Introduction to Control Divergence

Goal

- Understand optimizations to improve the performance of kernels in the presence of control divergence
- Better understanding of constraints imposed by the microarchitecture
Reading


- Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2013


---

Thread Block Compaction

W. Fung and T. Aamodt

HPCA 2011
Goal

- Overcome some of the disadvantages of dynamic warp formation
  - Impact of Scheduling
  - Breaking implicit synchronization
  - Reduction of memory coalescing opportunities

DWF Pathologies: Starvation

- Majority Scheduling
  - Best Performing
  - Prioritize largest group of threads with same PC

- Starvation
  - LOWER SIMD Efficiency!

- Other Warp Scheduler?
  - Tricky: Variable Memory Latency
DWF Pathologies: Extra Uncoalesced Accesses

- Coalesced Memory Access = Memory SIMD
  - 1st Order CUDA Programmer Optimization
- Not preserved by DWF

\[ E: B = C[tid.x] + K; \]

<table>
<thead>
<tr>
<th>No DWF</th>
<th></th>
<th>Acc = 3</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>E: 1 2 3 4</td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>E: 5 6 7 8</td>
<td></td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>E: 9 10 11 12</td>
<td></td>
<td>180</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>With DWF</th>
<th></th>
<th>Acc = 9</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>E: 1 2 7 12</td>
<td></td>
<td>100</td>
<td>L1 Cache Absorbs Redundant Memory Traffic</td>
</tr>
<tr>
<td>E: 9 6 3 8</td>
<td></td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>E: 5 10 11 4</td>
<td></td>
<td>180</td>
<td></td>
</tr>
</tbody>
</table>

DWF Pathologies: Implicit Warp Sync.

- Some CUDA applications depend on the lockstep execution of "static warps"

```
Warp 0  Thread 0 ... 31
Warp 1  Thread 32 ... 63
Warp 2  Thread 64 ... 95
```

Performance Impact


### Thread Block Compaction

- **Block-wide Reconvergence Stack**
  - Regroup threads within a block
  - Better Reconv. Stack: Likely Convergence
    - Converge before Immediate Post-Dominator
  - Robust
    - Avg. 22% speedup on divergent CUDA apps
    - No penalty on others
Observation

- Compute kernels usually contain divergent and non-divergent (coherent) code segments
- Coalesced memory access usually in coherent code segments
  - DWF no benefit there
Thread Block Compaction

- Barrier @ Branch/reconverge pt.
  - All avail. threads arrive at branch
  - Insensitive to warp scheduling
- Run a thread block like a warp
  - Whole block move between coherent/divergent code
  - Block-wide stack to track exec. paths reconvg.
- Warp compaction
  - Regrouping with all avail. threads
  - If no divergence, gives static warp arrangement

```
A: K = A[tid.x];
B: if (K > 10)
   C: K = 10;
else
D: K = 0;
E: B = C[tid.x] + K;
```
Thread Block Compaction

- Barrier every basic block?! (Idle pipeline)
- Switch to warps from other thread blocks
  - Multiple thread blocks run on a core
  - Already done in most CUDA applications

High Level View

- DWF: warp broken down every cycle and threads in a warp shepherded into a new warp (LUT and warp pool)
- TBC: warps broken down at potentially divergent points and threads compacted across the thread block
Microarchitecture Modification

- Per-Warp Stack $\rightarrow$ Block-Wide Stack
- I-Buffer + TIDs $\rightarrow$ Warp Buffer
  - Store the dynamic warps
- New Unit: Thread Compactor
  - Translate activemask to compact dynamic warps
- More Detail in Paper
Thread Compactor

- Convert *activemask* from block-wide stack to *thread IDs* in warp buffer
- Array of Priority-Encoder

---

**Likely-Convergence**

- Immediate Post-Dominator: Conservative
  - All paths from divergent branch must merge there
- Convergence can happen earlier
  - When any two of the paths merge

```c
while (i < K) {
    X = data[i];
    A:   if (X = 0)
    B:     result[i] = Y;
    C:   else if (X = 1)
    D:     break;
    E:   i++;
}
F: return result[i];
```

- Extended Recvg. Stack to exploit this
  - TBC: 30% speedup for Ray Tracing
    "Details in Paper"

**Likely-Convergence (2)**

- NVIDIA uses break instruction for loop exits
  - That handles last example
- Our solution: Likely-Convergence Points

<table>
<thead>
<tr>
<th>PC</th>
<th>RPC</th>
<th>LPC</th>
<th>LPC Pos</th>
<th>Active Thds</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>--</td>
<td>--</td>
<td>1 2</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>E</td>
<td>1</td>
<td>3 4</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>E</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

"Convergence!"

- This paper: only used to capture loop-breaks
Likely-Convergence (3)

- Likely-Convergence (3)
- Check!
- Merge inside the stack


Likely-Convergence (4)

- Applies to both per-warp stack (PDOM) and thread block compaction (TBC)
  - Enable more threads grouping for TBC
  - Side effect: Reduce stack usage in some case
Effect on Memory Traffic

- TBC does still generate some extra uncoalesced memory access

\[ \text{\#Acc} = 4 \]

\[ \begin{array}{c|cccc} 
C & 1 & 2 & 7 & 8 \\
\hline 
C & 5 & 11 & 12 & \end{array} \]

2\(^{nd}\) Acc will hit the L1 cache

No Change to Overall Memory Traffic In/out of a core

Normalized Memory Stalls

Normalized to Baseline

Conclusion

- Thread Block Compaction
  - Addressed some key challenges of DWF
  - One significant step closer to reality

- Benefit from advancements on reconvergence stack
  - Likely-Convergence Point
  - Extensible: Integrate other stack-based proposals
Goals

- Improve the performance of inter-warp compaction techniques
- Predict when branches diverge
  - Borrow philosophy from branch prediction
- Use prediction to apply compaction only when it is beneficial
Issues with Thread Block Compaction

TBC: warps broken down at potentially divergent points and threads compacted across the thread block

Barrier synchronization overhead cannot always be hidden

When it works, it works well

The presence of divergence does not imply utility of compaction

Divergence Behavior: A Closer Look

Figure from M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA 2012
Compaction-Resistant Branches

if (tx == 0) // Conditional branch that is divergent but compaction-ineffective
    input_node[ty] = input_cude[index_in];

Control flow graph

Figure from M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA 2012

Compaction-Resistant Branches(2)

for (i = 1; i < __log2(HEIGHT); i++)
{
    ... // Loop-end branch: Conditional branch that is non-divergent
}

Control flow graph

Figure from M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA 2012
Impact of Ineffective Compaction

Threads shuffled around with no performance improvement

However, can lead to increased memory divergence!

Insight

- Synchronization barrier delays cannot be amortized for some branches
  - Barrier restricts the parallelism available to hide latency
- Distinguish between productive and unproductive candidates

Goal: More parallelism, more latency hiding, more scheduling targets, retain compaction
Basic Idea

- Only stall and compact when there is a high probability of compaction success
- Otherwise allow warps to bypass the (implicit) barrier
- Compaction adequacy predictor!
  - Think branch prediction!

Example: TBC vs. CAPRI

Bypassing enables increased overlap of memory references

Figure from M. Rhu and M. Erez, “CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures,” ISCA 2012
No Divergence → No Stalling

(a) Because Wₐ does not diverge upon arriving at PC=BRₑₑ, Wₐ is bypassed and increments Wₐ of path B by one. The mask is updated to [11] to have Wₐ’s active mask not be considered for generating compacted warps.

Tracking outstanding warps on a control path

Do not consider for compaction

On Divergence

(b) Wₐ arrives at PC=BRₑₑ and is stalled upon divergence. Wₐ of path A is decremented by one and UMask remains at [011] because the active mask for Wₐ needs to be considered for compaction.

Diverge (TBC)  Decrement (TBC)

Update prediction table

Diverge, stall, initialize history, all others will now stall
History-Based Compaction

(c) WCnt becomes zero as W0 arrives at PC=BRw. The CTA-wide active masks at TOS is forwarded to WCU and compaction is initiated. Note that W0’s active mask is not considered for compaction as UMask is [011].

All warps have arrived
Compact (except for w0)

• Diverge and history available, predict, update prediction
• Compact
• CAPT updated

Initialize & Proceed

(d) WCU generates two warps and increments WCnt by two. As the Predictor evaluates PC=BRw to be connection-ineffective, the history bit is reset. UMask is initialized to [111].

Compact and update total number of new warps
The Predictor

- Prediction uses active masks of all warps
  - Need to understand what could have happened
- Actual compaction only uses actual stalled warps
- Minimum provides the maximum compaction ability, i.e., \#compacted warps
- Update history predictor accordingly
  - Learn from branch predictor

Figure from M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA 2012

Behavior

(a) Divergent benchmarks

(b) Non-divergent benchmarks

Figure from M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA 2012
Impact of Implicit Barriers

(b) Normalized idle cycles accumulated across all cores.

- Idle cycle count helps us understand the negative effects of implicit barriers in TBC

Figure from M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA 2012

Summary

- The synchronization overhead of thread block compaction can introduce performance degradation
- Some branches more divergence than others
- Apply TBC judiciously \( \rightarrow \) predict when it is beneficial
- Effectively predict when the inter-warp compaction is effective.
Improving GPU Performance via Large Warps and Two-Level Scheduling
V. Narsiman et. al
MICRO 2011

Goals

- Improve performance of divergent code via compaction of threads within a warp
- Integrate warp scheduling optimization with intra-warp compaction
Resource Underutilization

32 warps, 32 threads per warp, single SM, RR warp scheduler

Due to control divergence

Due to memory divergence

Relationship between control and memory divergence

Warp Scheduling and Locality Effects

- Opportunities for memory coalescing
- Potential for exposing memory stalls
- Exploit row buffer locality
- Degrades memory reference locality and row buffer locality
- Overlaps memory accesses

(47)

(48)
Challenges and Tradeoffs

- Hitting long latency operations at the same time vs. exploiting locality in the memory hierarchy
- Generating enough parallelism to hide memory latencies vs. starvation and last warp effects
- Can we do both → intra-warp compaction + scheduling
  - Large Warp Microarchitecture
  - Two-Level Round Robin scheduling

Key Ideas

- Conventional design today → warp size = #SIMD lanes
- Use large warps → multi-cycle issue of sub-warsps
  - Compact threads in a warp to form fully utilized sub-warsps
- 2-level scheduler to spread memory accesses in time
  - Reduce memory related stall cycles enough
  - Maintain enough locality
Large Warps

Typical Operation

- Warp size = 4
- Warp size = SIMD width
- SIMD Width = 4

Proposed Approach

- Warp size = 16
- Large Warp
- Multi-cycle Issue

• Large warps converted to a sequence of sub-warps
• Similar in spirit to Intel Gen strategy for intrawarp compaction

Impact on the Register File

Baseline Register File

Large Warp Active Mask Organization

Need separate decoders per bank
Sub-Warp Compaction

- Iteratively select one thread per column to create a packed sub-warp
- Dynamic generation of sub-warsps

Scheduling Constraints

- Next large warp cannot be scheduled until first sub-warp completes execution
  - Scoreboard checks for issue dependencies
    - Thread not available for packing into a sub-warp unless previous issue (sub-warp) has completed → single bit status
    - Simple check
  - Re-fetch policy for conditional branches
    - Must wait till last sub-warp finishes
- Optimization for unconditional branch instructions
  - Do not create multiple sub-warsps
  - Sub-warping always completes in a single cycle
Effect of Control Divergence

- Note that divergence is unknown until all sub-warsps execute
  - Divergence management just happens on large warp boundaries
  - Need to buffer sub-warp state, e.g., active masks

- The last warp effect
  - Cannot fetch the next instruction in a warp until all sub-warsps issue
  - Trailing warp (warp divergence effect) can lead to many idle cycles

- Effect of the last thread
  - E.g., in data dependent loop iteration count across threads
  - Last thread can hold up reconvergence

A Round Robin Warp Scheduler

- Exploit inter-warp reference locality in the cache
- Exploit inter-warp reference locality in the DRAM row buffers
- However, need to maintain latency hiding
Two Level Round Robin Scheduler

Fetch Group 0
- LW0
- LW1
- LW2
- LW3

Fetch Group 1
- LW4
- LW5
- LW6
- LW7

Fetch Group 2
- LW8
- LW9
- LW10
- LW11

Fetch Group 3
- LW12
- LW13
- LW14
- LW15

Scheduler Behavior

- Need to set fetch group size carefully → tune to fill the pipeline
- Timeout on switching fetch groups to mitigate the last warp effect
Some Interesting Behaviors

- Large warp size/Fetch Group size effects
  - Tuning latency hiding → avoiding last warp effects
- Timeout on warp execution
  - 32K instructions (empirical number)
- What is the coherent vs. non-coherent regions and large warps?
- Variance in divergence patterns
  - Have larger negative impact

Summary

- Intra-warp compaction made feasible due to multi-cycle warp execution
  - Mismatch between warp size and SIMD width enables flexible intra-warp compaction
- Do not make warps too big → last thread effect begins to dominate
M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013

Goals

• Understand the limitations of compaction techniques and proximity to ideal compaction

• Provide mechanisms to overcome these limitations and approach ideal compaction rates
Key Insight

- Divergence patterns are correlated to statically defined elements
  - Architecture: mapping of threads to lanes (via mapping to warps)
  - Program elements that are constant: e.g., grid parameters

- Lane aware compaction is limited by these correlations

- Break the correlation to improve compaction techniques, e.g., thread block compaction

Mapping Threads to Lanes: Today

Linearization of thread IDs

Modulo assignment to lanes

Grid 1

Block (6, 6)  Block (6, 1)
Block (1, 6)  Block (1, 1)

Thread (0,0)  Thread (0,0,1)  Thread (0,0,2)  Thread (0,0,3)
Thread (0,1,0)  Thread (0,1,1)  Thread (0,1,2)  Thread (0,1,3)

(1,0,0)  (1,0,1)  (1,0,2)  (1,0,3)

Warp 0  Warp 1  • • •
Baseline Register File Organization

- Wide register format
- Static assignment of threads to lanes
- Single decoder across the bank
- Comprised of 1R/1W banks

Aligned Divergence

- Each of these threads are aligned
- Break this alignment to improve compactability
- Remapping at the microarchitecture level
Aligned Divergence and Compaction

Code #3) Branch dependent on a programmatic value - (i)

0 // Code snippet from the kernel of BITONIC benchmark
1 2 const unsigned int tid = threadIdx.x;
3 ...
4 for (unsigned int k = 2; k <= N; k *= 2) {
5 for (unsigned int j = k/2; j>0; j/=2) {
6   unsigned int ixj = tid ^ j;
7     if (ixj > tid) {
8       if ( tid & k)==0 ) (...} else {...
9     }
10   _syncthreads();
11 }
12 }

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR WU</td>
<td>TID 0, 1, 2, 3</td>
<td>4, 5, 6, 7</td>
<td>8, 9, 10, 11</td>
<td>12, 13, 14, 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR W1</td>
<td>TID 16, 17, 18, 19</td>
<td>20, 21, 22, 23</td>
<td>24, 25, 26, 27, 28, 29, 30, 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR W2</td>
<td>TID 32, 33, 34, 35</td>
<td>36, 37, 38, 39</td>
<td>40, 41, 42, 43</td>
<td>44, 45, 46, 47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR W3</td>
<td>TID 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Threads that are active when ‘(j = 1)’, having ((ixj>tid) == true)

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013 (67)

Aligned Divergence

• Threads mapped to a lane tend to evaluate (programmatic) predicates the same way
  ✓ Empirically, rarely exhibited for input, data dependent control flow behavior

• Compaction cannot help in the presence of lane conflicts

  → performance of compaction mechanisms depends on both divergence patterns and lane conflicts

• We need to understand impact of lane assignment

(68)
Branch Behavior

- Branches dependent on constants have correlated behaviors → aligned divergence
- Branches that depend on data values are typically do not exhibited correlated behaviors

Data Dependent Branches

```
Code #11 Branch depending on data arrays - (1)
0    // Code snippet from the kernel of BFS benchmark
1 2 // g_graph_visited and g_graph_edges are data array parameters.
3         int tid = blockIdx.x * MAX_THREADS_PER_BLOCK + threadIdx.x;
4     ...
5 6 int id = g_graph_edges[id];
7 if (g_graph_visited[id])
8       
9     ...
10   }
```

- Data dependent control flows less likely to produce lane conflicts, i.e., aligned divergent behavior

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013
Programmatic Branches

- Programmatic branches can be correlated to lane assignments (with modulo assignment)
- Program variables that operate like constants across threads can produce correlated branching behaviors

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013

P-Branches vs. D-Branches

- P-branches are the problem!
- Existing techniques work for D-branches

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013
Compaction Opportunities

- Lane reassignment can improve compaction opportunities
  - Imagine these threads had different home lanes
- Transparent to software

Impact of Lane Reassignment

Goal: Improve “compactability”

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013

(73)
Random Permutations

- Does not always work well \( \rightarrow \) works well on average
- Better understanding of programs can lead to better permutations choices

Figure from M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation," ISCA 2013

---

Mapping Threads to Lanes: New

SIMD Width = 8

What criteria do we use for lane assignment?
• Distribute active threads in a warp across lanes in a balanced manner
• Analogous to skewed assignment of threads in a warp to lanes
### Balanced Permutation (2)

<table>
<thead>
<tr>
<th>Lane-ID</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>For W0</td>
<td>XOR-000</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>For W1</td>
<td>XOR-111</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>For W2</td>
<td>XOR-001</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>For W3</td>
<td>XOR-110</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>For W4</td>
<td>XOR-010</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>For W5</td>
<td>XOR-101</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>For W6</td>
<td>XOR-011</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>For W7</td>
<td>XOR-100</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Repeat above

\[ \text{XOR}_{\text{evenWID}} = \frac{\text{evenWID}}{2} \]

Logical TID of 0 in each warp is now assigned a different lane

---

### Remapping Threads to Lanes

Spread out aligned thread states

---

*Figure from M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation," ISCA 2013*
Characteristics

- **Vertical Balance**: Each lane only has logical TIDs of distinct threads in a warp
- **Horizontal Balance**: Logical TID \( x \) in all of the warps is bound to different lanes
- This works when CTA have fewer than SIMD Width warps: why?
- Note that random permutations achieve this only on average

Impact on Memory Coalescing

- Modern GPUs do not require ordered requests
- Coalescing can occur across a set of requests \( \rightarrow \) specific lane assignments do not affect coalescing behavior
- Increase is L1 miss rate offset by benefits of compaction

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013
Speedup of Compaction

- Can improve the compaction rate of divergence due to the majority of programmatic branches

Figure from M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation," ISCA 2013

Compaction Rate vs. Utilization

Distinguish between compaction rate and utilization!

Figure from M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation," ISCA 2013
Application of Balanced Permutation

- Permutation is applied when the warp is launched
- Maintained for the life of the warp
- Does not affect the baseline compaction mechanism
- Enable/disable SLP to preserve target specific, programmer implemented optimizations

Summary

- Structural hazards limit the performance improvements from inter-warp compaction
- Program behaviors produce correlated lane assignments today
- Remapping threads to lanes enables extension of compaction opportunities
The Dual Path Execution Model for Efficient GPU Control Flow
M. Rhu and M. Erez
HPCA 2013

Goal

- Reconvergence-based techniques permit only a single divergent path to be active at any point in time
  - Elegant, but serializes execution

- Support parallel execution using stack-based reconvergence
  - Combine principals of dynamic subdivision and stack-based reconvergence

- Increase parallelism without reducing SIMD efficiency
Baseline Execution Profile

- Only one active path at a time for a warp
- Serialization of path execution

Concurrency Behavior

- Note only two active paths at a time in a single stack (nested control flow segment of code)
- Top level divergent set of threads are not interleaved

Figure from M. Rhu and M. Erez, “The Dual Path Execution Model for Efficient GPU Control Flow,” HPCA 2013
Comparison Point: Memory Divergence

• How do we hide latency between threads in a warp?

• The "last thread" effect due to implicit lock step progress

• Dynamic Warp Subdivision (DWS) as a solution
  - Warp splits ala convergence barriers

Dynamic Warp Subdivision

• Dynamically split warps into smaller, independently schedulable units
  - Overlap miss behavior as much as possible without adding new threads
  - Allow the active threads to run-ahead

• When to split and reconverge?
• How do we orchestrate (schedule) better memory behaviors?
MLP for Divergent Branches

- Overlap memory access in different branch paths

Run-Ahead Execution

- Early issue of memory instructions
Intra-Warp Latency Hiding

- Threads in the same warp can progress overlapping memory access latency

Prefetching Behavior of Warp Splits

Figure from J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," ISCA 2010
PDOM Execution

Figure from M. Rhu and M. Erez, "The Dual Path Execution Model for Efficient GPU Control Flow," HPCA 2013

Dual Path Execution

Figure from M. Rhu and M. Erez, "The Dual Path Execution Model for Efficient GPU Control Flow," HPCA 2013
Handling Dependencies in a Warp

- What about data dependencies?
  - Dual instruction issue from a warp

- Need a per warp scoreboard

Dual Path Scoreboard

- Replicate Scoreboard for each path

- How do we distinguish cross-path vs. in-path dependencies?

Figure from M. Rhu and M. Erez, "The Dual Path Execution Model for Efficient GPU Control Flow," HPCA 2013
Dependency Management

Case I
- Incorrect execution
- Deadlock

Case II
- Pending writes before reconvergence

Case III
- False dependencies

Case IV
- Distinct registers

Example code

```
// Path A
load r0, MEM[-];
if(()) // Path B
  load r1, MEM[-];
else // Path C
  add r5, r9, r2;
...
if(()) // Path D
  add r4, r1, r3;
else // Path E
  sub r4, r1, r3;
// Path F
... load r7, MEM[-];
// Path G
add r8, r1, r7;
```

Left Path Right Path

```
// Path A
load r0, MEM[-];
if(()) // Path B
  load r1, MEM[-];
else // Path C
  add r5, r9, r2;
...
if(()) // Path D
  add r4, r1, r3;
else // Path E
  sub r4, r1, r3;
// Path F
... load r7, MEM[-];
// Path G
add r8, r1, r7;
```

On divergence shadow bits set
- Actually write (B) unrelated to reads (D&E)

Shadow bits not set
- These false dependencies avoided

Figure from M. Rhu and M. Erez, “The Dual Path Execution Model for Efficient GPU Control Flow,” HPCA 2013
Dependency Management (3)

Example code

```
// Path A
load r0, MEM[-];
if() // Path B
  load r1, MEM[-];
else // Path C
  add r5, r0, r2;
...
if() // Path D
  add r4, r1, r3;
else // Path E
  sub r4, r1, r3;
// Path F
  load r7, MEM[-];
// Path G
  add r8, r1, r7;
```

Left Path | Right Path
---|---
// Path A | // Path A
load r0, MEM[-]; | load r0, MEM[-];
if() // Path B | if() // Path B
  load r1, MEM[-]; | else // Path C
else // Path C | load r5, r0, r2;
  add r5, r0, r2;
...
if() // Path D | if() // Path D
  add r4, r1, r3; | else // Path E
else // Path E
  sub r4, r1, r3;
// Path F | // Path F
  load r7, MEM[-]; | ...
// Path G | load r7, MEM[-];
  add r8, r1, r7;
```

- Shadow bits are set
  - these are false dependencies

Figure from M. Rhu and M. Erez, “The Dual Path Execution Model for Efficient GPU Control Flow,” HPCA 2013

Updating the Scoreboard

- Indicates cross-path dependency due to pending write before divergence
- Copy P35 at divergence

```
// Path A
load r0, MEM[-];
if() // Path B
  load r1, MEM[-];
else // Path C
  add r5, r0, r2;
...
if() // Path D
  add r4, r1, r3;
else // Path E
  sub r4, r1, r3;
// Path F
  load r7, MEM[-];
// Path G
  add r8, r1, r7;
```

- When checking scoreboard during issue, check shadow bits in the other scoreboard
  - E.g., make sure Path A write has completed for Path C
- Writes need single bit to indicate which scoreboard to clear

Figure from M. Rhu and M. Erez, “The Dual Path Execution Model for Efficient GPU Control Flow,” HPCA 2013
Warp Scheduler Impact

- Typically multiple warp schedulers per SM
- Dual path execution can double the number of schedulable entries
- Expect it can be more sensitive to the warp scheduling policy

Opportunity

- Non-interleavable vs. interleavable branches
- How do we assess opportunity?

\[
\text{Avg}_{\text{path}} = \frac{1}{N} \sum_{i=1}^{N} \text{NumPath}_i
\]
Some Points to Note

- Achieve reduction in idle cycles → expected
- Impact on L1 cache misses → can increase
- Use for handling memory divergence
- Path forwarding to enable concurrency across stack levels (not just at the TOS) → not clear this is worth the cost
Summary

- Looking for structured concurrency → living with control divergence handling solutions
- Low cost overhead for applications that have high percentage of interleavable divergent branches
- Extends the performance of PDOM

Summary: Inter-Warp Compaction

Co-Design of applications, resource management, software, microarchitecture,