Introduction to Control Divergence

Objectives

- Understand the occurrence of control divergence and the concept of thread reconvergence
  - Also described as branch divergence and thread divergence
- Cover a basic thread reconvergence mechanism – stack-based reconvergence
- Cover a non-stack based reconvergence mechanism – convergence barriers
- Dynamic Warp formation – inter-warp compaction
  - Increase lane utilization by merging warps
- Dynamic Warp Formation – intra-warp compaction
  - Increase lane utilization by compacting threads in a warp
Reading


Handling Branches

- CUDA Code:
  if(…) … (True for some threads)  
  else … (True for others)

- What if threads takes different branches?
- Branch Divergence!
Branch Divergence

- Occurs within a warp
- Branches lead serialization of branch dependent code
  - Performance issue: low warp utilization

```c
if(...) {
    ...
} else {
    ...
}
```

Idle threads

Reconvergence!

- Different threads follow different control flow paths through the kernel code
- Thread execution is (partially) serialized
  - Subset of threads that follow the same path execute in parallel

Example:

Courtesy of Wilson Fung, Ivan Sham, George Yuan, Tor Aamodt
Basic Idea

- **Split**: partition a warp
  - Two mutually exclusive thread subsets, each branching to a different target
  - Identify subsets with two activity masks → **effectively** two warps

- **Join**: merge two subsets of a previously split warp
  - Reconverge the mutually exclusive sets of threads

- **Orchestrate** the correct execution for nested branches

- **Note** the long history of techniques in SIMD processors (see background in Fung et. al.)

![Thread Warp Diagram](image)

Thread Reconvergence

- **Fundamental problem**: 
  - Merge threads with the same PC
  - How do we sequence execution of threads? Since this can effect the ability to reconverge

- **Question**: When can threads productively reconverge?

- **Question**: When is the best time to reconverge?
**Dominator**

- Node \( d \) dominates node \( n \) if every path from the entry node to \( n \) must go through \( d \)

**Immediate Dominator**

- Node \( d \) immediate dominates node \( n \) if every path from the entry node to \( n \) must go through \( d \) and no other nodes dominate \( n \) between \( d \) and \( n \)
Post Dominator

- Node $d$ post dominates node $n$ if every path from the node $n$ to the exit node must go through $d$.

Immediate Post Dominator

- Node $d$ immediate post dominates node $n$ if every path from node $n$ to the exit node must go through $d$ and no other nodes post dominate $n$ between $d$ and $n$. 
Baseline: PDOM

• A stack entry is a specification of a group of active threads that will execute that basic block

• The natural nested structure of control exposes the use of stack-based serialization

Example:

(14)
More Complex Example

- Stack based implementation for nested control flow
  - Stack entry RPC set to IPDOM
- Re-convergence at the immediate post-dominator of the branch


Implementation

- GPGPUSim model:
  - Implement per warp stack at issue stage
  - Acquire the active mask and PC from the TOS
  - Scoreboard check prior to issue
  - Register writeback updates scoreboard and ready bit in instruction buffer
  - When RPC = Next PC, pop the stack
- Implications for instruction fetch?
Implementation (2)

• warpPC (next instruction) compared to reconvergence PC

• On a branch
  - Can store the reconvergence PC as part of the branch instruction
  - Branch unit has NextPC, TargetPC and reconvergence PC to update the stack

• On reaching a reconvergence point
  - Pop the stack
  - Continue fetching from the NextPC of the next entry on the stack

Stack Based Reconvergence: Summary

• Compatible with the programming model

• Interactions with the scheduler
  - Uniform progress \(\rightarrow\) efficiency issue

• Conservative
  - Can we do better, i.e., earlier reconvergence?

• Issues?
  - Porting MIMD models
  - Interactions between scheduler (forward progress) and programming model (bulk synchronous execution)
Revisiting SIMT vs. MIMD (1)

- Properties of current MIMD implementations
  - Independent progress of a thread
  - Progress expectations of the scheduler $\rightarrow$ fairness
  - Producer-consumer dependencies between threads
- In general, scheduling orders alone cannot guarantee forward progress

(19)

Revisiting SIMT vs. MIMD (2)

- Properties of current SIMT implementations
  - Serialization of divergent thread execution
  - Re-convergence at the earliest point (e.g., @IPDOM)
  - Fairness: progress expectations of the scheduler
- Independent thread progress assumption does not apply
  - Blocking at the reconvergence point

(20)
SIMT Deadlock

A: *mutex = 0
B: while(!atomicCAS(mutex, 0, 1));
C: // critical section
   atomicExch(mutex, 0);

- Successful thread blocked at the reconvergence point
- Waiting threads blocked on lock release \(\rightarrow\) deadlock
- Need progress on divergent threads to continue execution
  - Fairness and progress demands on the warp scheduler
- Violation of (expected) programming model behaviors?

Figure from T. Aamodt, W. L., Fung. R. Rogers, "General Purpose Graphics Architectures", Synthesis Lectures on Computer Architecture, Draft

Goals

- Be able to use MIMD semantics \(\rightarrow\) use existing algorithmic idioms
  - Maintain correctness
- Porting of existing multithreaded applications to GPUs
- Avoid assumptions about compiler optimizations or schedulers
Convergence Barriers: Basic Idea

- Allocate a named barrier
- Add threads to the barrier

Barrier Hardware Structures

Split the warp \(\rightarrow\) two warps

Warp splits reconverge at the named barrier

Convergence Barriers: Barrier Behavior

- Allocate a named barrier
- Add threads to the barrier

- Richer barrier semantics
- Thread states
- Complex conditions for clearing a barrier
- Target for compiler analysis
- Target for warp schedulers

Warp splits reconverge at the named barrier
Tracking Data Structures

Convergence Barrier: Example -1 (1)

- ADD Instruction inserted by compiler to update barrier participation mask
- Named barrier, B0, is allocated
Convergence Barrier: Example – 1

Scheduler can schedule either warp independently.

- Warp 0 arrives at barrier B0
- Executes \textit{WAIT} instruction that changes thread state of active threads to blocked
  - Warp 0 waits
Convergence Barrier: Example – 1 (4)

1. Warp 1 arrives at B0 and executes WAIT instruction.
2. Check participation mask.
3. Update barrier structure.
4. Release the barrier.

Figure from patent US 2016/0019066A1

Nested Control Flow (1)

Add threads to barrier B0.

Figure from patent US 2016/0019066A1
Nested Control Flow (2)

Warp split

Figure from patent US 2016/0019066A1

Nested Control Flow (3)

Add 4 active threads to barrier B1

Figure from patent US 2016/0019066A1
Nested Control Flow (4)

Warp split

Figure from patent US 2016/0019066A1

Nested Control Flow (5)

Reconverge via wait

Figure from patent US 2016/0019066A1
Nested Control Flow (6)

Figure from patent US 2016/0019066A1

Nested Control Flow (7)

All threads arrive

Figure from patent US 2016/0019066A1
Nested Control Flow (8)

Reconverge

Figure from patent US 2016/0019066A1 (37)

Iteration (1)

• Compiler inserted YIELD instructions
• Threads are suspended and do not participate in the barrier at B1

Warp size

barrier participation mask

barrier state

thread state

PC

thread active

Figure from patent US 2016/0019066A1 (38)
**Iteration (2)**

- Compiler inserted YIELD instructions
- Threads are suspended and do not participate in the barrier at B1

**Warp split**

- Loop entry: ADD B1, LOOP, EXIT
- Warp size
- Barrier participation mask
- Barrier state
- Thread state
- PC
- Thread active

**Warp size**

- Barrier participation mask
- Barrier state
- Thread state
- PC
- Thread active

**Iteration (3)**

- Threads that have executed YIELD, placed in yield state
- Execution suspended

**Warp size**

- Barrier participation mask
- Barrier state
- Thread state
- PC
- Thread active

**Scheduler/compiler optimizations to minimize missed reconvergence opportunities**

- Barrier clears with active threads in warp 1
- Ignore threads in yield state
- These threads may continue without the yielded instructions (missed reconvergence op)
• Compiler responsibility to ensure divergent paths do not execute indefinitely
• Periodically yield to the scheduler
• Compiler-scheduler cooperation
• Heuristics for insertion of YIELD instructions vs. conditions, e.g., timeouts

Early Reconvergence
• Early exit
• Threads taking this path opt-out of the barrier → Opt-Out Instructions
Behavior at the Barrier

All threads here? Yes → Clear barrier

No → Remove Opt-out threads

Thread state is "EXITED"

Ignore yielded threads

Clear barrier → Clear YIELD states

Remaining threads here? Yes

No → scheduler

Non-Stack Based Reconvergence

• Close relationship between scheduler and compiler
  ➢ To guarantee progress

• Mechanisms for decoupling execution constraints (e.g., blocking/waiting @reconvergence) from dependencies
  ➢ Rely on compiler or hardware (e.g. timers) to decouple dependencies
  ➢ Performance cost via missed reconvergence opportunities

• Side effect is more scheduling flexibility
Can We Do Better?

- Warps are formed statically
- Key idea of dynamic warp formation
  - Find a pool of warps \( \Rightarrow \) how they can be merged?
- At a high level what are the requirements?

Compaction Techniques

- Can we reform warps so as to increase utilization?
- Basic idea: Compaction
  - Reform warps with threads that follow the same control flow path
  - Increase utilization of warps
- Two basic types of compaction techniques
  - Inter-warp compaction
    - Group threads from different warps
    - Group threads within a warp
      - Changing the effective warp size
Inter-Warp Thread Compaction: Dynamic Warp Formation

Goal

Warp 0
Warp 1
Warp 2
Warp 3
Warp 4
Warp 5

Merge threads?
if(…)
else {
...
}

(47)
Reading


Formation

• How do we get from statically formed warps from kernel launch (recall grid $\rightarrow$ warp mapping) to dynamic warps drawing threads from multiple warps?

• Constraints imposed by the register file organization
DWF: Example

D WF: Example

How Does This Work?

- Criteria for merging
  - Same PC
  - Complements of active threads in each warp
  - Recall: many warps/TB all executing the same code

- What information do we need to merge two warps
  - Need thread IDs and PCs

- Ideally how would you find/merge warps?
DWF: Microarchitecture Implementation

- **Thread Scheduler**
- **PC-Warp LUT**
- **Warp Pool**
- **Warp Allocator**

- Identify available lanes
- Identify occupied lanes
- Point to warp being formed
- Assists in aggregating threads

- Warps formed dynamically in the warp pool
- After commit check PC-Warp LUT and merge or allocated newly forming warp

**Lane aware**

 Courtesy of Wilson Fung, Ivan Sham, George Yuan, Tor Aamodt

---

DWF: Microarchitecture Implementation

- **Thread Scheduler**
- **PC-Warp LUT**
- **Warp Pool**
- **Warp Allocator**

- **Allegheny Valley**
- **Warp Update Register T**
- **Warp Update Register NT**

- **No Lane Conflict**

**Lane aware**

 Courtesy of Wilson Fung, Ivan Sham, George Yuan, Tor Aamodt
Resource Usage

• Ideally would like a small number of unique PCs in progress at a time → minimize overhead

• Warp divergence will increase the number of unique PCs
  √ Mitigate via warp scheduling

• Scheduling policies
  √ FIFO
  √ Program counter – address variation measure of divergence
  √ Majority/Minority- most common vs. helping stragglers
  √ Post dominator (catch up)

Hardware Consequences (1)

• Expose the implications that warps have in the base design
  √ Implications for register file access → lane aware DWF

• Register bank conflicts

**Hardware Consequences (2)**

Ignoring bank and lane assignments

Lane Aware

---

**Relaxing Implications of Warps**

- **Thread swizzling**
  - Essentially remap work to threads so as to create more opportunities for DWF \(\rightarrow\) requires deep understanding of algorithm behavior and data sets

- **Lane swizzling in hardware**
  - Provide limited connectivity between register banks and lanes \(\rightarrow\) avoiding full crossbars
Intra-Warp Thread Compaction: Cycle Compression

Goals

• Improve utilization in divergent code via intra-warp compaction

• Become familiar with the architecture of Intel’s Gen integrated general purpose GPU architecture
Integrated GPUs: Intel HD Graphics

Figure from The Computer Architecture of the Intel Processor Graphics Gen9, https://software.intel.com/en-us/articles/intel-graphics-developers-guides

Gen graphics processor
- 32-byte bidirectional ring
- Dedicated coherence signals
- Coherent distributed cache
- Shared with GPU
- Operates as a memory side cache

Shared physical memory

Figure from The Computer Architecture of the Intel Processor Graphics Gen9, https://software.intel.com/en-us/articles/intel-graphics-developers-guides
Inside the Gen9 EU

- Up to 7 threads
- 128, 256-bit registers/thread (8-way SIMD)
- Each thread executes a kernel
  - Threads may execute different kernels
  - Multi-instruction dispatch


Operation (1)

SIMD-16, SIMD-8, SIMD-32 instructions

- Divergence/reconvergence management
- Support both FP and Integer operations
  - 4, 32-bit FP operations
  - 8, 16-bit integer operations
  - 8, 16-bit FP operations
  - MAD operations each cycle

- 96 bytes/cycle read BW
- 32 bytes/cycle write BW
- Dispatch 4 instructions from 4 threads
- Constraints between issue slots

Figure from *The Computer Architecture of the Intel Processor Graphics Gen9*, https://software.intel.com/en-us/articles/intel-graphics-developers-guides (64)
Mapping the BSP Model

- Map multiple threads to SIMD instance executed by a EU Thread
- All threads in a TB or workgroup mapped to same thread (shared memory access)
Slice Organization

Slice: 24 EUs

- Shared memory
  - 64Kbyte/slice
  - Not coherent with other structures

Flexible partitioning
- SM
- Data cache
- Buffers for accelerators

Coherent Memory Hierarchy

- Not coherent

From The Computer Architecture of the Intel Processor Graphics Gen9,

(67)
Baseline Microarchitecture Operation

Microarchitecture Operation

Per-thread pre-fetch
- Per-thread decode: variable width SIMD + EX masks

Per-thread scoreboard check
- Thread arbitration, dual issue/2-cycles, compaction cntrl

Operand fetch/swizzle
- Encode swizzle in RF access

Instruction execution happens in waves of 4-wide operations
- Note: variable width SIMD instructions

16-way SIMD

Executed in 4 cycles
Divergence Assessment

Coherent applications

SIMD Efficiency =
Average Lane Utilization
Average SIMD width

Basic Cycle Compression (1)

- Compress cycles by suppressing dispatch
  - Operand fetch
  - Instruction execution
  - Complex MRF access patterns

- Actual operation depends on data types, execution cycles/op
- Note power/energy savings
Basic Cycle Compression (2)

<table>
<thead>
<tr>
<th>Basic Cycle Compression: MRF Access</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Instruction Predicate Mask</th>
<th>Execution Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch Mask</td>
<td></td>
</tr>
<tr>
<td>Divergence State</td>
<td></td>
</tr>
</tbody>
</table>

Cycle compression applies to:
- Dispatch (complex MRF accesses can take multiple cycles)
- Predication
- Other cases with idle cycles

### Instruction Fetch Instruction

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Instruction Decoding

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Execution

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Store/Load Queue

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Rename/Dispatch

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Execution Mask

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Register File

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Baseline Register File

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

### Organization for BCC

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-MR</td>
<td>D-MR</td>
<td>D-MR</td>
</tr>
</tbody>
</table>

(73)
Swizzle Cycle Compression

- Use more flexible thread → lane assignment
- Operand data flow?
- Compact threads to create idle cycles in the pipeline

Register File Access for BCC

32-bits
32-bit Operand Mapping for SIMD16

Add (16) R12 R8 R10 [EXEC MASK 0xFFFF]

- Implicitly use pairs of registers for all 32b operands
- R8, R9 → 16 x 32 = 512 bits

ADD (16) R12 R8 R10
ADD.Q0 (4) R12.H0 R8.H0 R10.H0
ADD.Q2 (4) R13.H0 R9.H0 R11.H0
ADD.Q3 (4) R13.H1 R9.H1 R11.H1

Quad instructions

Operand Flow (1)


128 bits, 4-Lane ALU
128 bits, 4-Pane ALU
Operand Flow (2)


ADD.Q0 (4) R12.H0 R8.H0 R10.H0
ADD.Q2 (4) R13.H0 R9.H0 R11.H0
ADD.Q3 (4) R13.H1 R9.H1 R11.H1

Quad instructions

128 bits, 4-Lane ALU
128 bits, 4-Pane ALU

Operand Flow (3)

ADD.Q0 (4) R12.H0 R8.H0 R10.H0
ADD.Q2 (4) R13.H0 R9.H0 R11.H0
ADD.Q3 (4) R13.H1 R9.H1 R11.H1

128-bit buses

128 bits, 4-Lane ALU
128 bits, 4-Pane ALU
Compressing Idle Cycles (1)

ADD (16) R12 R8 R10 [EXEC MASK 0XF0F0]

R8  R9  R10  R11  R12  R13

ADD.Q0 (4) R12.H0 R8.H0 R10.H0
ADD.Q2 (4) R13.H0 R9.H0 R11.H0

active operands in this instruction

Suppressed

Compressed

Compressing Idle Cycles (2)

ADD (16) R12 R8 R10 [EXEC MASK 0XF0F0]

R8  R9  R10  R11  R12  R13

ADD.Q0 (4) R12.H0 R8.H0 R10.H0
ADD.Q2 (4) R13.H0 R9.H0 R11.H0

active operands in this instruction

Suppressed

Compressed
Swizzle Cycle Compression

- Use more flexible thread → lane assignment
  - Operand data flow?
  - Compact threads to create idle cycles in the pipeline

Distributing Data to Lanes (1)

- Restrict swizzle patterns
  - Does not support all possible compression patterns
  - Need fast computation of efficient swizzle settings
Distributing Data to Lanes (2)

- Each lane connected to 32 bits of the 128 bit bus
- Each element of the Quad is connected to 32 bits of a 128-bit segment

SCC Operation

- Can compact across Quads
- Pack lane inputs into a quad
- RF for a Single Operand
- Swizzle settings overlapped with RF access
- Note increased area, power/energy
- Note the need for inverse permutations
Compaction Opportunities

For K active threads what is the maximum cycle savings for SIMD N instructions?

Performance Savings

• Difference between saving cycles and saving time
  - When is #cycles ≠ time?

(87)
Summary

- Multi-cycle warp/SIMD/work_group execution
- Optimize #cycles/warp by compressing idle cycles
  - Rearrange idle cycles via swizzling to create opportunity
- Sensitivities to the memory interface speeds
  - Memory bound applications may experience limited benefit

Intra-Warp Compaction

- Scope limited to within a warp
- Increasing scope means increasing warp size, explicitly, or implicitly (treating multiple warps as a single warp)
Summary

• Control flow divergence is a fundamental performance limiter for SIMT execution

• Dynamic warp formation is one way to mitigate these effects
  ❖ We will look at several others

• Must balance a complex set of effects
  ❖ Memory behaviors
  ❖ Synchronization behaviors
  ❖ Scheduler behaviors