

# Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow



AUBURN

---

UNIVERSITY

W. W. L. Fung et al.  
(University of British Columbia)

MICRO 2007 & HPCA 2011



# Outline

- **Introduction**
- Hardware Model
- Dynamic Warp Formation (DWF)
- Warp Barriers
- Conclusions



# Introduction: NVIDIA GPU Threading Model

- GPGPU views graphics processors as massively thread- and data-parallel math co-processors
  - CPUs offload computationally intensive code to GPUs
  - CPUs perform I/O and coordinate activities
    - GPUs becoming more autonomous: CUDA4.0
- CPU threads activate kernels as grids of blocks of SIMD warps of threads
  - Overall somewhat distinct from traditional SIMD
  - MIMD > SPMD > SIMT > SIMD
  - Difference has to do with branch divergence



# Introduction: SIMD Scheduling Problem

- Branch divergence occurs when a warp's threads exhibit distinct control-flow behavior
  - Warp: scheduling unit consisting of  $n$  (e.g., 16 or 32) contiguous (based on thread ID) threads
- Distinct control-flow means different instructions, but SIMD HW can't execute different instructions in lockstep
- Punchline: to guarantee correct program behavior, branch divergence must be specifically addressed at some level
  - Don't want to remove SIMT (weak SPMD) support
  - Handle divergence at a lower level!



# Introduction: Naïve Solution

```
__global__ void foo(float *a, float *b, float c)
{
    int n = blockIdx.x * blockDim.x
           + threadIdx.x;           // B1
    if (n % 2 == 0) {               // Br
        a[n] += b[n];              // B2
    } else {
        a[n] -= b[n];              // B3
    }
    a[n] = 0.5f * (a[n] + b[n]);    // B4
    b[n] = b[n] / c;                // B5
}
```

thread0	thread1
B1	B1
Br	Br
B2	-
-	B3
B4	-
-	B4
B5	-
-	B5

↓ time

- At a branch, serialize threads in a warp
  - For complex kernels, results in lots of serialization
  - Threads assumed divergent; stalls at every instruction
  - Lose all benefit from SIMD pipeline



# Introduction: Post-Dominator Solution


```
__global__ void foo(float *a, float *b, float c)
{
    int n = blockIdx.x * blockDim.x
           + threadIdx.x;           // B1
    if (n % 2 == 0) {               // Br
        a[n] += b[n];               // B2
    } else {
        a[n] -= b[n];               // B3
    }
    a[n] = 0.5f * (a[n] + b[n]);    // B4
    b[n] = b[n] / c;                // B5
}
```

- Post-dominator: all branches eventually reach this point
  - e.g. (B4 pdom Br) and (B5 pdom Br)
- Immediate post-dominator: “closest” post-dominator
  - e.g. only (B4 immediate-pdom Br)



# Introduction: Post-Dominator Solution

- Can re-converge at any post-dominator
  - Intuitively, immediate post-dominator is best
  - For some pathological code, it is not
  - For most real applications, post-dominator is nearly as good as an oracle
- Helps increase parallel efficiency
- Easy to support (even at compile-time)
- Loses some efficiency compared to MIMD
- Is there a better way?

thread0	thread1	
B1	B1	
Br	Br	
B2	-	
-	B3	
B4	B4	
B5	B5	



# Outline

- Introduction
- **Hardware Model**
- Dynamic Warp Formation (DWF)
- Warp Barriers
- Conclusions

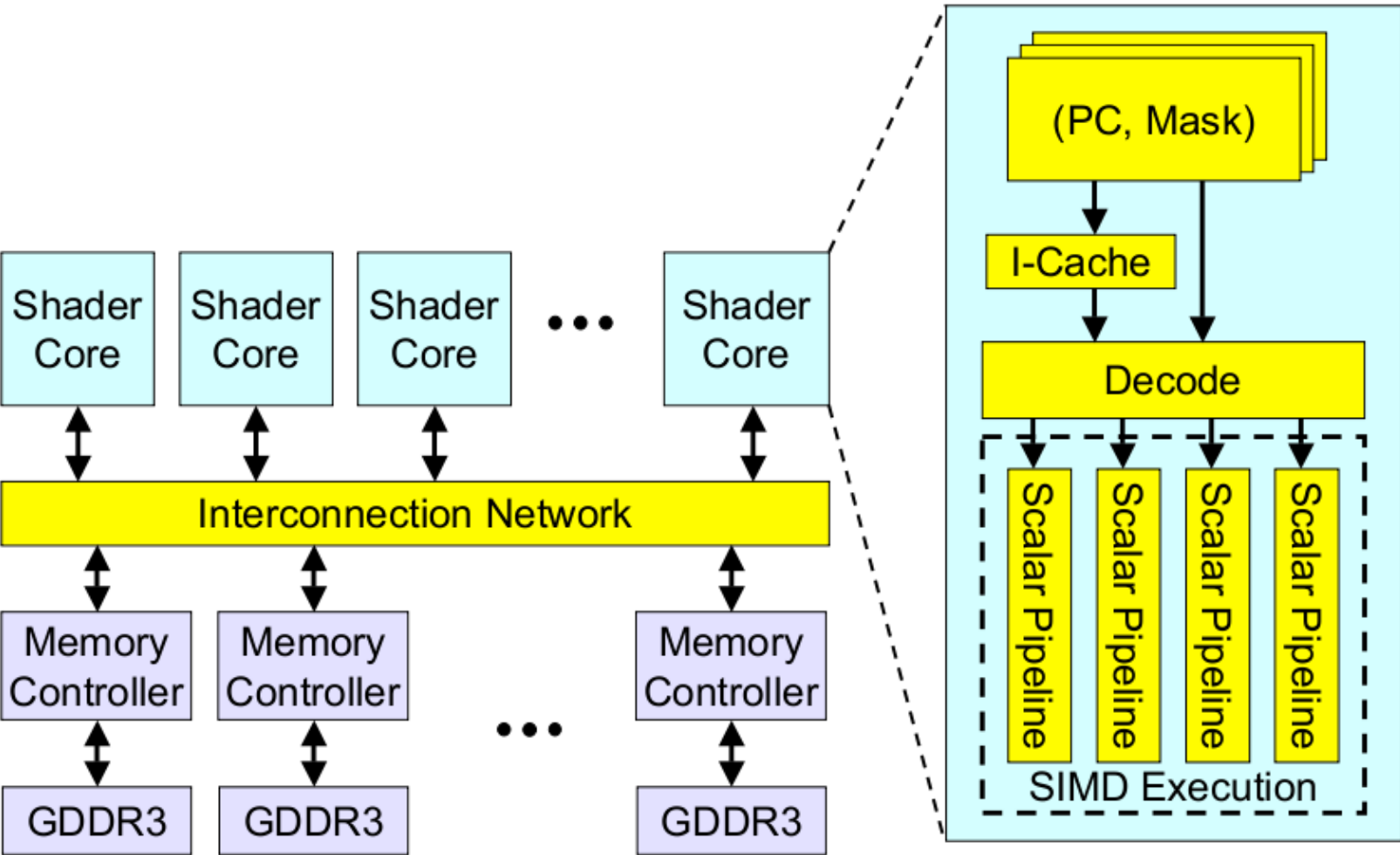


# Hardware Model – GPGPU & CUDA

- GPUs dedicate more die area to FP arithmetic
  - Branch prediction and caching are minimal
- NVIDIA's Compute Unified Device Architecture
  - Hierarchical memory and threading model
    - Kernels => grid > block > warp > thread
    - Global, constant, texture and shared memory + registers
  - Extensions to C++ for efficient GPU support

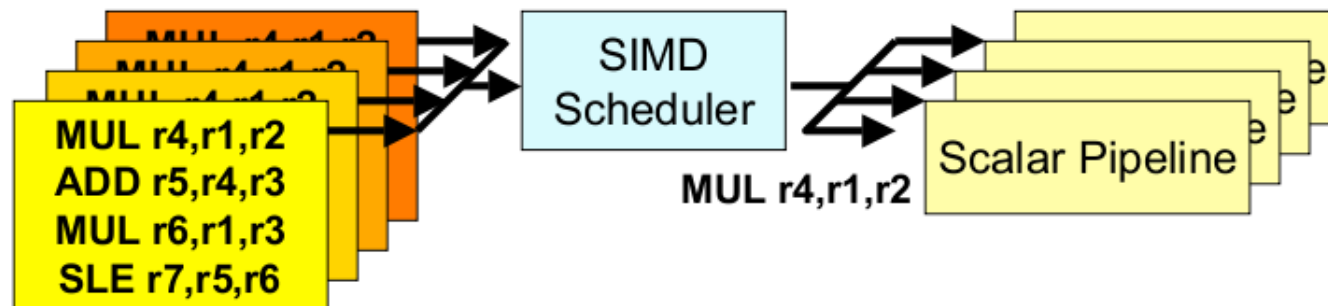


# Hardware Model – Streaming Architecture



# Hardware Model: Latency Hiding and SIMD

- Barrel processing proactively hides latency of potential stalls before they occur
  - Fetch unit interleaves instructions from threads
  - Threads waiting for memory removed from ready pool
- SIMD hardware supports SPMD programs when control flow is similar
  - Instructions from shader threads grouped into warp
  - Warps scheduled across multiple scalar pipelines



# Outline

- Introduction
- Hardware Model
- **Dynamic Warp Formation (DWF)**
- Thread Block Compaction (TBC)
- Conclusion



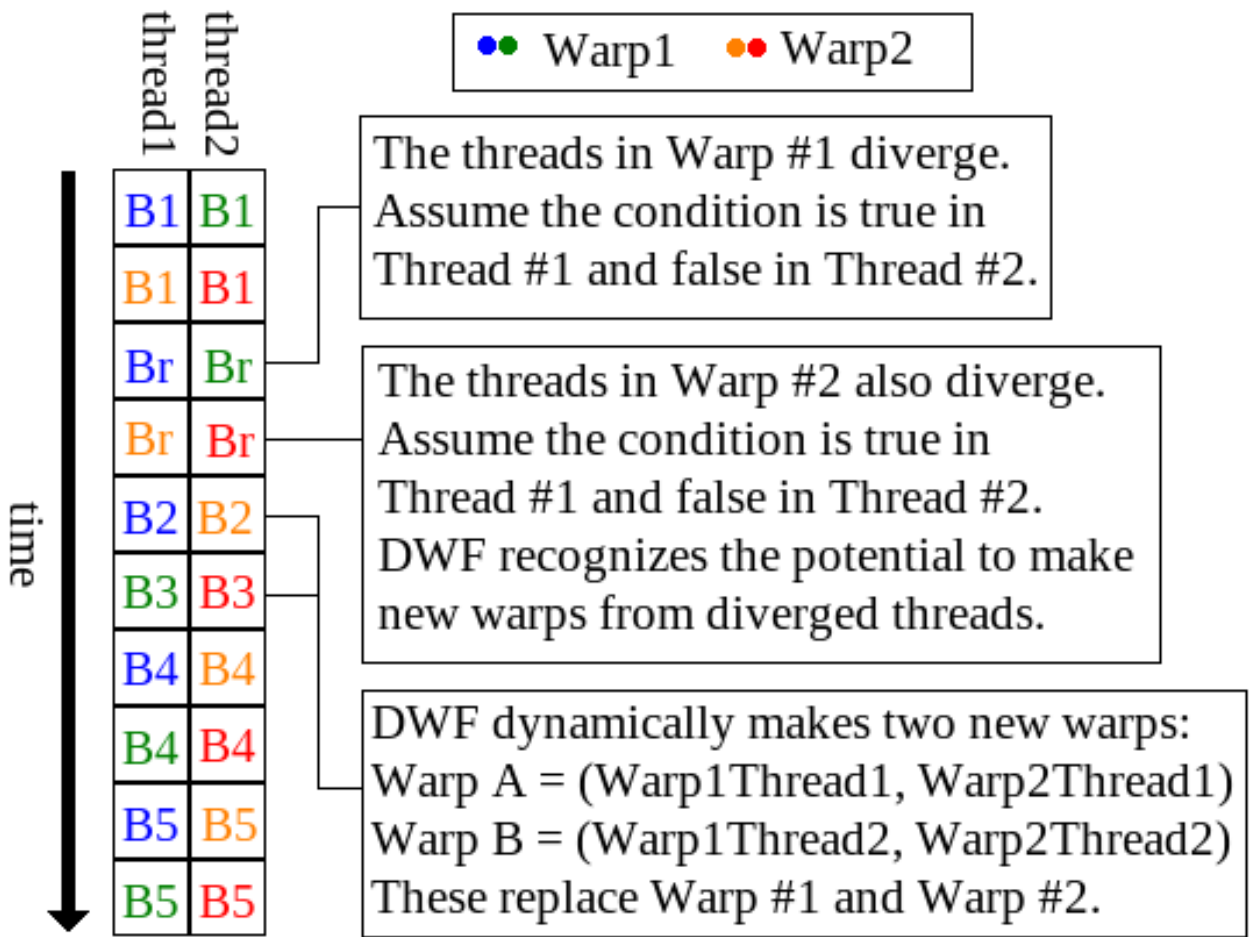
# Dynamic Warp Formation: Idea

- Blocks consist of several warps running same code
- If one warp diverges, why not others?
- Same idea as barrel processing:
  - If a warp diverges, let another run
  - Form full warps dynamically by merging similarly diverged threads from previous warps
  - Increases occupancy/utilization!

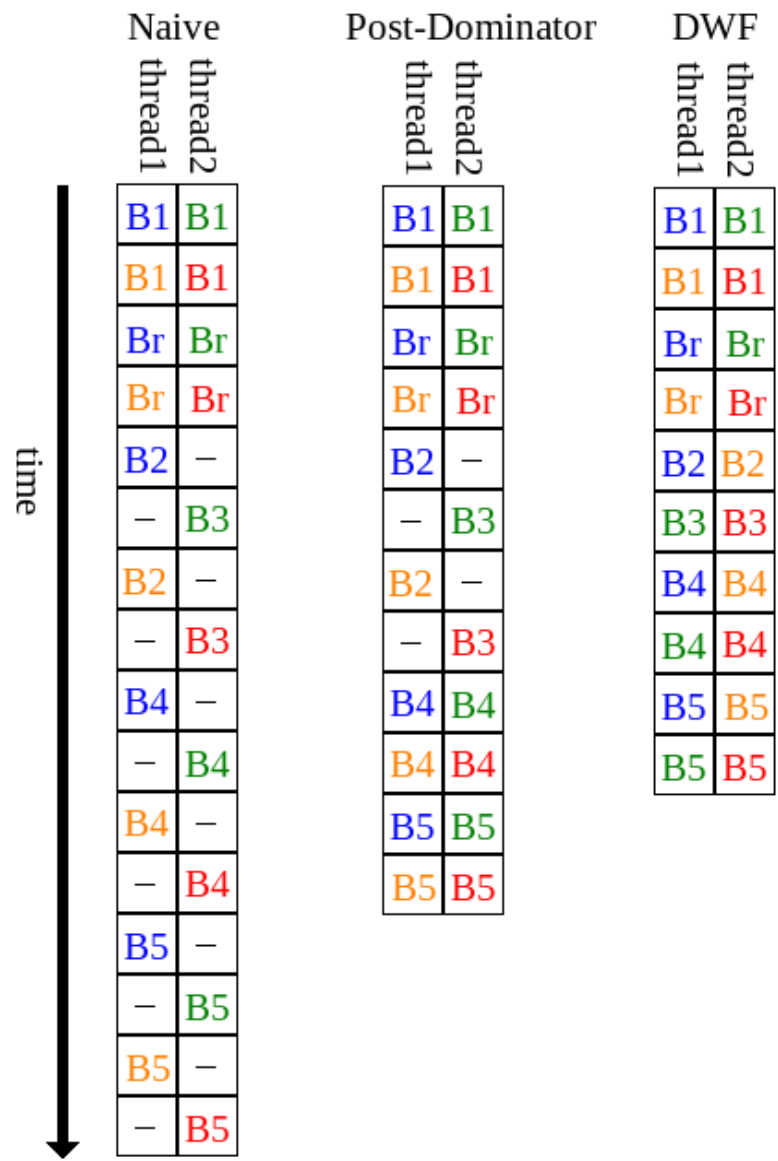


# Dynamic Warp Formation: Idea

- Assume two warps and two threads per warp. Then:



# Dynamic Warp Formation: Comparison



# Dynamic Warp Formation: Lane-Aware DWF

- Threads must access register file
- Don't want to migrate values
- Solution is straightforward:
  - Migrated threads access same registers as before
  - Threads are only dynamically combined if they do not cause bank conflicts (introducing stalls into pipeline)
    - This is called “lane-aware” DWF



# Dynamic Warp Formation: Lane-Aware DWF

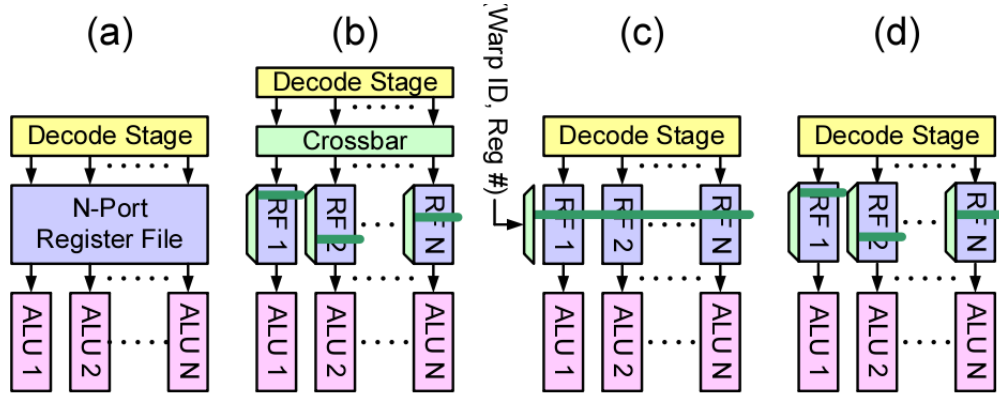


Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.

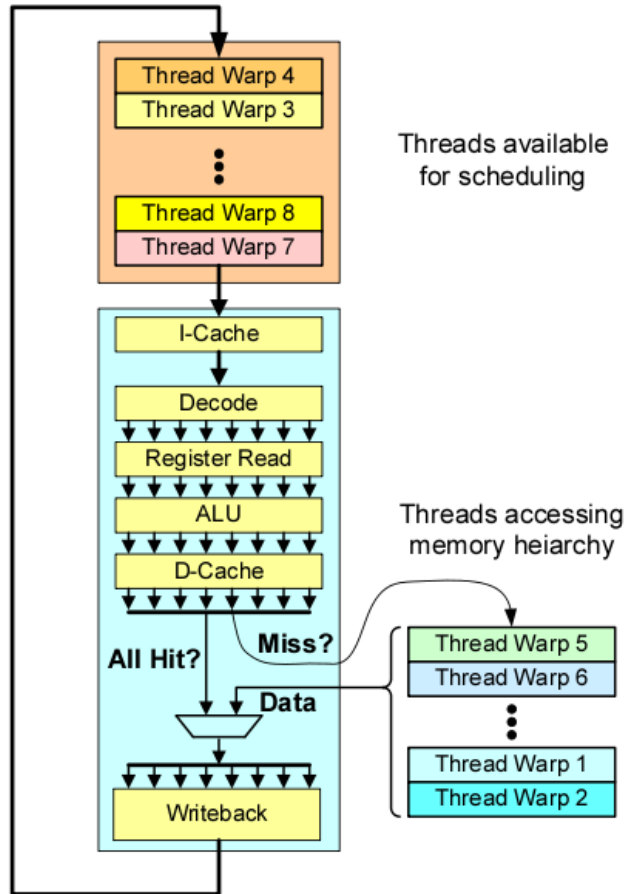


Figure 3. Using barrel processing to hide data memory access latency.



# Dynamic Warp Formation: Scheduling

- For DWF to work, threads need to progress at nearly the same rate (must have lots of the same PCs)
- The scheduling policy can affect the distribution of PCs
- Authors explore several heuristic policies to help DWF:
  - **Majority**: issue warps with the most common PC first
  - **Minority**: issue warps with the least common PC first
  - **Time stamp**: issue the oldest warp first
  - **Post-Dominator Priority**: issue warps having passed the fewest post-dominators first
  - **Program counter**: issue warps with lower program counters first



# Dynamic Warp Formation: Scheduling

- Authors' treatment of scheduling somewhat unsatisfactory
  - No “null hypothesis” or baseline
    - e.g. random, round-robin, LIFO, FIFO, etc.
  - Definitions of scheduling heuristics unclear
    - Priorities updated every cycle or every round?
    - Ties broken by thread ID or by other heuristics?
  - Selection of heuristics not motivated.
- Still, authors did try several different methods and saw different performance.



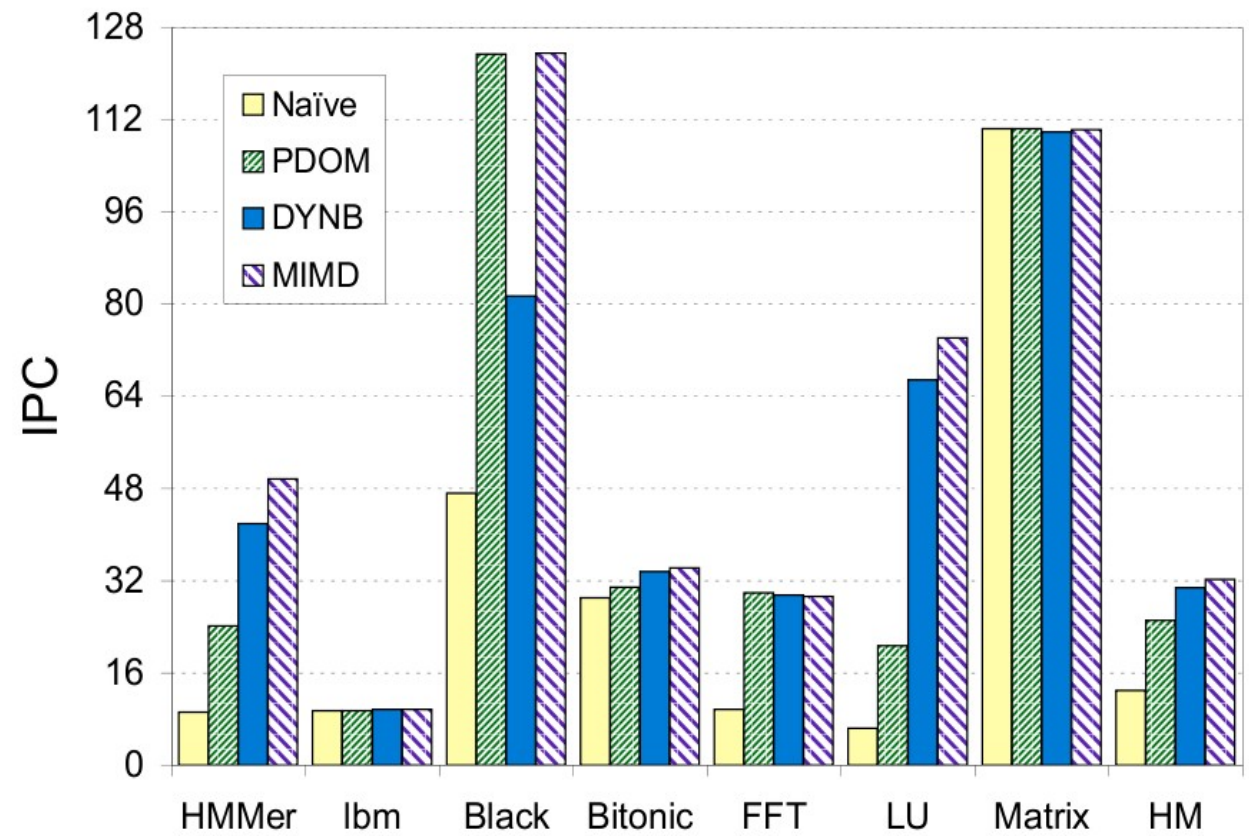
# Dynamic Warp Formation: Experiments

- Constructed simulator, GPGPU-sim, from SimpleScalar version 3.0d
- Benchmarks from SPEC CPU2006, SPLASH2, CUDA
- Performance tests:
  - Naive vs. PDOM vs. DWF
  - Heuristic scheduling policies
  - Lane-aware DWF vs. Lane-unaware DWF



# Dynamic Warp Formation: Results

- PDOM vs Naive: 93.4% speedup
- DWF achieves a further speedup of 22.5% using the Majority heuristic
- DWF performance only 4.6% worse than MIMD!



**Figure 12. Performance comparison of Naïve, PDOM, and DYNB versus MIMD.**



# Dynamic Warp Formation: Results

- Majority is the best, on average
- Post-dominator, program counter do well, too
- Minority and time stamp do poorly
- In some cases, Majority is worse: starvation eddies
- More work is possible!

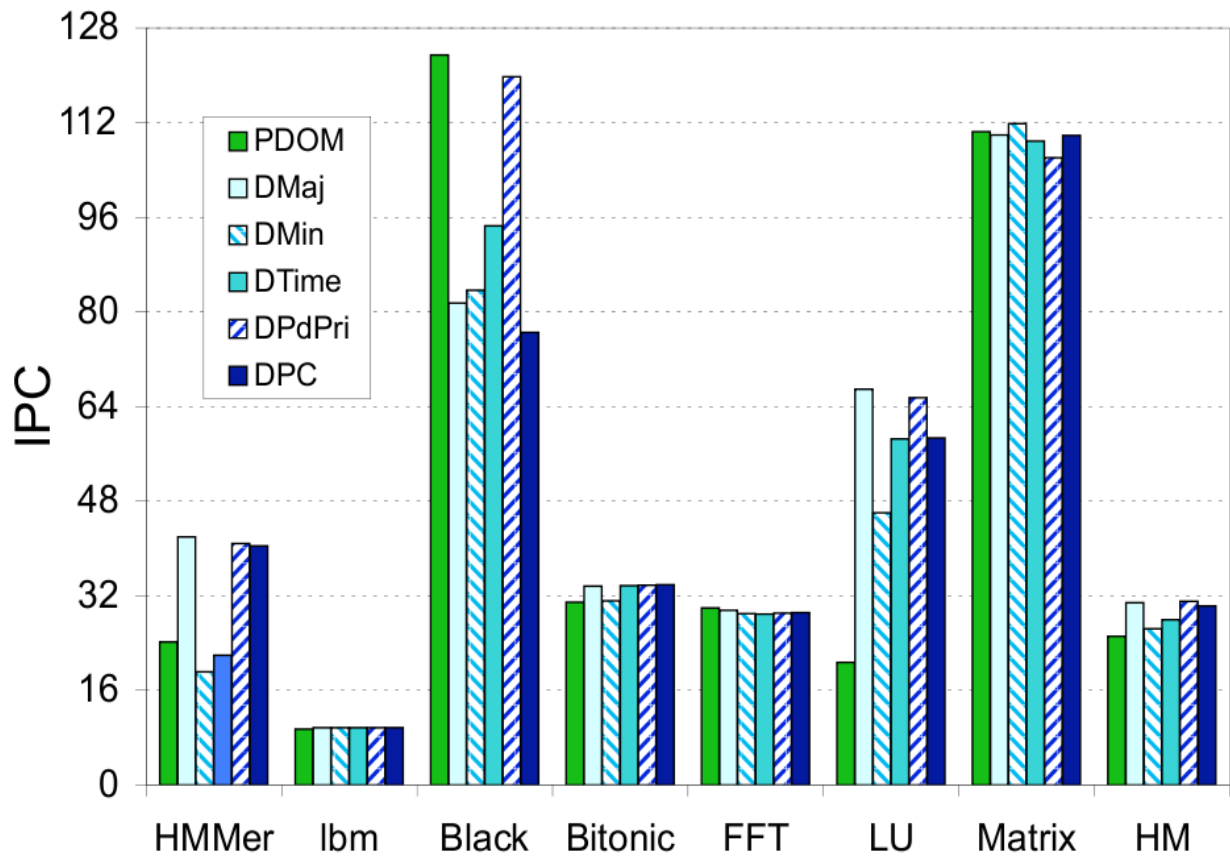
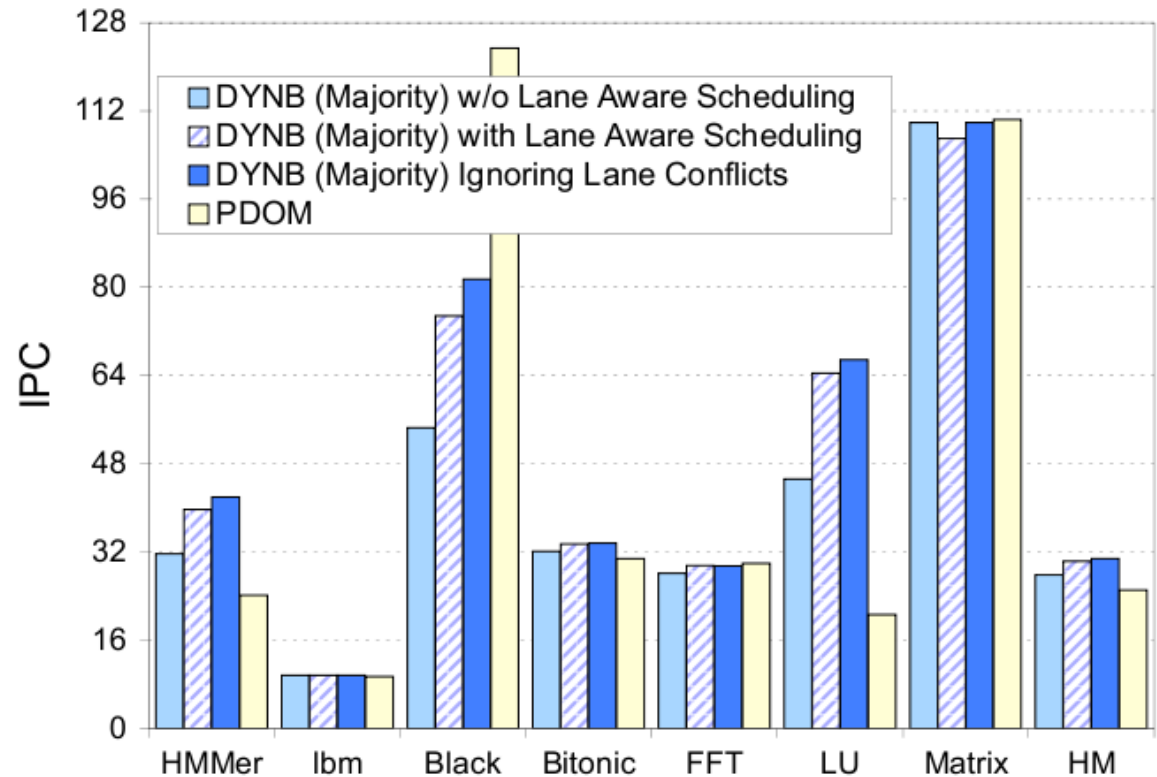


Figure 15. Comparison of warp issue heuristics.



# Dynamic Warp Formation: Results

- Surprisingly, performance is better when lane conflicts are completely ignored!
- Authors' response is pretty good:
  - Even though it's better to ignore lanes, it's still faster than PDOM, so it's good!



**Figure 17. Performance of dynamic warp formation with lane aware scheduling and accounting for register file bank conflicts and scheduler implementation details.**



# Outline

- Introduction
- Hardware Model
- Dynamic Warp Formation (DWF)
- **Warp Barriers**
- Conclusions



# Warp Barriers: Motivation

- Three problems with DWF as previously described:
  - Applications relying on implicit synchronization in a static warp (a really bad idea) don't work under DWF
  - “Starvation eddies” using Majority heuristic lead to reduced SIMD performance
  - DWF increases non-coalesced memory accesses and shared memory bank conflicts
- Addressing these problems is now the focus



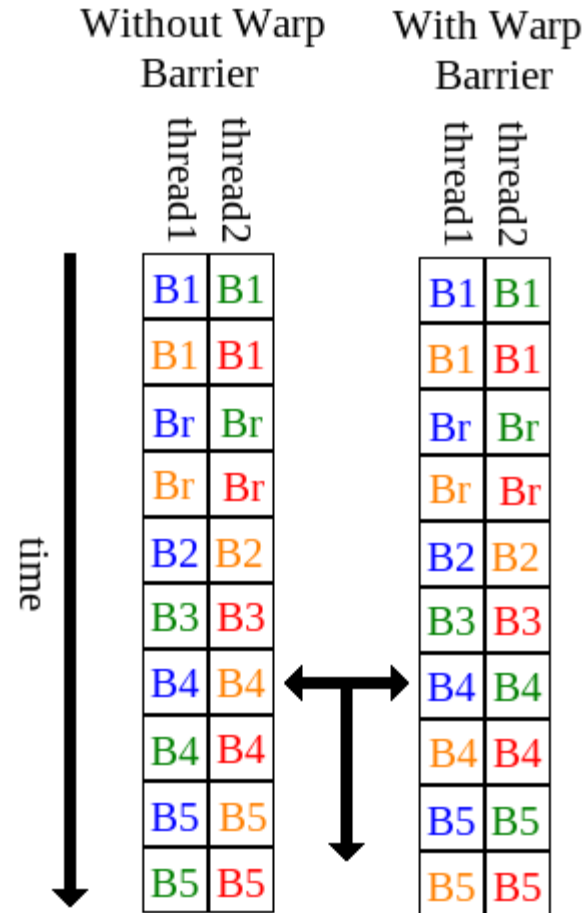
# Warp Barriers: Idea

- Warp barriers are a refinement of the original DWF
- Top-level divergent threads rearranged normally, except:
  - A per-warp “warp barrier” is set at the immediate post-dominator
  - When threads reach the warp barrier, original warps are reconstituted
- Several benefits:
  - Confines starvation eddies
  - Reduces memory conflicts/contention
  - Most importantly, solves correctness problem



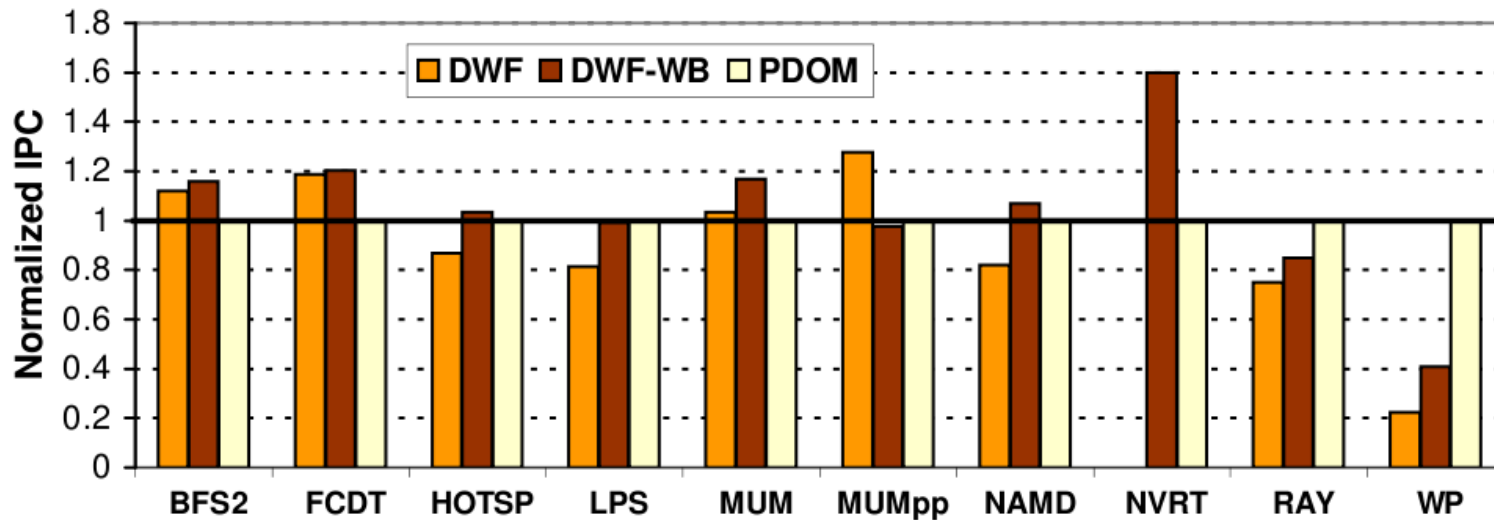
# Warp Barriers: Idea

- After threads from the original warps can reconverge, put them back in their original warps
- Indeed, threads wait for their fellow threads to make it to the warp barrier
- Synchronization does not help performance; improving memory access and confining starvation eddies does
- Parallelism traded for memory access & scheduling efficiency



# Warp Barriers: Performance

- Warp barriers improve performance in all cases (except for MUMmerGPU++...)
- DWF not reported for NVRT – DWF causes incorrect execution, which warp barriers fix!



**Figure 6.** DWF with and without warp barrier compared against baseline per-warp PDOM stack reconvergence.



# Outline

- Introduction
- Hardware Model
- Dynamic Warp Formation (DWF)
- Warp Barriers
- **Conclusions**



# Conclusions

- GPGPU offers lots of parallel computation capacity
- SIMT allows SPMD programs to run on SIMD hardware
- Making SPMD perform well on SIMD is the challenge!
- Several solutions come to mind:
  - Naive: addresses correctness, not performance
  - PDOM: intuitive solution for correctness/performance
  - DWF: an idea to get better utilization
  - Warp Barriers: a refinement to address memory
- Other approaches and solutions are possible; active area of research.



# Questions?

