GPU Architecture Optimizations 1

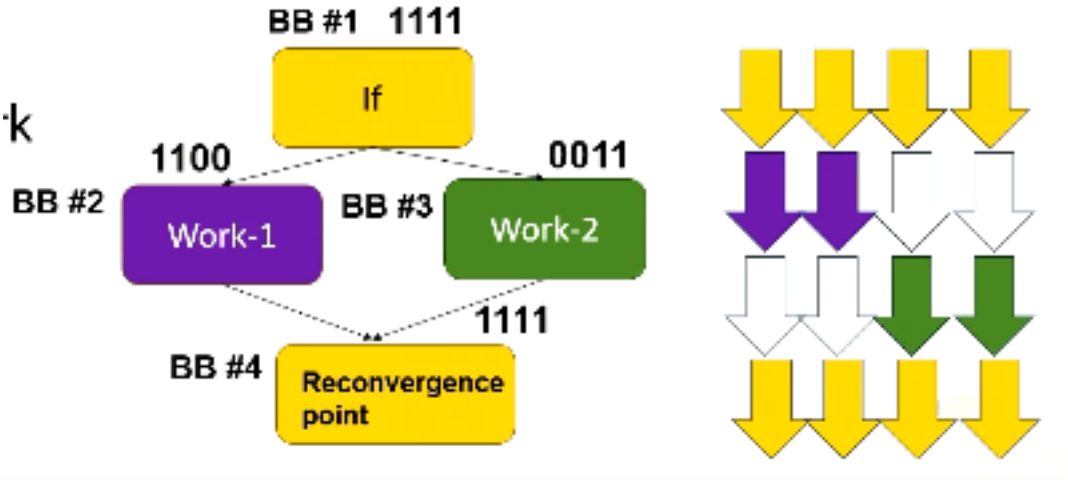
Handling Divergent Branches

Learning Objectives

1. Explain the definition of divergent warps
2. Apply hardware techniques to handle divergent warps

Divergent Branches

1. Warp: A group of threads that are executed together
2. Within a warp, not all threads need to be executed (if-else statements)
   - Not all threads will do the work
   - Use active mask
     - BB: Basic block
     - Reconvergence point: Immediate Post Dominator



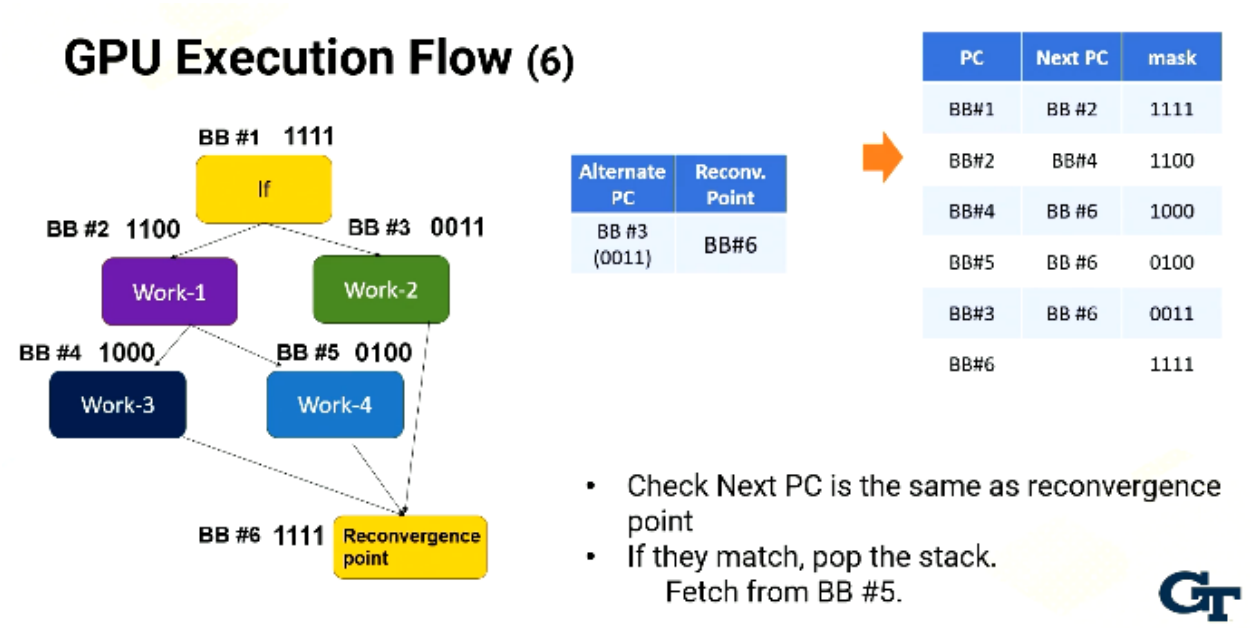
Divergent Branches

If-else Conversion: Predicated Execution

1. If-else conversion
   - Predicate the work (do the work on both branches)
   - Eliminate a branch
   - Commonly used in vector processors
   - Compiler optimization technique is called if-else conversion

GPU Execution Flow

1. Detecting divergent branch -> Not all PCs are the same within a warp
2. Store both PC addresses in a stack
3. Let's assume we go to path one first
   - How do we know when to fetch BB #3?
   - Store alternative PC and then go to BB3 after BB2
   - What if there are nested branches?
     - Similar to handling a function call
       - Use a stack (SIMT) to keep track of alternative PC and reconvergence points



GPU Executiion Flow

How to Identify the Reconvergence Point

1. Compile-time analysis
   - Static time analysis (control flow graph analysis)
   - Difficult to detect divergent branch information (very difficult)
2. Typical hardware implementation is done with a stack
   - Hardware can manage stack
   - Compiler can insert explicit stack operations: push and pop

Large Warp

1. The width of warp is a microarchitecture feature
2. Benefits of large warp:
   - One instruction's fetch/decode can generate many executions: 32, 64, 128, etc
   - Large width -> wide execution machine
     - Wide vector processors
3. Downside:
   - Chance of divergence is high
4. Application trade-off
   - Some applications have more uniform execution width
   - Some applications might have frequent divergent branches

Dynamic Warp Formation

1. Once the divergent warps are split, the execution utilization is very low
   - e.g., only half of the execution units are untilized
2. Regroup different threads from different warps
   - Same program PC: same execution
   - Dynamic warp formation
   - Hardware mechanisms
3. Many optimizations were proposed to increase the efficiency
4. Avoiding register file conflict is one big challenge

Summary

1. Reviewed the concept of divergent branches
2. Studied how GPU fetches divergent branches
3. SIMT stack is studied
4. Large warps, dynamic warp formation concepts are reviewed

Register Optimizations

Learning Objectives

1. Explain the challenges of register file sizes and opportunities
2. Discuss various optimizations techniques aimed at increasing the size of register files effectively

Register File Challenges

1. Each SM has a large number of threads, each of which has many registers
   - # of threads * # of registers -> 64KB, 128KB, 256KB, etc.
2. Large size (space), bandwidth and latency are issues
   - Read bandwidth = execution bandwidth
3. Optimization Methods
   - Access latency -> Hierarchical approach to provide different access latency
   - Size -> Reduce overall size through resource sharing

Observation #1: Not All Threads are Active

1. SPMD Programming Model
   - All threads have the same register files
2. GPU execution time is very asynchronous
3. Varying thread completion time
   - Some threads might be finishing early but some threads finish very late
   - Finished threads' register files are no longer needed
4. Different program paths result in diverse register file usages
   - Register file usages are different at run-time

Observation #2: Not All Registers are Live

1. Live/Dead register concept
   - add R1, R2, #1 <- R2 is no longer live
   - mov R2, #10
2. GPU's multiple threads may have long gaps between register usage
   - CPU is very short
   - Next instruction might be executed 10s of warp instruction later
     - Depends on execution pattern

Observation #3: Short Distance between Instructions

1. Add instruction can get results for previous instruction
   - move r2, #10
   - add r3, r2, #20 <- R2 is used immediately
2. Combining with an instruction scheduler, the hardware can reduce the distance (time between move and add) can be even shorter

Observation #4: Most Register Values are Read-Only

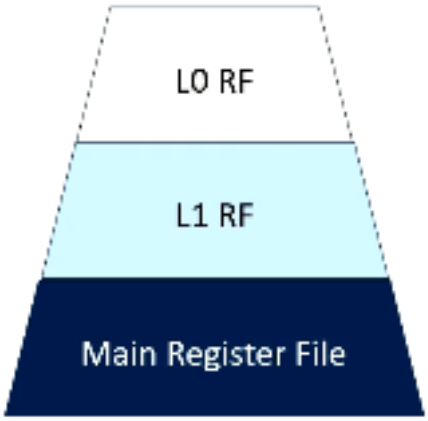
1. Many instructions use register values only once
   - mov r2, #10

- add r2, r2, #20 <- R2 is read by add and re-written

2. Is it necessary to store them in a register file?

Optimization #1: Hierarchical Register Files

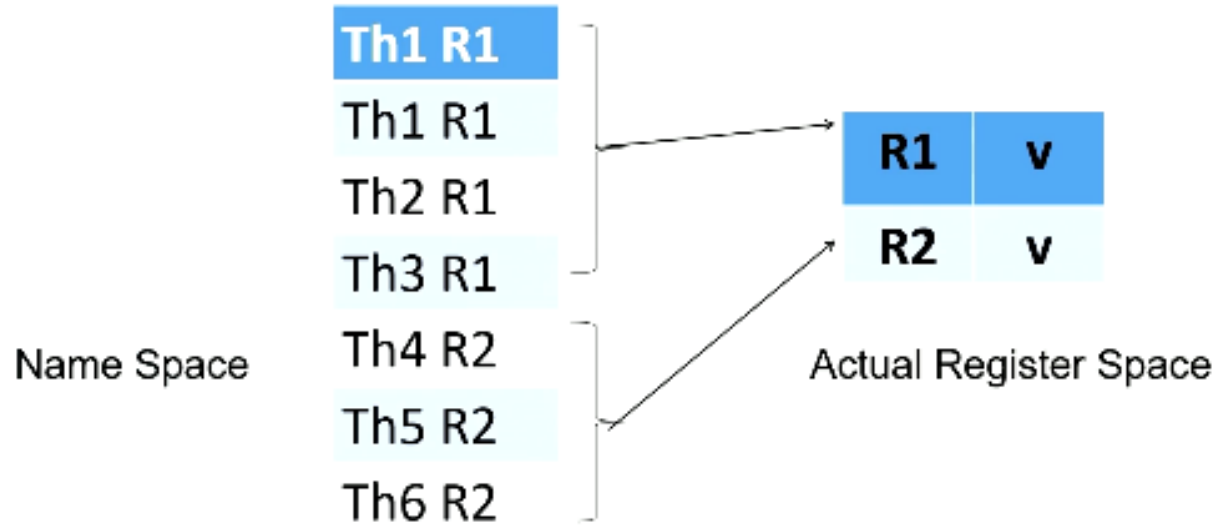
1. Register file cache
   - Small size of cache to cache the register files
   - Most register values can be cached and used once
   - The main register file size can be smaller or the same
2. Partition register file
   - Slow and Fast register segments
   - Slow and large register file can be implemented with NVM
     - NVM: non-volatile memory



Hierarchical Register Files

Optimization #2: Register File Virtualizations

1. Virtualization ~= Resource sharing
2. Only name space needs to be maintained



Register File Virtualizations

Summary

1. Not all registers are actively used
2. Studied register file optimization techniques to increase the resource sharing and to improve the latency

Unified Register File and Shared Memory

Learning Objectives

1. Explain optimization opportunities for unified memory and register file
2. Differentiate between register files and memory
3. Review the concept of occupancy in GPU programming

Review Occupancy

1. Resource constraints affect the maximum number of blocks on each SM
   - Shared memory requirements or register file size
   - Example: In HW, each SM has 64K registers, no shared memory
     - SW case 1: Each block has 64 registers, no shared memory
       - Shared memory is underutilized
     - SW case 2: Each block has 4 registers, 2KB shared memory
       - Register file size is underutilized (4 * cuda blocks * threads)
   - What if shared memory and register files are sharing the same resource?

Hardware Cache vs Software-Managed Cache

1. Hardware-managed cache: Hardware decides which memory contents will be cached
   - Address tag bits are needed to check whether an address is in the cache or not
2. Software-managed cache: Programmer specifies
   - Address tag bits are not needed: all software managed
   - Shared memory is already in a cache

Register File vs Software Managed Cache

1. Software managed cache: On-chip memory is small

2. Common characteristics:
    - Both are storing data only. No need to have a tag for address.
    - Both, location indicates the address
3. Differences
    - Number of read and write ports
    - Register is typically 2 source operands per instruciton vs memory is typically one address
    - Register file access patterns are known at compile time (static)

Proposed Solution: Unified Structures

1. Use the same hardware for shared memory and register files, creating a unified structure
2. Requirements: High memory bandwidth
    - Sufficient ports
    - Bank structures
3. Enable flexible resource sharing
    - Resource constraint: Register file size + shared memory size

Summary

1. Reviewed differences between software-managed cache and hardware cache
2. Explored the benefits of a unified memory and register file structure