction Multiple Threads

- Coined by Nvidia
- Combines SIMD execution within a Block (on an SM) with SPMD execution across Blocks (distributed across SMs)
- Terms to be defined…

CUDA Thread Block Overview

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Calling a Kernel Function – Thread Creation in Detail

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

Only for data that is not statically allocated

- Any call to a kernel function is asynchronous from CUDA 1.0 on
- Explicit synchronization needed for blocking continued host execution (next slide)

Host Blocking: Common Examples

- How do you guarantee the GPU is done and results are ready?
- Timing example (excerpt from simpleStreams in CUDA SDK):

```
cudaEvent_t start_event, stop_event;
cudaEventCreate(&start_event);
cudaEventCreate(&stop_event);
cudaEventRecord(start_event, 0);
 init_array<<<blocks, threads>>>(d_a, d_c, niterations);
 cudaEventRecord(stop_event, 0);
 cudaEventSynchronize(stop_event);
 cudaEventElapsedTime(&elapsed_time, start_event, stop_event);
```

- A bunch of runs in a row example (excerpt from transpose in CUDA SDK)

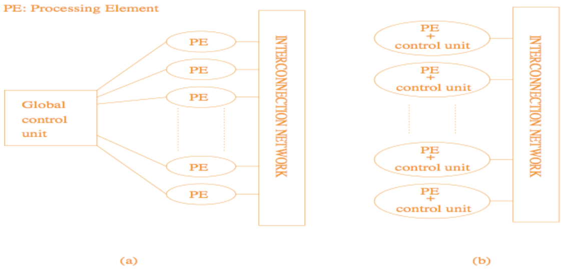
```
for (int i = 0; i < numIterations; ++i) {
        transpose<<< grid, threads >>>(d_odata, d_idata, size_x, size_y);
}
cudaThreadSynchronize();
```

Predominant Control Mechanisms: Some definitions

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 95: A[1:n] = A[1:n] + B[1:n] Kernel fns w/in block: compute<<<gs,bs,msize>>>
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	OpenMP parallel loop: forall (i=0; i<n; i++) Kernel fns across blocks compute<<<gs,bs,msize>>>

Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code:
		if ($threadIdx.x == 0) {
		}

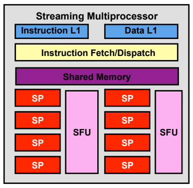
SIMD vs. MIMD Processors

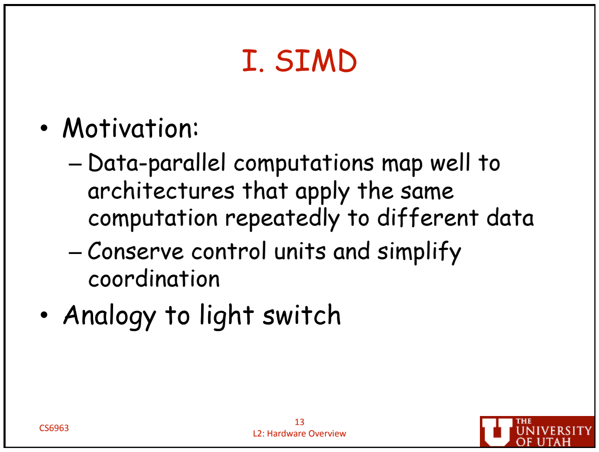


A typical SIMD architecture (a) and a typical MIMD architecture (b).

Streaming Multiprocessor (SM)

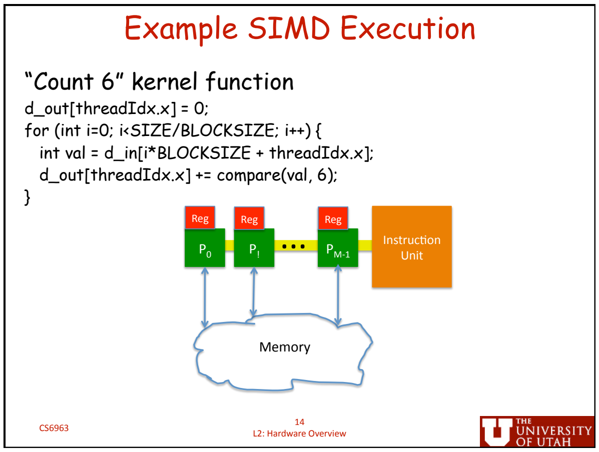
- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- DRAM texture and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





I. SIMD

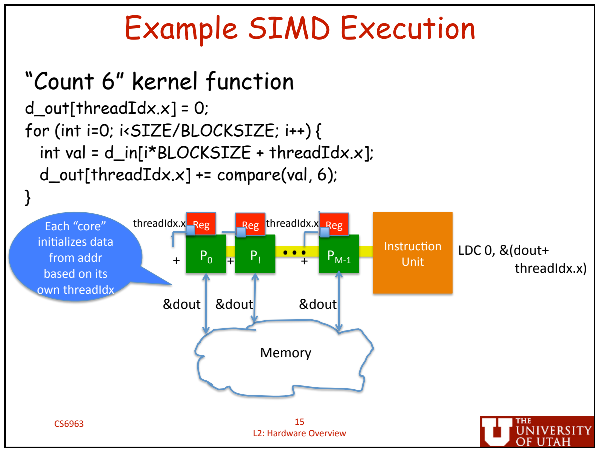
- Motivation:
  - Data-parallel computations map well to architectures that apply the same computation repeatedly to different data
  - Conserve control units and simplify coordination
- Analogy to light switch



Example SIMD Execution

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```



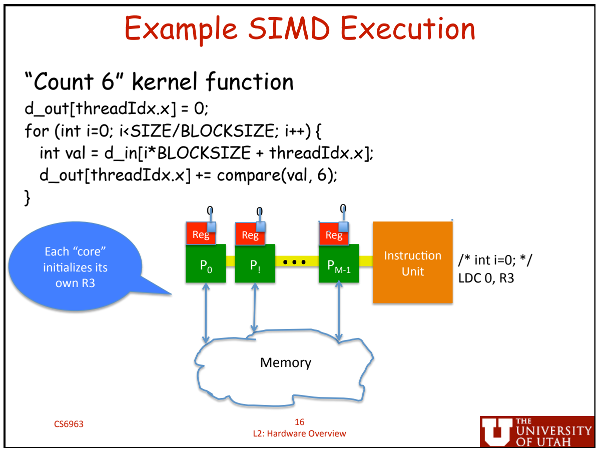
Example SIMD Execution

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```

Each "core" initializes data from addr based on its own threadIdx

LDC 0, &(dout+ threadIdx.x)



Example SIMD Execution

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```

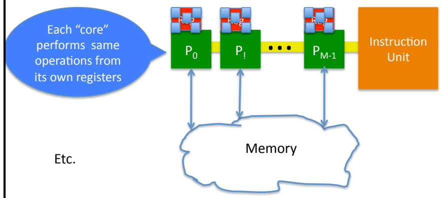
Each "core" initializes its own R3

/* int i=0; */
LDC 0, R3

Example SIMD Execution

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```



```
/* i*BLOCKSIZE
   + threadIdx    */
LDC BLOCKSIZE,R2
MUL R1, R3, R2
ADD R4, R1, RO
```

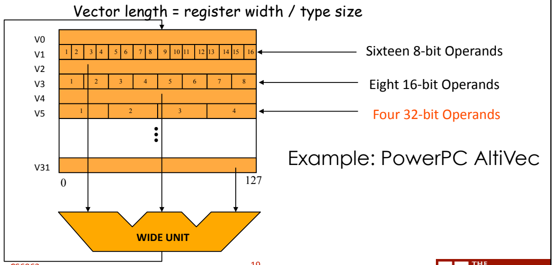
Overview of SIMD Programming

- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE-3
  - Graphics and games processors (example, IBM Cell)
  - Accelerators (e.g., ClearSpeed)
- Is there a dominant SIMD programming model?
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures were programmed by scientists
  - Multimedia extension architectures are programmed by systems programmers (almost assembly language!) or code is automatically generated by a compiler
  - GPUs are programmed by games developers (domain-specific)
  - Accelerators typically use their own proprietary tools

Aside: Multimedia Extensions like SSE-4

- COMPLETELY DIFFERENT ARCHITECTURE!
- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:

    Vector length = register width / type size

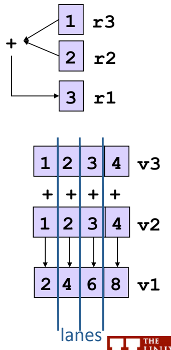


Example: PowerPC AltiVec

Aside: Multimedia Extensions Scalar vs. SIMD Operation

Scalar: `add r1,r2,r3`

SIMD: `vadd<sws> v1,v2,v3`



II. Multithreading: Motivation

- Each arithmetic instruction includes the following sequence

Activity	Cost	Note
Load operands	As much as O(100) cycles	Depends on location
Compute	O(1) cycles	Accesses registers
Store result	As much as O(100) cycles	Depends on location

- ***Memory latency,*** the time in cycles to access memory, limits utilization of compute engines

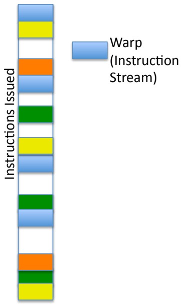
Thread-Level Parallelism

- Motivation:
  - a single thread leaves a processor under-utilized for most of the time
  - by doubling processor area, single thread performance barely improves
- Strategies for thread-level parallelism:
  - multiple threads share the same large processor reduces under-utilization, efficient resource allocation
    Multi-Threading
  - each thread executes on its own mini processor simple design, low interference between threads
    Multi-Processing

Slide source: Al Davis

What Resources are Shared?

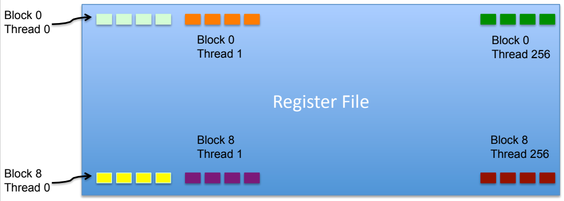
- Multiple threads are simultaneously active (in other words, a new thread can start without a context switch)
- For correctness, each thread needs its own program counter (PC), and its own logical regs (on this hardware, each thread w/in block gets its own physical regs)
- Functional units, instruction unit, i-cache shared by all threads



Aside: Multithreading

- Historically, supercomputers targeting non-numeric computation
  - HEP, Tera MTA, Cray XMT
- Now common in commodity microprocessors
  - Simultaneous multithreading:
    - Multiple threads may come from different streams, can issue from multiple streams in single instruction issue
    - Alpha 21464 and Pentium 4 are examples
- CUDA somewhat simplified:
  - A full warp scheduled at a time

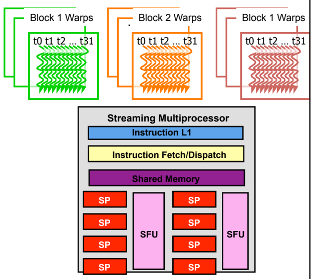
How is context switching so efficient?



- Large register file (16K registers/block)
  - Each thread assigned a "window" of physical registers
  - Works if entire thread block's registers do not exceed capacity (otherwise, compiler fails)
  - May be able to schedule from multiple blocks simultaneously
- Similarly, shared memory requirements must not exceed capacity for all blocks simultaneously scheduled

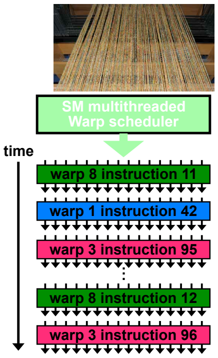
Example: Thread Scheduling on G80

- Each Block is executed as 32-thread Warps
  - 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



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

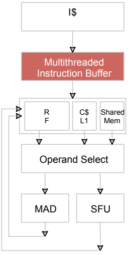
SM Warp Scheduling



- 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 minimum of 13 Warps are needed to fully tolerate 200-cycle memory latency

SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand ***scoreboarding*** used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp

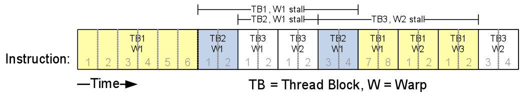


Scoreboarding

- How to determine if a thread is ready to execute?
- A ***scoreboard*** is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) they need
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation

Scoreboarding

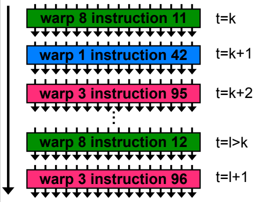
- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes ready after the needed values are deposited
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions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



TB = Thread Block, W = Warp

Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

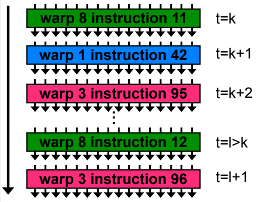


Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
…		

Schedule at time k

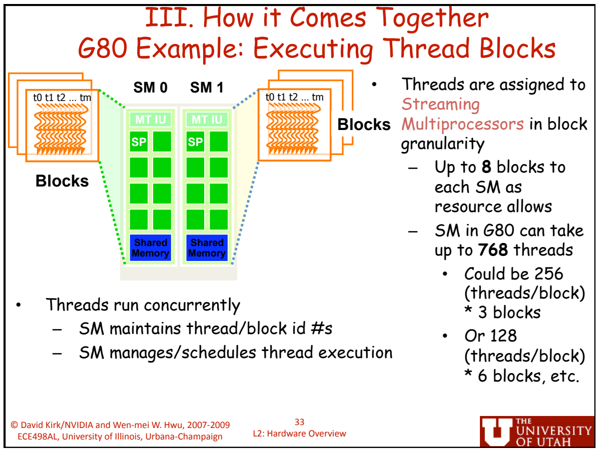
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8



Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

Schedule at time k+1



III. How it Comes Together
G80 Example: Executing Thread Blocks

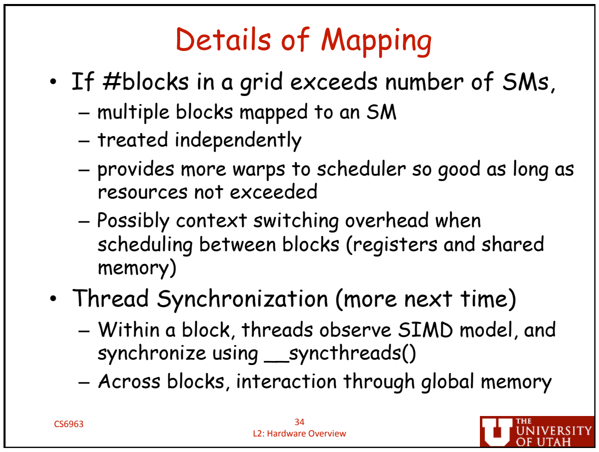
- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.

- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

33
L2: Hardware Overview

THE UNIVERSITY OF UTAH

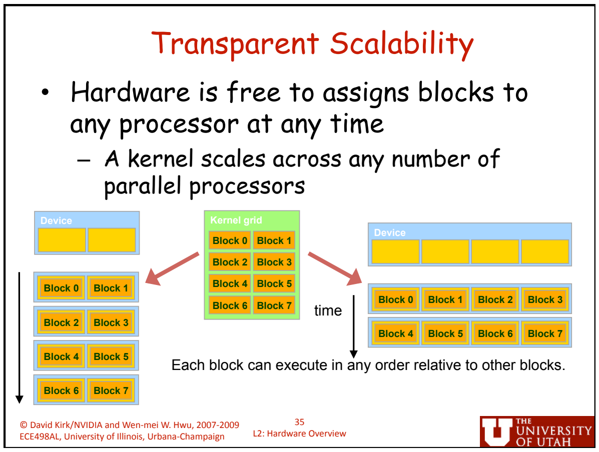


Details of Mapping

- If #blocks in a grid exceeds number of SMs,
  - multiple blocks mapped to an SM
  - treated independently
  - provides more warps to scheduler so good as long as resources not exceeded
  - Possibly context switching overhead when scheduling between blocks (registers and shared memory)
- Thread Synchronization (more next time)
  - Within a block, threads observe SIMD model, and synchronize using __syncthreads()
  - Across blocks, interaction through global memory

CS6963
34
L2: Hardware Overview

THE UNIVERSITY OF UTAH



Transparent Scalability

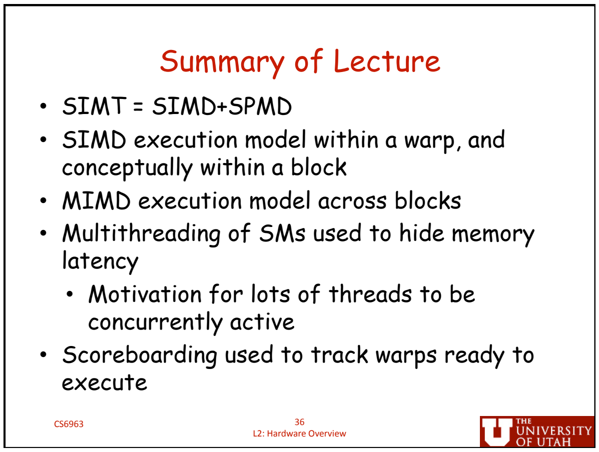
- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

35
L2: Hardware Overview

THE UNIVERSITY OF UTAH

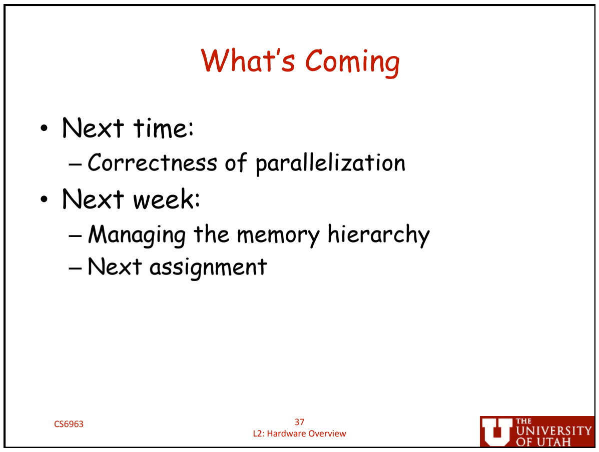


Summary of Lecture

- SIMT = SIMD+SPMD
- SIMD execution model within a warp, and conceptually within a block
- MIMD execution model across blocks
- Multithreading of SMs used to hide memory latency
  - Motivation for lots of threads to be concurrently active
- Scoreboarding used to track warps ready to execute

CS6963
36
L2: Hardware Overview

THE UNIVERSITY OF UTAH



What's Coming

- Next time:
  - Correctness of parallelization
- Next week:
  - Managing the memory hierarchy
  - Next assignment