Introduction to Control Divergence

Objective

- 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 – PDOM
  - Set up discussion of further optimizations and advanced techniques
- Explore one approach for mitigating the performance degradation due to control divergence – dynamic warp formation
Reading

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

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 branch dependent code
  - Performance issue: low warp utilization

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

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
Basic Idea

- **Split**: partition a warp
  - Two mutually exclusive thread subsets, each branching to a different target
  - Identify subsets with two activity masks \(\rightarrow\) 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 affect 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 stack-based serialization.
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

Can We Do Better?

- Warps are formed statically
- Key idea of dynamic warp formation
  - Show a pool of warps and 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 Techniques

Lectures Slides and Figures contributed from sources as noted (20)
Goal

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

Merge threads?

if(...) {
    ...
}  
else {
    ...
}

Reading


**DWF: Example**

**Legend**
- A: Execution of Warp x at Basic Block A
- A: Execution of Warp y at Basic Block A
- D: A new warp created from scalar threads of both Warp x and y executing at Basic Block D

**Baseline**

**Dynamic Warp Formation**

**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

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

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

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

- Register bank conflicts

From Fung, et. al., "Dynamic Warp Formation: Efficient MIMD Control Flow in SIMD Graphics Hardware, ACM TACO, June 2009"
Relaxing Implications of Warps

• Thread swizzling
  ▶ Essentially remap work to threads so as to create more opportunities for DWF → requires deep understanding of algorithm behavior and data sets

• Lane swizzling in hardware
  ▶ Provide limited connectivity between register banks and lanes → avoiding full crossbars

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
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

---

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

---

```
#Acc = 3
```

---

```
#Acc = 9
```
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


Wilson Fung, Tor Aamodt

---

Performance Impact

(a) SIMD Efficiency
(b) Normalized Memory Stalls

Thread Block Compaction

- Block-wide Reconvergence Stack

- Better Reconv. Stack: Likely Convergence
  - Converge before Immediate Post-Dominator

- Robust
  - Avg. 22% speedup on divergent CUDA apps
  - No penalty on others

---

GPU Microarchitecture

- SIMT Core
- SIMD Datapath
- Memory Subsystem
- Interconnection Network
- More Details in Paper

---

Wilson Fung, Tor Aamodt

Thread Block Compaction

Wilson Fung, Tor Aamodt

Thread Block Compaction

Wilson Fung, Tor Aamodt

Thread Block Compaction

19
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
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 → 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 (2)


Operation

Thread Compactor

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

```
1 5
P-Enc
5
```

```
2 -- 7 8 -- 11 12
P-Enc
```

```
Warp Buffer
C 1 2 7 8
C 5 -- 11 12
```

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

```java
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

```
A
B
C
E
D
F
```

```
Rarely Taken

iPDom of A
```
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>
```

- 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

![Graph showing speed vs. No. LCP for different benchmarks](image)

Evaluation

- Simulation: GPGPU-Sim (2.2.1b)
  - ~Quadro FX5800 + L1 & L2 Caches
- 21 Benchmarks
  - All of GPGPU-Sim original benchmarks
  - Rodinia benchmarks
  - Other important applications:
    - Face Detection from Visbench (UIUC)
    - DNA Sequencing (MUMMER-GPU++)
    - Molecular Dynamics Simulation (NAMD)
    - Ray Tracing from NVIDIA Research

![Graphs showing max stack usage for different benchmarks](image)
Experimental Results

- 2 Benchmark Groups:
  - COHE = Non-Divergent CUDA applications
  - DIVG = Divergent CUDA applications

![IPC Relative to Baseline](image)

Serious Slowdown from pathologies
No Penalty for COHE
22% Speedup on DIVG

Per-Warp Stack

Effect on Memory Traffic

- TBC does still generate some extra uncoalesced memory access

![Memory Traffic](image)

No Change to Overall Memory Traffic In/out of a core
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
/* Example from _global void mem_layerforward CUDA() kernel of BACED.*/
lib 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_node[ty + index_in];
__syncthreads();
for (i = 0; i < _log2(HEIGHT); i++)
    // Loop-end branch: Conditional branch that is non-divergent
```

Compaction ineffective branch
Loop executes a fixed number of times

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

Compaction-Resistant Branches

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)

```c
for (i = 1; i < \_log2(HEIGHT); i++) {
  ...
  // Loop-end branch: Conditional branch that is non-divergent
}
```

![Control flow graph](Image)

Impact of Ineffective Compaction

Threads shuffled around with no performance improvement

However, can lead to increased memory divergence!
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
### 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

---

**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

---

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

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
Summary

• 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.

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

Code #4) Branch dependent on programmatic values - (ii)

```c
0  // Code snippet from the kernel of Mandelbrot benchmark
1  // imageW and imageH are scalar input parameters of the kernel
2  const int ix = blockDim.x * blockIdx.x + threadIdx.x;
3  const int iy = blockDim.y * blockIdx.y + threadIdx.y;
4  ...
5  if (ix < imageW && iy < imageH)
6      |
7      ...
8  }
9  }
```

<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 W0</td>
<td>TID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>For W1</td>
<td>TID</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>For W2</td>
<td>TID</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>For W3</td>
<td>TID</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
</tr>
</tbody>
</table>

Threads that are active when `(j == 1)`, having `(i[i] > tid) == true`

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

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

(77)

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

(78)
Impact of Lane Reassignment

Goal: Improve “compactability”

![Diagram showing lane reassignment examples.]

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

![Diagram showing random permutations examples.]

Figure from M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation,” ISCA 2013
Mapping Threads to Lanes; New

What criteria do we use for lane assignment?

Balanced Permutation

Even warps: permutation within a half warp

Odd warps: swap upper and lower

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

Figure from M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation," ISCA 2013
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>000</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>011</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

$XOR_{evenWID} = \frac{evenWID}{2}$

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

Figure from M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization on GPGPUs with SIMD Lane Permutation," ISCA 2013
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
Distinguish between compaction rate and utilization!

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,