GPU Architecture

Multi-threaded Architecture

Learning Objectives

1. Explain the difference between multithreading and context switching
2. Describe resource requirements for GPU multithreading

GPU Architecture

1. Many multiprocessors
2. Multithreading
3. Warp/wave front execution
4. Shared memory
5. Hardware caches

Multithreading

1. In an in-order processor, following instructions must stall on a cache miss
    - In an out-of-order processor, instructions only have to stall if there's a dependency
2. When multithreading, the processor can simply switch to another thread when one is stalled

Benefits of Multithreading

1. Hide processor stall time:
    - Cache misses
    - Branch instructions
    - Long latency operations (ALU operations)
2. GPUs use multithreading to hide latency
    - Out of order processors (OOO) use cache and ILP to hide latency
3. Longer memory latency requires a greater number of threads to hide latency

Front-end Extension for Multithreading

1. Multiple PC registers for warps (one PC for each warp)
2. One static instruciton for one warp
    - SPMD programming model
3. Individual registers for each thread
    - Minimizes context switch overhead
    - Significant resource

CPU Context Switching

1. CPU context switch: Store PC, architecture registers in stack/memory
2. High overhead of CPU context switching

Hardware Support for Multithreading

1. Front-end needs to have multiple PCs
    - One PC for each warp since all threads in a warp share the same PC
    - Later GPUs have other advanced features
2. Large register file
    - Each thread needs "K" number of architecture registers
    - Total register file size requirement = K * number of threads
    - "K" varies by applications
3. Remember occupancy calculation?

- Each SM can execute Y number of threads, Z number of registers, etc.
- Y is related to # of PC registers
- Z is related to K

Revisit Previous Occupancy Calculation Example

1. Hardware example: SM can execute 256 threads, 64K registers, 32 KB shared memory, warp size is 32
   - How many PCs are needed in one SM?
     - 256 / 32 = 8 PCs
   - If a program has 10 instructions, how many times does one SM fetch an instruction?
     - 10 * 8 = 80

Summary

1. GPU supports multithreading for performance enhancement
2. GPU hardware manages resources by warp for scalability

Bank Conflicts

Learning Objectives

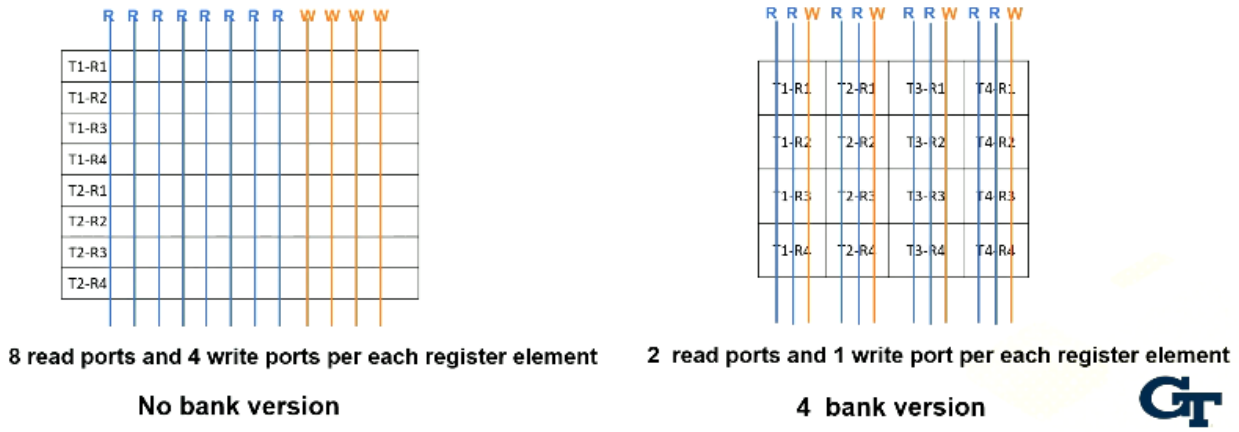
1. Explain SIMT behavior of register file accesses and shared memory accesses
2. Describe techniques to enhance bandwidth in register and shared memory accesses

CUDA Block/Threads/Warps

1. Multiple blocks can be running on one multiprocessor
2. Each block has multiple threads
3. A group of threads are executed as a warp
4. Registers are per thread
5. Execution width * 3 (2 source registers, 1 destination register) total register accesses

Port vs Bank

1. Port: Hadware interface for data access
   - Each thread requires 2 read and 1 write ports and execution width is 4
     - 8 read ports and 4 write ports
2. Bank: A partition (group) of the register file
   - Instead of placing 2 read and 1 write port per thread, place 2 read and 1 write per register element
   - Multiple banks can be accessed simultaneously
   - More ports mean more hardware wiring and resource usage



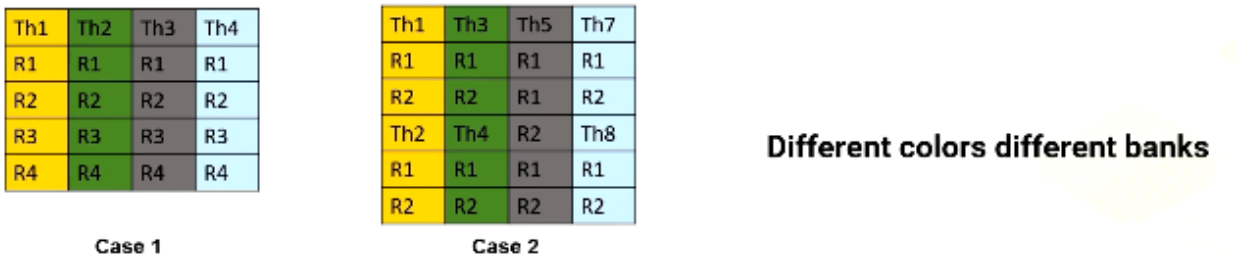
Port vs Bank

Bank Conflict

1. Scenario 1: Read R1 from T1, T2, T3, T4
   - Each thread on different banks, no conflict
2. Scenario 2: Read R1, R2, R3, R4 from T1
   - Bank conflict!
     - Can't read 8 values from any location, only 2 values from each bank
   - Takes multiple cycles

Variable Number of Registers per Thread

1. CUDA programming will get benefits from different register counts per thread
2. Instruction R3 = R1 + R2
   - Case 1: 4 registers per 1 thread
     - Reading registers would not cause a bank conflict
   - Case 2: 2 registers per 1 thread
     - Read R1, R2 from multiple threads would cause a bank conflict!
     - Remember: GPU executes a group of threads (warp), so multiple threads are reading the same registers



Bank Conflict

Solutions to Overcome Register Bank Conflict

1. Compiler-driven solution for optimizing code layout
   - Register ID is known at static time
2. Note: Not all register bank conflicts can be avoided
3. Complexity beyond a 5-stage pipline:

- Register file access takes more than 1 cycle
- Source register values are buffered

Static vs Dynamic

1. In this course, static often means before running code. The property is not dependent on input of the program. Dynamic means that the property is dependent on input of the program
   - e.g., static/dynamic number of instructions
     - LOOP: ADD R1, R1, 1
     - BREQ R1, 10, LOOP
2. Let's say that a loop iterates 10 times. Static number of instructions is 2, dynamic number of instructions is 20
   - Static time analysis = compile time analysis

Scoreboarding

1. Widely used in CPU to enable out of order execution
2. Dynamic instruction scheduling
3. GPUs: check when all source operands within a warp are ready; the warp is sent to execution units
   - Choose which one to send to the execution unit among multiple warps
     - e.g., oldest first

Reading Register Values

1. Reading register files might take a few cycles
2. Ready register values are stored at a buffer

Shared Memory Bank Conflicts

1. Shared memory is an on-chip storage
   - Scratch pad memory
   - Also composed with banks
2. Let's assume the following shared memory:
   - 4 banks; number is memory address
     - **shared** float shared_input[index1];
     - index1 = threadIdx.x * 4;
   - Then thread1:mem[4], thread2:mem[8], thread3:mem[12], ...
     - All threads will generate bank conflicts
   - Solution: Change the software structures (covered more in later lectures)

Summary

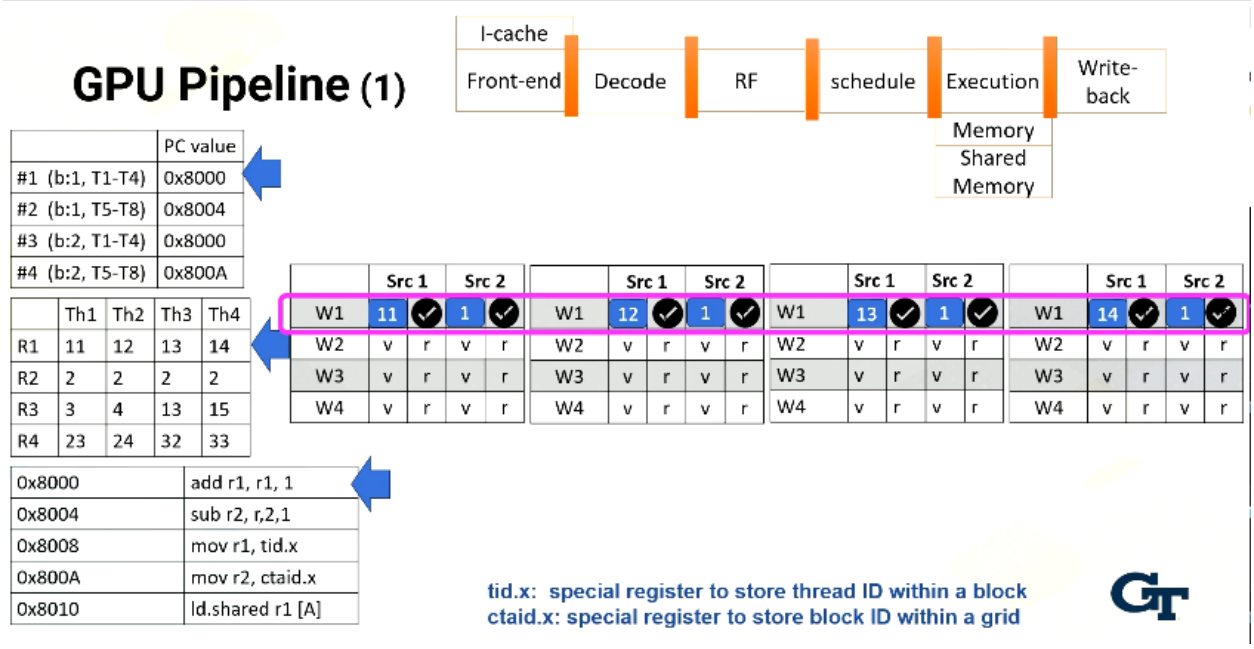
1. Learned the benefits of banks in register and shared memory
2. The reasons of bank conflicts in register files and shared memory

GPU Architecture Pipeline

Learning Objectives

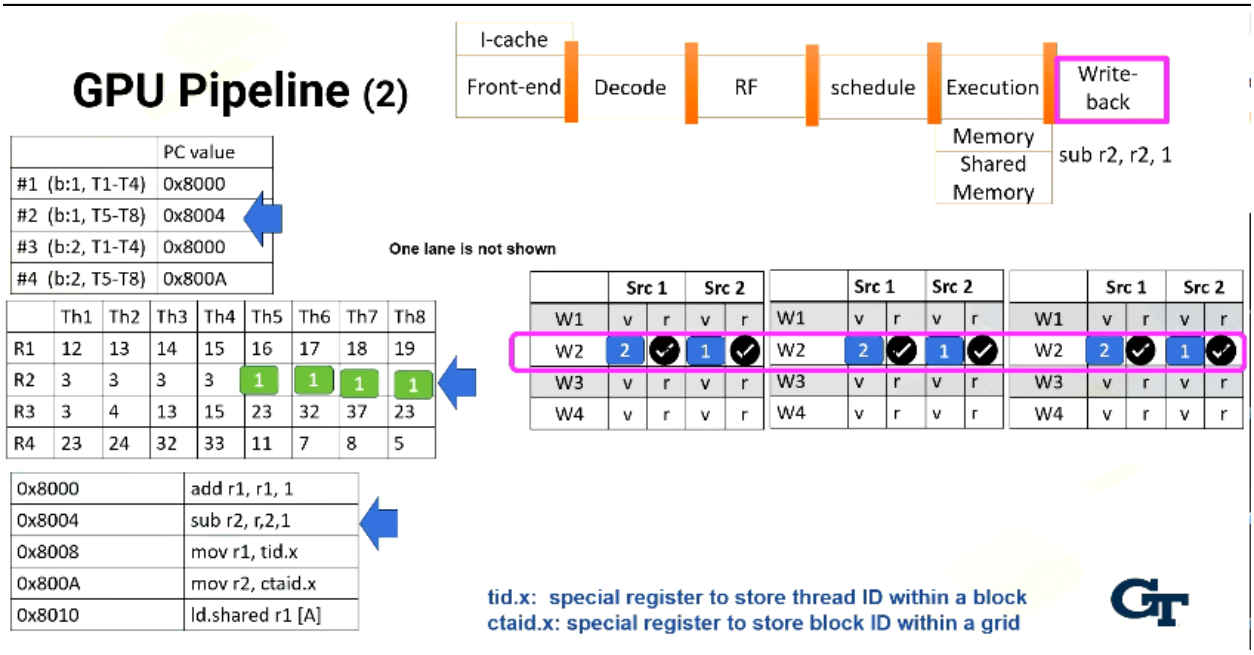
1. Describe GPU pipeline behavior with multithreading and register file access
2. Explain how mask bits are used

GPU Pipeline 1



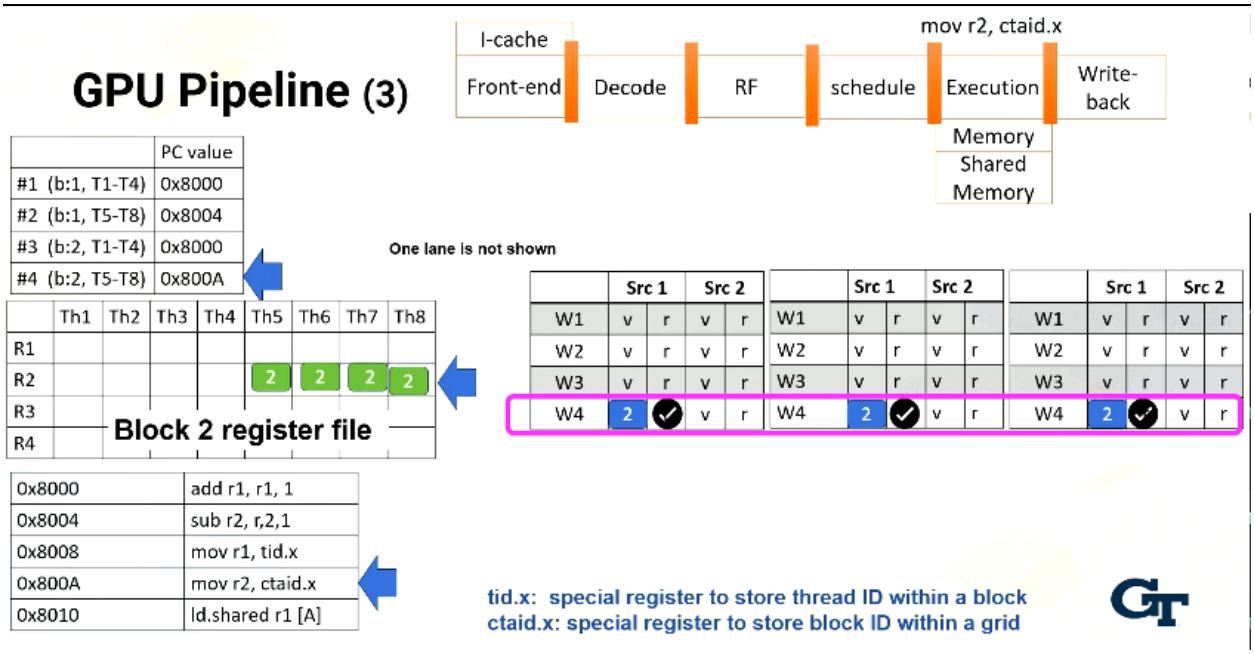
GPU Pipeline 1

GPU Pipeline 2



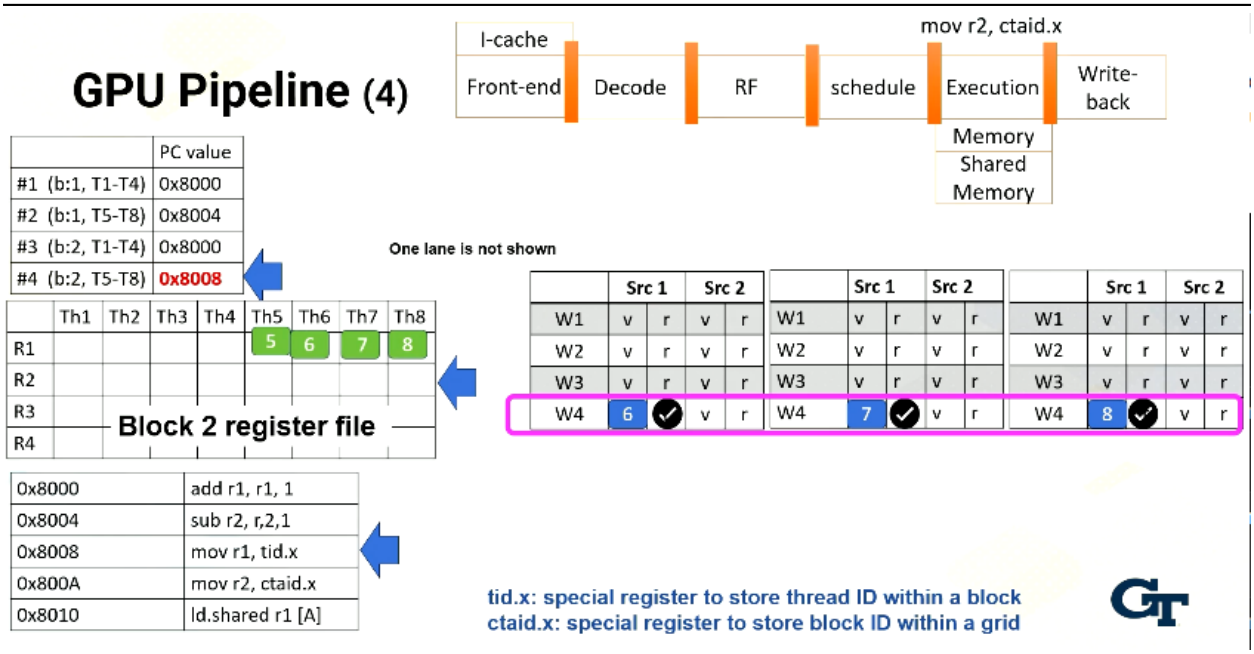
GPU Pipeline 2

GPU Pipeline 3



GPU Pipeline 3

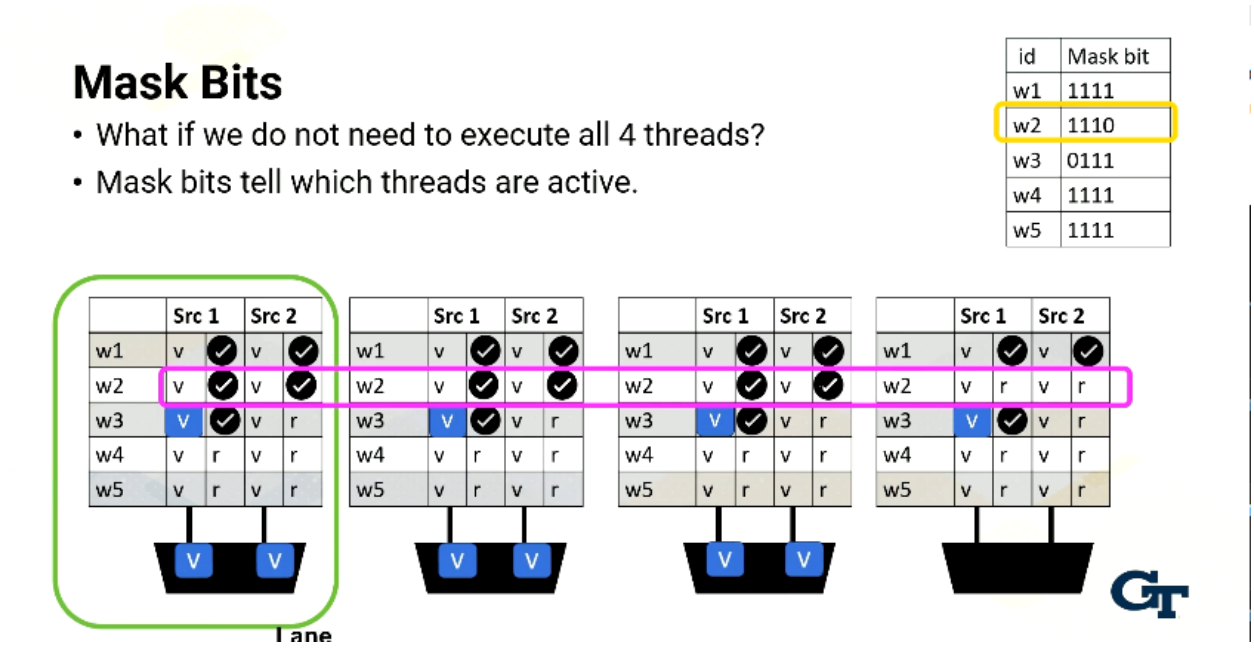
GPU Pipeline 4



GPU Pipeline 4

Mask Bits

1. What if we do not need to execute all 4 threads?
   - Mask bits tell us which threads are active



Mask Bits

Summary

1. Reviewed the GPU pipeline's instruction flow
2. Highlighted the use of special registers for thread ID and block ID
3. Introduced the concept of an active mask for identifying active SIMT lanes

Global Memory Coalescing

Learning Objectives

1. Explore global memory accesses
2. Explain memory address coalescing
3. Describe how one warp can generate multiple memory requests

Global Memory Accesses

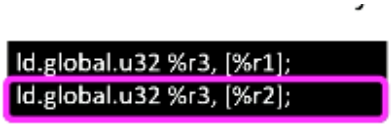
1. 1 memory instruction could generate many memory requests to DRAM
2. 1 warp can generate up to 32 memory requests (if warp size is 32)
3. Number of requests can easily be a large number
   - 32 SMs -> 32 * 32 = 1024 requests in one cycle
   - 1024 * 64B = 64KB per cycle (1 GHz GPU) -> 64 TB/s memory bandwidth

DRAM vs SRAM

1. SRAM: 6 transistors
   - Used in caches
   - Maintains charge over time
2. DRAM: 1 transistor (capacitor)
   - Loses charge over time, must be refreshed
3. GPU with High-Bandwidth Memory (HBM)
   - 3D-stack of DRAM with silicon interposer to interface
   - HBM provides significant memory bandwidth

Memory Coalescing

1. Even if the memory can provide high bandwidth memory, reducing memory requests is critical
2. GPU cache is very small
3. DRAM memory requests size is 64 ~ 128B
4. All memory requests from the first load can be combined into one memory request
    - mem[0] + 28 -> called coalesced
    - Second load cannot be easily combined -> uncoalesced



	Th1	Th2	Th3	Th4	Th5	Th6	Th7	Th8
R1	0	4	8	12	16	18	20	24
R2	0	128	256	512	768	1024	1152	1280
R3								
R4								

Memory Coalescing

Coalesced Memory

1. Combining multiple memory requests into a single or more efficient memory request
2. Consecutive memory requests can be coalesced
3. Reduce the total number of memory requests
4. One of the key software optimization techniques

Summary

1. Uncoalesced global memory requests significantly degrade performance
2. GPUs have the potential to saturate memory bandwidth
3. Coalescing memory requests is crucial for efficient memory access and better GPU performance