Introduction to Control Divergence

Lectures Slides and Figures contributed from sources as noted

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.


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

```
B: if (K > 10)
C:   K = 10;
else
D:   K = 0;
E:   B = C[tid.x] + K;
```
DWF Pathologies:
Extra Uncoalesced Accesses

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

DWF Pathologies: Implicit Warp Sync.

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

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

Wilson Fung, Tor Aamodt  Thread Block Compaction
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

Wilson Fung, Tor Aamodt  Thread Block Compaction  13

---

Thread Block Compaction

<table>
<thead>
<tr>
<th>PC RPC</th>
<th>Active Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>D E</td>
<td>5 6 7 8</td>
</tr>
<tr>
<td>C E</td>
<td>9 10 11 12</td>
</tr>
</tbody>
</table>

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

A: 1 2 3 4
A: 5 6 7 8
A: 9 10 11 12

---

Time

Wilson Fung, Tor Aamodt  Thread Block Compaction  14
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

![Diagram showing thread block compaction]

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

---

![Microarchitecture Modification](image_url)

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

```
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</th>
<th>ActiveThds</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 (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

![Diagram](image)

- \#Acc = 4
- Memory
- No Change to Overall Memory Traffic
- In/out of a core

![Chart](image)

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

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

Divergence Behavior: A Closer Look

Example 1 Code that contains two potentially divergent branches.

```c
/* - Imported from _global_ void kpuon_layerforward_CUDA() kernel of BMVP.
  - input_code[] designates a global memory region */

int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;

__shared__ float input_node[HEIGHT];

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

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

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

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

15 for (i = 1; i < __log2(HEIGHT); i++)
16 {
17     ...
18     // Loop-end branch: Conditional branch that is non-divergent
19 }

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!

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

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

---

CAPRI: Example

No divergence, no stalling

Diverge, stall, initialize history, all others will now stall

- Diverge and history available, predict, update prediction
- All other warps follow (one prediction/branch)
- CAPT updated

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

Available threads in a lane

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

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

Summary

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

- The synchronization overhead of thread block compaction can introduce performance degradation

- Some branches more divergence than others

- Apply TBC judiciously → 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

Due to control divergence
Due to memory divergence

Warp Scheduling and Locality

Opportunities for memory coalescing
Potential for exposing memory stalls

• Degrades memory reference locality and row buffer locality
• Overlaps memory accesses

(43)

(44)
Key Ideas

- Conventional design today \( \rightarrow \) warp size = #SIMD lanes

- Use large warps \( \rightarrow \) 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

Large Warps

- Large warps converted to a sequence of sub-warsps
Sub-Warp Compaction

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

Impact on the Register File

Need separate decoders per bank
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 \( \rightarrow \) single bit status
  - Simple check
  - However on a branch, all sub-warsps must complete before it is eligible for instruction fetch scheduling
- 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

Fetch Group 1

Fetch Group 2

Fetch Group 3

Thread Fetch Group Size: Enough to keep the pipeline busy
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

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
Limitations of Compaction

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

Mapping Threads to Lanes: Today

Linearization of thread IDs

Modulo assignment to lanes

Warp 0

Warp 1

(57)

(58)
Data Dependent Branches

- Data dependent control flows less likely to produce lane conflicts

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!

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

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

---

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
Random Permutations

- Does not always work well → 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?
Balanced Permutation

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

- For W0: XOR-000
- For W1: XOR-111
- For W2: XOR-001
- For W3: XOR-110
- For W4: XOR-010
- For W5: XOR-101
- For W6: XOR-011
- For W7: XOR-100

### Even Warps: Permutation within a Half Warp

Each lane has a single instance of a logical thread from each warp.

### Odd Warps: Swap Upper and Lower

\[
XOR_{\text{evenWID}} = \frac{\text{evenWID}}{2}
\]

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

---

Balanced Permutation (2)

<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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

- For W0: XOR-000
- For W1: XOR-111
- For W2: XOR-001
- For W3: XOR-110
- For W4: XOR-010
- For W5: XOR-101
- For W6: XOR-011
- For W7: XOR-100

### Odd Warps: Swap Upper and Lower

\[
XOR_{\text{evenWID}} = \frac{\text{evenWID}}{2}
\]

Logical TID of 0 in each warp is now assigned a different lane.
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 → 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.
Summary: Inter-Warp Compaction

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