Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance

Jiayuan Meng, David Tarjan, Kevin Skadron
LAVAlab
Department of Computer Science
University of Virginia
Motivation

• SIMD lockstep execution must wait for the slowest lanes
  – Cache misses (scatter/gather)
  – Branch divergence
DWS outline

• Background and problem statement
• Divergence and possible solutions illustrated
• Experimental results
• Related work and conclusions
Central Idea

• SIMD better than MIMD for some applications
  – Highly data parallel applications can use many lanes
  – SIMD saves IF, ID, and issue logic -> lower area, power
  – Many modern examples including SSE2, Cell, Larrabee; Clearspeed, nVIDIA and ATI GPUs, Imagine/Merrimac…

• But, SIMD has performance challenges when threads diverge

• Dynamic Warp Subdivision
  – a set of HW (and SW) optimization techniques
  – help mitigate divergence penalties
Basic Definitions

• “SIMD” = Single Instruction, Multiple Data
  – Vector: a few wide registers in a shared RF
    • Vector SIMD is explicit in software
  – Array: each scalar element has its own (scalar) RF
    • Array SIMD usually implicit, aka SIMT or “Single Instruction, Multiple Thread”

• “Lane” = Pipeline of scalar processing units
• “Thread” = Instruction stream for a lane
• “Warp” = Threads operating in lockstep
• “Warp Processing Unit” (WPU)
  – SIMT HW block with a fixed number of lanes
  – Lanes are simple scalar in-order pipelines
  – Lanes share I-cache and L1 D-cache
Motivation: Branch Divergence

• *Branch divergence* and role of the *re-convergence stack*
  
  – Threads in the same warp may branch to different control paths
  – The WPU chooses one path and executes corresponding threads
  – A bit mask is pushed to the re-convergence stack to mark the active threads
  – *Post-dominators* signal where diverged threads can be re-converged.
Motivation: Memory Latency Divergence

• *Memory Latency Divergence*
  
  – Threads from a single warp experience different memory-reference latencies caused by cache misses or accessing different DRAM banks
  
  – The entire warp must wait until the last thread has its reference satisfied
  
  – Often occurs during *gather* or *scatter*
SIMT Performance Tradeoffs

• Why not increase SIMT width?
  – Exploit more data parallelism
    • But, more branch divergence and likelihood of memory latency stalls

• Why not deeper multi-threading?
  – More opportunity for warp latency hiding
  – But, too many total threads not feasible
    • Fast warp switching requires an RF per thread—large area overhead for many threads
    • Too many in-flight threads can cause cache thrashing (Meng et al., ICCD’09, Guz et al., CAL’09)

• Dynamic warp subdivision (DWS) finds a balance
Key Observations

• Without adding more warps, exploit *intra*-warp latency hiding
  – Allow threads that are unnecessarily suspended to run ahead

• Warps can be subdivided into warp-splits; each can be regarded as an independent scheduling entity
  – Initiates cache misses earlier and/or prefetches for other threads

• Challenges of warp subdivision
  – When to subdivide? When to re-converge? How?
Subdividing on Branches—Benefits (1)

• Different branch paths hide each other’s latency
Subdividing on Branches—Benefits (2)

- Memory requests beyond the post-dominator can be issued earlier

(a) Conventional execution for a diverged branch

(b) Ideal execution for a diverged branch
Subdividing on Cache Misses—Benefits (1)

- Threads in the same warp hide each other’s latency

(a) Conventional execution for a memory divergence

(b) Ideal execution for a memory divergence
Subdividing on Cache Misses—Benefits (2)

- Runahead threads prefetch data for fall-behind threads
Implementation: Reconvergence Challenges

• Over-subdivision:
  – Aggressively subdividing warps may lead to many narrow warp-splits, which can otherwise run altogether as a wider warp.
  – Judiciously select branches to subdivide warps; don’t allow deep nesting
  – Subdivide only when all available warp-splits are suspended

• Unrelenting subdivision (failure to reconverge):
  – When warp-splits independently execute the same instruction sequence while there is not much latency to hide, SIMD resources are under-utilized for little benefit.
  – PC-based re-convergence
  – On every cycle, match PCs within active warp
  – Not latency critical due to multithreading
  – Reconvergence still guaranteed to happen at the PD registered on the top of the reconvergence stack
Implementation: DWS Upon Branch Divergence

- Use a *warp-split table* to handle some branches instead of the re-convergence stack and post-dominators
- Warp-split Table: Each entry corresponds to a warp-split. Using bit masks to keep track of threads belonging to the same warp-split.
- Hazard: threads do not re-converge at F, risking pipeline under-utilization

![Diagram of program execution and warp split table](image)
Methodology: Simulation

- Simulation using MV5, an event-driven, cycle-level multicore simulator based on M5
  - Four cores in this study
  - Two level cache hierarchy, private I- and D-caches share the same L2 cache
  - IPC of 1 for all instructions except memory ops
  - Alpha ISA
  - D-caches banked to match lanes, conflicts are modeled
  - Coalescing performed using MSHRs
  - Multiported TLB
### Methodology: Benchmarks

- **Selected from SPLASH2, MineBench, and Rodinia**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform (SPLASH2)</td>
<td>1D array, 64K</td>
</tr>
<tr>
<td>Filter</td>
<td>Edge Detection—convolution, 3x3 stencil</td>
<td>Grayscale 2D image, 500x500</td>
</tr>
<tr>
<td>HotSpot (2D)</td>
<td>Heat transport, 4-pt stencil (Rodinia)</td>
<td>300x300 2D grid, 100 iterations</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition (SPLASH2)</td>
<td>300x300 2D matrix</td>
</tr>
<tr>
<td>Merge</td>
<td>Mergesort</td>
<td>1D array, 300,000 ints</td>
</tr>
<tr>
<td>Short</td>
<td>Path search in chess, dynamic programming</td>
<td>6 steps each with 150,000 choices</td>
</tr>
<tr>
<td>KMeans</td>
<td>Iterative clustering (MineBench)</td>
<td>10,000 points in a 20-D space</td>
</tr>
<tr>
<td>SVM</td>
<td>Support vector machine (MineBench)</td>
<td>100,000 vectors with a 20-D space</td>
</tr>
</tbody>
</table>
Stack vs PC Reconvergence

- Using reconvergence stack, reconvergence is forced at *immediate* PD even if warp-splits must stall to wait
- PC-based reconvergence uses WST
  - Allows for dynamic re-convergence of warp-splits, even from different loop iterations!
- To avoid over-subdivision, statically pick which branches to use
- WST vs RS
  - Use heuristic of short post-dominators
  - This promotes earlier reconvergence and also avoids unrelenting subdivision
- Once in WST regime, can only further split using WST

![Graph showing speedup comparison between stack-based and PC-based reconvergence]
DWS Strategies for Miss Reconvergence

• Branch Limited Reconvergence
  – After a memory-latency divergence, reconverge at next branch/PD
  – Can use reconvergence stack
  – Limits usefulness of DWS on MD

• BranchBypass
  – Use WST to allow warp-split to live beyond next branch or PD (but a PD registered on the reconvergence stack will prevent bypass)
  – Synergy between BD and MD solutions!

• Heuristically schedule widest warp-split first
  – Tends to generate cache misses earlier
  – Therefore, helps to hide cache miss latency
Results (1)

- Each WPU has four 16-wide warps
- 32 KB, 8-way associative D$
- 4 MB, 16-way associative L2
- Speedups relative to conventional branch/miss handling

**DWS.BranchOnly**: Subdivide on branch divergence alone
**DWS.AggressiveSplit**: Subdivide on every divergent cache access
**DWS.LazySplit**: Subdivide when there is only one active warp left
**DWS.ReviveSplit**: Same with DWS.LazySplit, but also being able to subdivide suspended warp-splits
**DWS.ReviveSplit.MemOnly**: Subdivide on memory latency divergence only
**Slip**: Adaptive slip which synchronize threads upon branches, without aggr. predication
**Slip.BranchBypass**: Adaptive slip that allows runahead threads to bypass branches, but still without aggressive predication
Results (2)

• Subdividing upon branches or memory accesses alone leads to modest speedup

• Subdividing upon both branch and memory latency divergence yield 1.7X speedup on average

• No performance degradation is observed on any benchmarks

• Outperforms adaptive slip when aggressive predication is not available

• Area overhead is less than 1%
Related Work

• Adaptive Slip (Tarjan et al., SC’09)
  – Use re-convergence stack to mark threads that can run ahead, and merge with the fall behind when the run-ahead get to the same PC (assuming loops)
  – Drawbacks:
    • Forces reconvergence at branches
    • Relies on aggressive predication
    • The threshold to increase and decrease divergence is application-specific

• Dynamic Warp Formation (Fung et al., MICRO’07)
  – Merge from different warps threads that fall into the same branch path
  – Increases pipeline utilization
  – Less latency hiding; does not address divergent memory accesses
Conclusions

• DWS can help balance warp width and MT depth to hide latency and leverage MLP
• Subdivide warps upon divergence
  – Same physical storage but independent scheduling entities.
• Only simple HW structures such as WST are needed
  – Area overhead is less than 1%
• PC reconvergence with WST aids both BD and MD
• DWS provides a 1.7X speedup over conventional SIMT
  – Yet it is never worse across wide range of applications, cache sizes and associativities
• DWS is more general than Adaptive Slip, and outperforms it by 30%
Questions?

• Also see the TR for extended results
Backup slides
WST Hardware Overhead

- Number of warp-groups increases complexity of thread scheduler
- So, limit WST to twice the number of warps
- Only 2.2% worse for ReviveSplit
- WST entries
  - 16 bits for active mask (if warps 16 wide)
  - 2 bits for warp ID (if 4 warps)
  - 2 bits for warp status
  - 64 bits for PC
  - 4 bits for schedule priority
- Total WST size is less than 1% of WPU die area
## Simulation Details

<table>
<thead>
<tr>
<th>Tech. Node</th>
<th>65 nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>Alpha ISA, 1.0 GHz, 0.9V Vdd. in-order. 16-way multithreaded: two SIMT groups of width eight</td>
</tr>
<tr>
<td>L1 Caches</td>
<td>physically indexed, physically tagged, write-back 16 KB I-cache and 16 KB D-cache, 32 B line size 16-way associative, 16 MSHRs, 3 cycle hit latency, 4 banks, LRU</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>16 banks, 1024 KB, physically indexed, physically tagged 16-way associative, 128 B line size, LRU, 32 cycle hit latency write-back, 64 MSHRs, ( \leq 8 ) pending requests each</td>
</tr>
<tr>
<td>Interconnect</td>
<td>crossbar, 300 MHz, 57 Gbytes/s</td>
</tr>
<tr>
<td>Memory Bus</td>
<td>266 MHz, 16 GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>50 ns access latency</td>
</tr>
</tbody>
</table>
Implementation: DWS Upon Memory Latency Divergence

- Subdivide warps according to a “hit mask” that marks threads that hit the D-cache.
- Use the same warp-split table as in handling branch divergence!
- Allow run-ahead threads to bypass branches to issue more memory requests early
DWS Sensitivity Analysis
DWS vs Adaptive Slip

- Adaptive Slip allows some threads to run-ahead on MD, but stalls others until a synch point.
- Adapts the amount of PC divergence allowed between thread groups.
- Increments divergence if WPU spends > 70% of time waiting on memory accesses.
- Decrement divergence if WPU spends < 50% of time waiting on memory accesses.
- How to determine correct thresholds?
- Slip.BranchBypass an improvement.
DWS vs Dynamic Warp Formation

• Dynamic Warp Formation
  – Allows threads from different warps to execute together if they all have same PC
  – Use of RS forbids run-ahead threads
  – Thread-groups can’t interleave
  – Added RF indexing complexity due to arbitrary collection of RF requests
Future Directions

• Compiler must statically decide BD split points
• Use speculation in ReviveSplit to pick best warp group to split
• Static or Dynamic Prediction for PC-based WR to decide when to reconverge warp groups
• Merging DWS with Dynamic Warp Formation
• Dynamic restriction of total threads when cache thrashing