Improving GPU Performance via Large Warps and Two-Level Warp Scheduling

Veynu Narasiman† Michael Shebanow‡ Chang Joo Lee¶
Rustam Miftakhutdinov† Onur Mutlu § Yale N. Patt†

†The University of Texas at Austin ‡Nvidia Corporation ¶Intel Corporation §Carnegie Mellon University

MICRO 2011

Presented by Ondrej Cernin
5.12.2018

Executive Summary

- GPU performance suffers performance penalties due to
  - Branch divergence
  - Long latency operations
- Paper proposes two new mechanisms to improve GPU performance by better resource utilization
  - **Large warp microarchitecture**
  - **Two-level round warp scheduling**

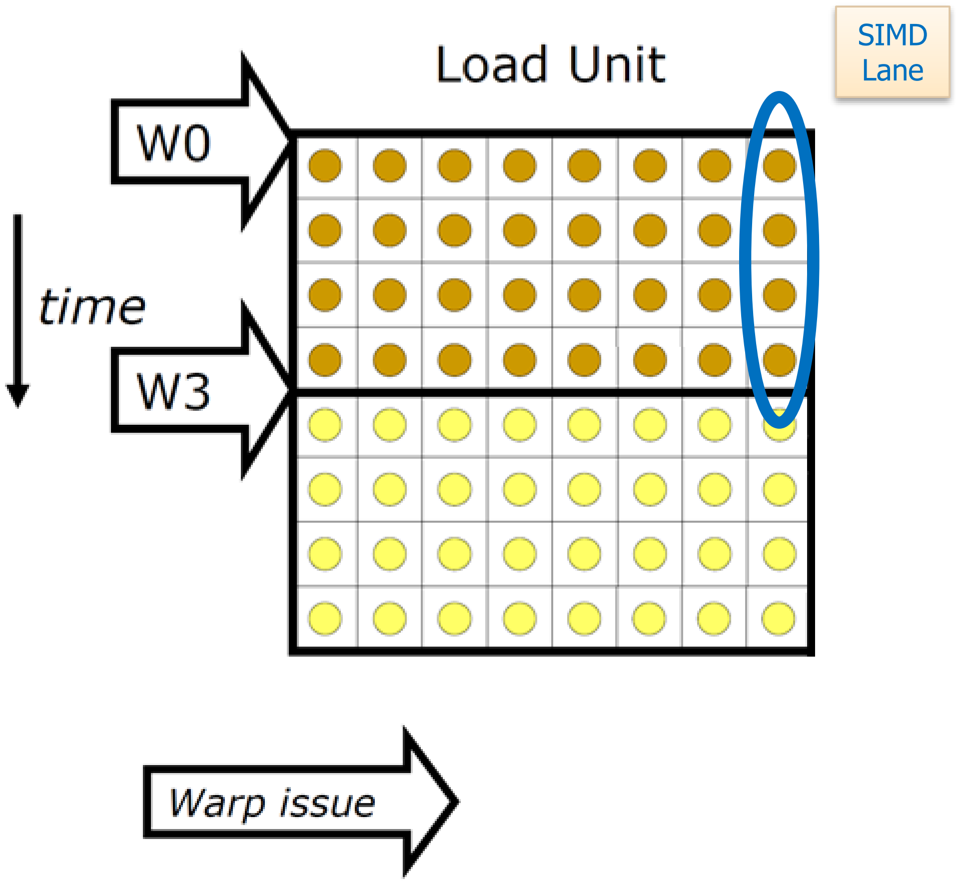
Background

- Graphic Processing Units (GPUs)
  - SIMD (single instruction, multiple data)
    - multiple functional units
  - Exploits *Thread-Level Parallelism (TLP)* exposed by the programmer
    - *Warps* – batches of threads executing the same code
    - GPU's concurrently execute many warps on a single core
      - Can execute on a different warp when one is stalled

Warp Instruction Level Parallelism

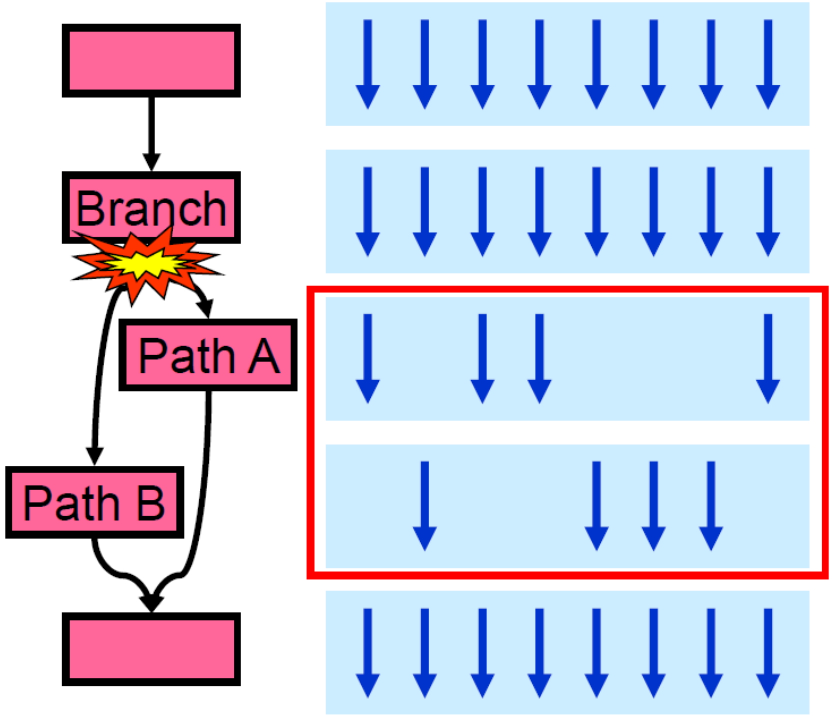
Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes



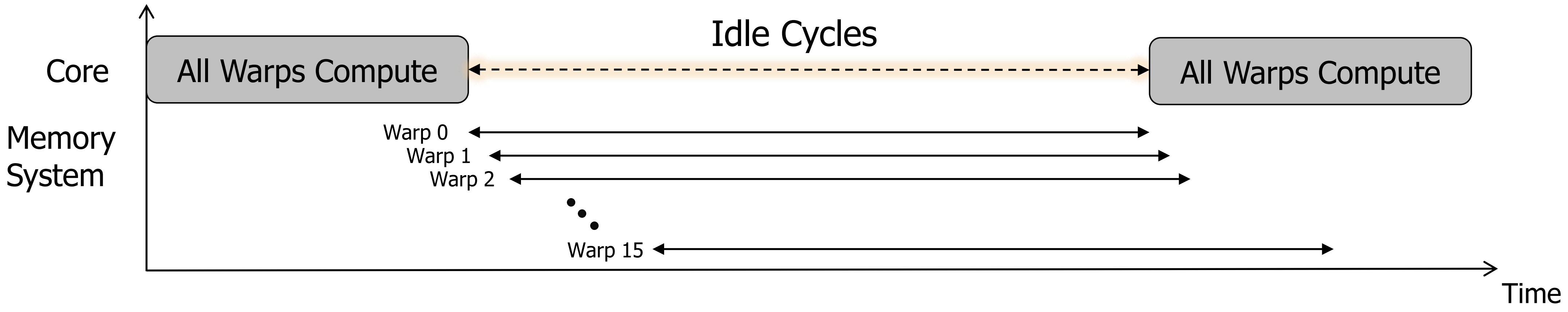
Problem 1

- Underutilization of the computational resources on a GPU core
  - GPU only efficient if threads remain on the same dynamic execution path
  - Conditional branch instructions cause threads to ***diverge***
  - GPU implementation allow only one PC (program counter) at a time for a warp



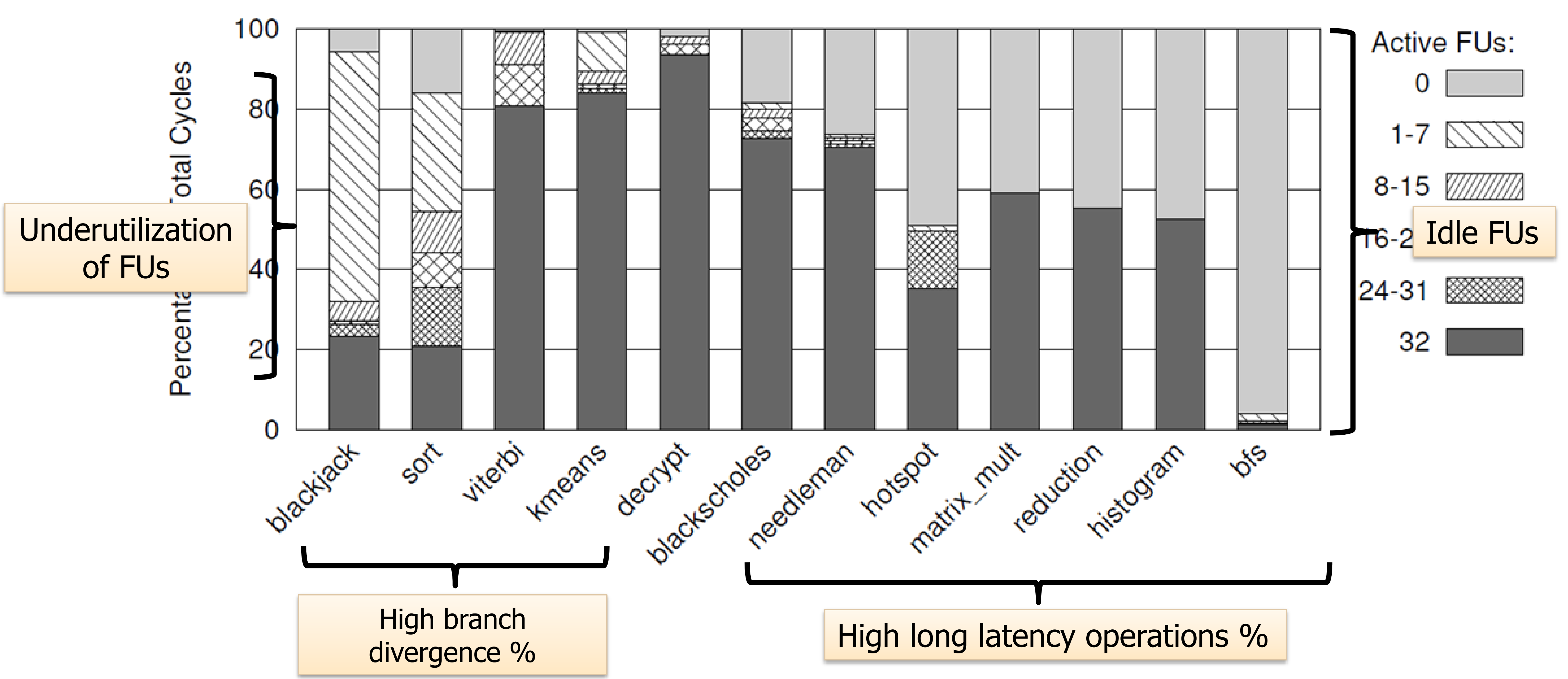
Problem 2

- Long latency operations
  - Warp instruction fetch policy on a GPU core can affect the total latency
    - i.e. *round-robin scheduling* with equal warp priority leads to all warps arriving at the same long latency at the same time
    - Allowing warps to progress at very different rates can result in starvation and destroy data locality



Baseline GPU round-robin scheduling with 16 warps

Computational Resource Utilization



Goals and Key Ideas

Problem 1 – Branch divergence

- Improve GPU performance by better utilizing computational resources
- Proposal: ***Large Warp Microarchitecture* (LWM)**

Problem 2 – long latency instructions

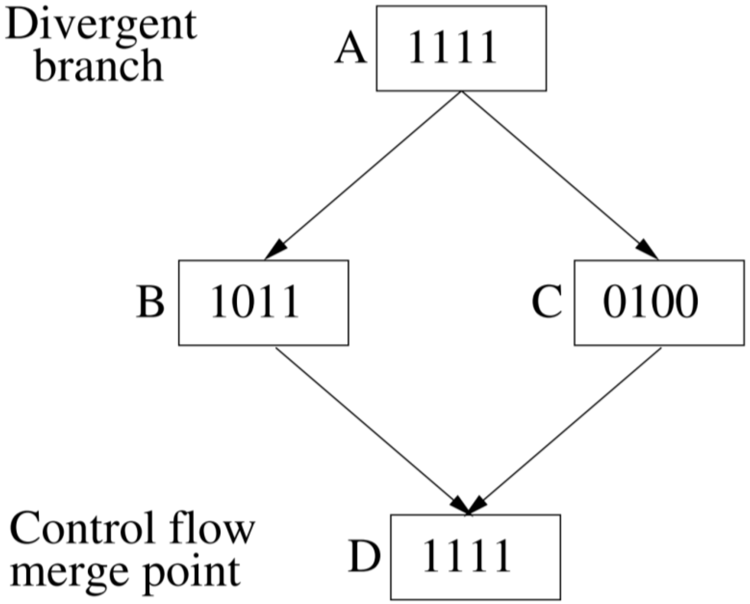
- Reduce the number of idle FU cycles
- Proposal: ***two-level round-robin*** warp instruction fetch scheduling policy

- … And combine both to achieve maximum speedup

Mechanisms

Conditional Branch Handling

- A warp can only have a single active PC at any given time
  → ***Branch divergence***
  - One path must be chosen first and the other is pushed on a divergence stack
- Divergence stack
  - Used to bring warp back together → *Control flow merge* (CFM) point

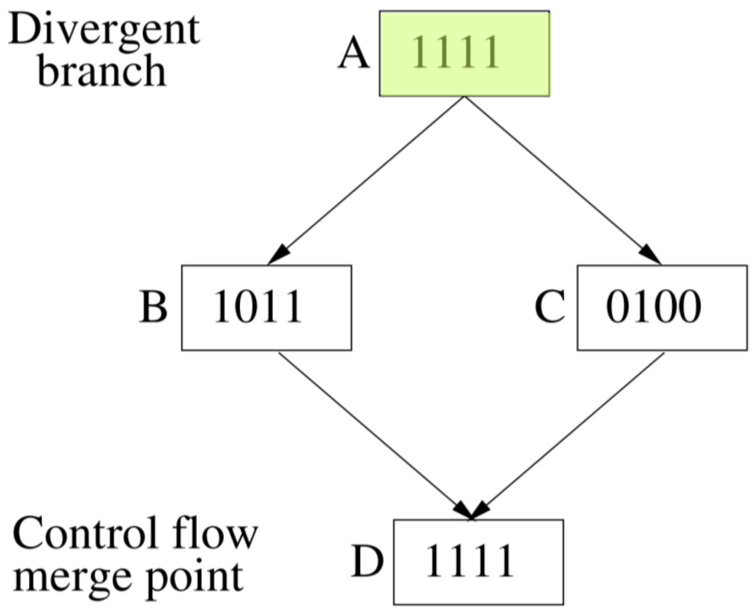


Current PC: A
Current Active Mask: 1111

Reconvergence PC	Active Mask	Execute PC

Conditional Branch Handling

1. When a divergent branch is reached, push a *join* entry onto the divergence stack.
   - Re-convergence PC and Execute PC are equal to the control flow merge point
   - Active mask field is set to the current active mask
   - One of the two divergent paths is selected to be executed first, while the other is added to the divergence stack

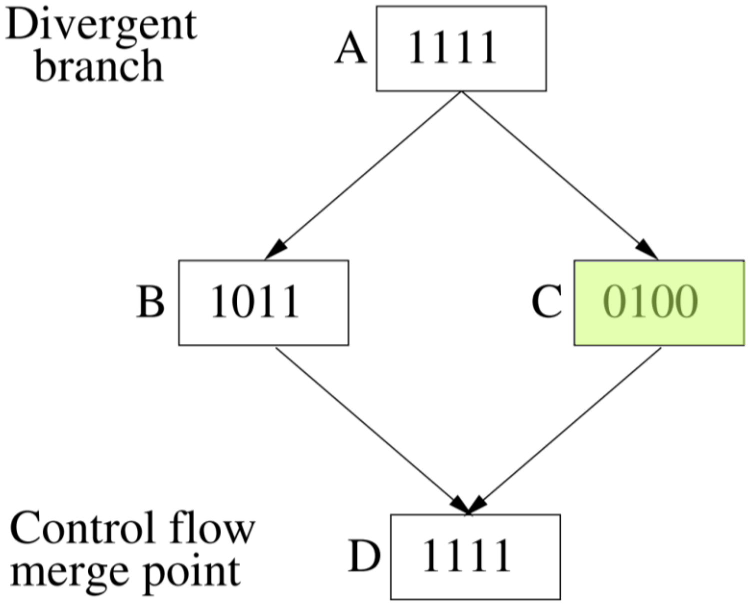


Current PC: A
Current Active Mask: 1111

Reconvergence PC	Active Mask	Execute PC
D	0100	C
D	1111	D

Conditional Branch Handling

2. Warp’s next PC is compared to the re-convergence PC at the top of the stack
   - If equal – stack is popped, active mask & PC updated



Current PC: C
Current Active Mask: 0100

Reconvergence PC	Active Mask	Execute PC
D	1111	D

Mechanism 1 - Large Warp Microarchitecture (LWM)

- Proposed solution to branch divergence
  → ***large warp microarchitecture***
  - Fewer, but larger warps
  - Total number of threads and SIMD width stay the same
  - Benefit: fully populated sub-warps can be formed dynamically from active threads

- Implemented using an ***active mask***
  - Each cell = single bit
  - # of columns = SIMD width
  - Storage cost stays the same

Large warp width = SIMD width = N

Large warp depth = K

Row 0	1	1	0	0
Row 1	1	0	0	1
Row 2	0	1	1	1
Row 3	0	0	1	0
	. . .	. . .	. . .	. . .
Row k-1	1	0	1	0

Large warp active mask

LWM: Sub-warp creation

- Goal: pack as many warps as possible into a sub-warp

- Implementation: specialized sub-warping logic
  - Examines the two-dimensional active mask of the large warp
  - Attempts to pick one active thread from each column

Large warp width = SIMD width = N

Large warp depth = K

Row 0	1	1	0	0
Row 1	1	0	0	1
Row 2	0	1	1	1
Row 3	0	0	1	0
	. . .	. . .	. . .	. . .
Row k-1	1	0	1	0

Large warp active mask

LWM: Sub-warp creation

After Fetch

Row 0	1	1	0	0
Row 1	1	0	0	1
Row 2	0	1	1	1
Row 3	0	0	1	0
Row 4	0	0	1	1
Row 5	0	1	0	1
Row 6	1	0	0	0
Row 7	1	0	1	0

Cycle x+1

1	**1**	0	0
1	0	0	**1**
0	1	**1**	1
0	0	1	0
0	0	[illegible]	[illegible]
0	1	0	1
1	0	0	0
1	0	1	0

Subwarp

Active Mask

1	1	1	1

Row ID

0	0	2	1

Cycle x+2

0	0	0	0
1	0	0	0
0	**1**	0	**1**
0	0	**1**	0
[illegible]	0	1	1
0	1	0	1
1	0	0	0
1	0	1	0

Active Mask

1	1	1	1

Row ID

1	2	3	2

Cycle x+3

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	**1**	**1**
0	**1**	0	1
1	0	0	0
1	0	1	0

Active Mask

1	1	1	1

Row ID

6	5	4	4

Cycle x+4

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	**1**
0	0	0	0
1	0	**1**	0

Active Mask

1	0	1	1

Row ID

7	-	7	5

How many cycles for baseline GPU? **8**

How many cycles for GPU with LWM? **4**

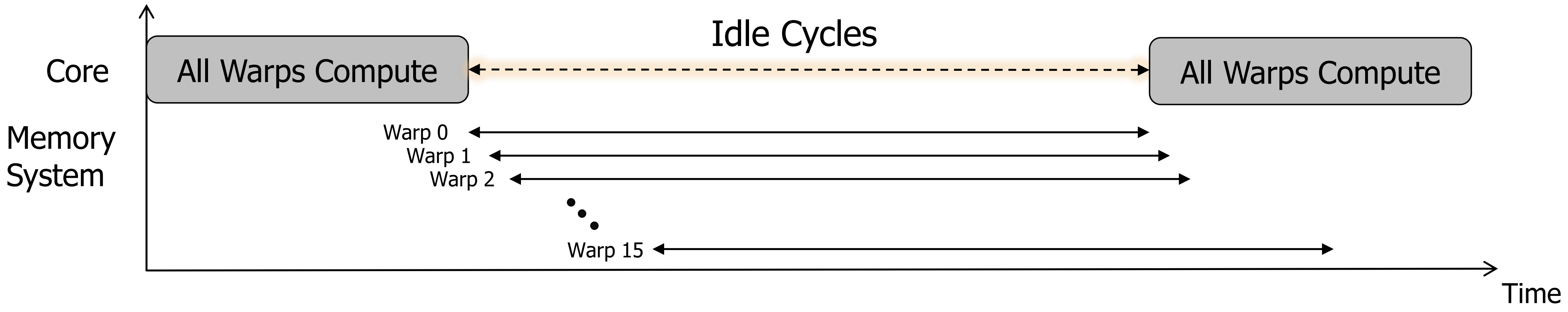
Large Warp Microarchitecture Adjustments

- Divergence stack
  - Handled at the large warp level
- Re-fetch policy
  - Wait for first sub-warp to finish
  - Re-fetch policy for conditional branches
    - Wait for last sub-warp
- Unconditional branch instructions
  - Execute in only one subwarp in a single cycle

Mechanism 2 – Two-Level Warp Scheduling

Problem 2 – long latency instructions

- Solution: **Two-Level Warp Scheduling**



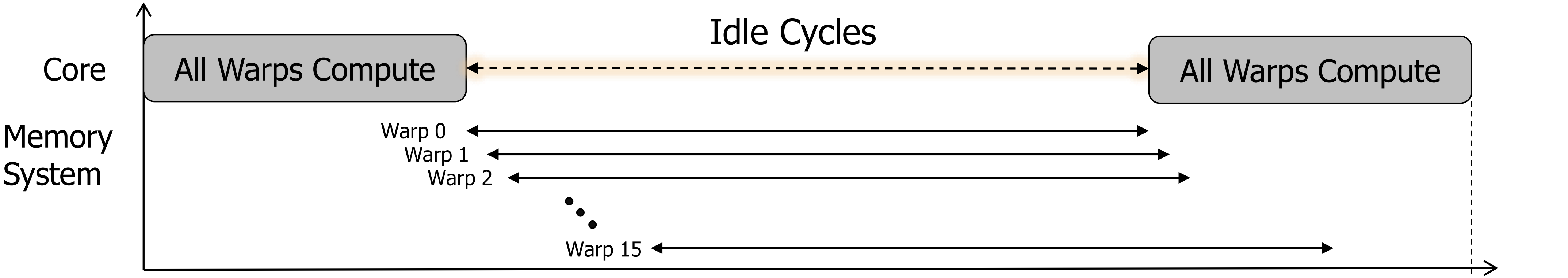
Baseline GPU round-robin scheduling with 16 warps

Two Level Warp Scheduling

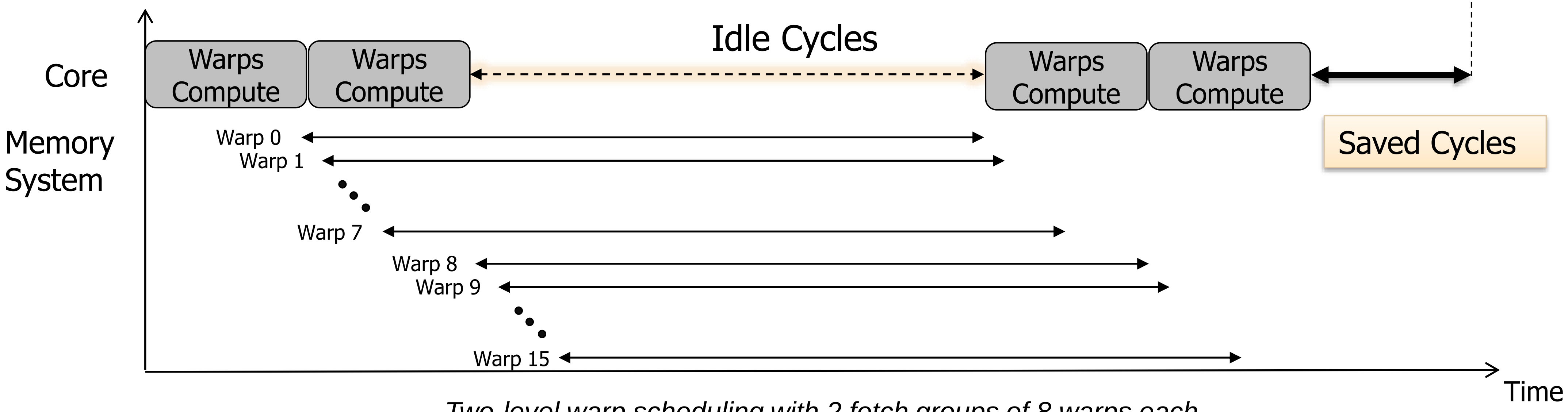
- All concurrently executing warps are grouped into fixed-sized *fetch groups*
  - For example, 32 warps into 4 fetch groups with 8 warps each and IDs 0-3
- Scheduling policy selects a single fetch group to prioritize
- Warps within the same fetch group
  - Have equal priority
  - Are scheduled in round-robin fashion
- Fetch group prioritization switch
  - Only once *all* warps in prioritized group are stalled
  - Also round robin

- Row buffer locality remains high due to two levels of round-robin

Two Level Warp Scheduling – Comparison



Baseline GPU round-robin scheduling with 16 warps



Two-level warp scheduling with 2 fetch groups of 8 warps each

Two-Level Scheduling – Fetch Groups

- Ideal fetch group size:
  - Enough warps to keep pipeline busy in absence of long latency operations
  - Too small
    - Uneven progression
    - Destroys data locality among warps
  - Too large
    - Takes longer to reach stalling point, limiting effectiveness of TLS
    - Large fetch group = greater number of warps stalling = less warps to hide latency

LWM & Two-Level Scheduling

- Best partitioning
  - LWS: 4 large warps, 256 threads each
  - TLS fetch group size: 1 large warp = 256 threads

- Applications with few long latency stalls cause issues
  - Lack of stalls leads to few fetch group changes
  - Starvation for a single large warp
  - Bubbles in pipeline due to branch re-fetch policy for large warps

- Solution: Fetch group priority change after a timeout
  - 32k instruction timeout period
  - Prevents starvation

Key Results:
Methodology and Evaluation

Methology - GPU

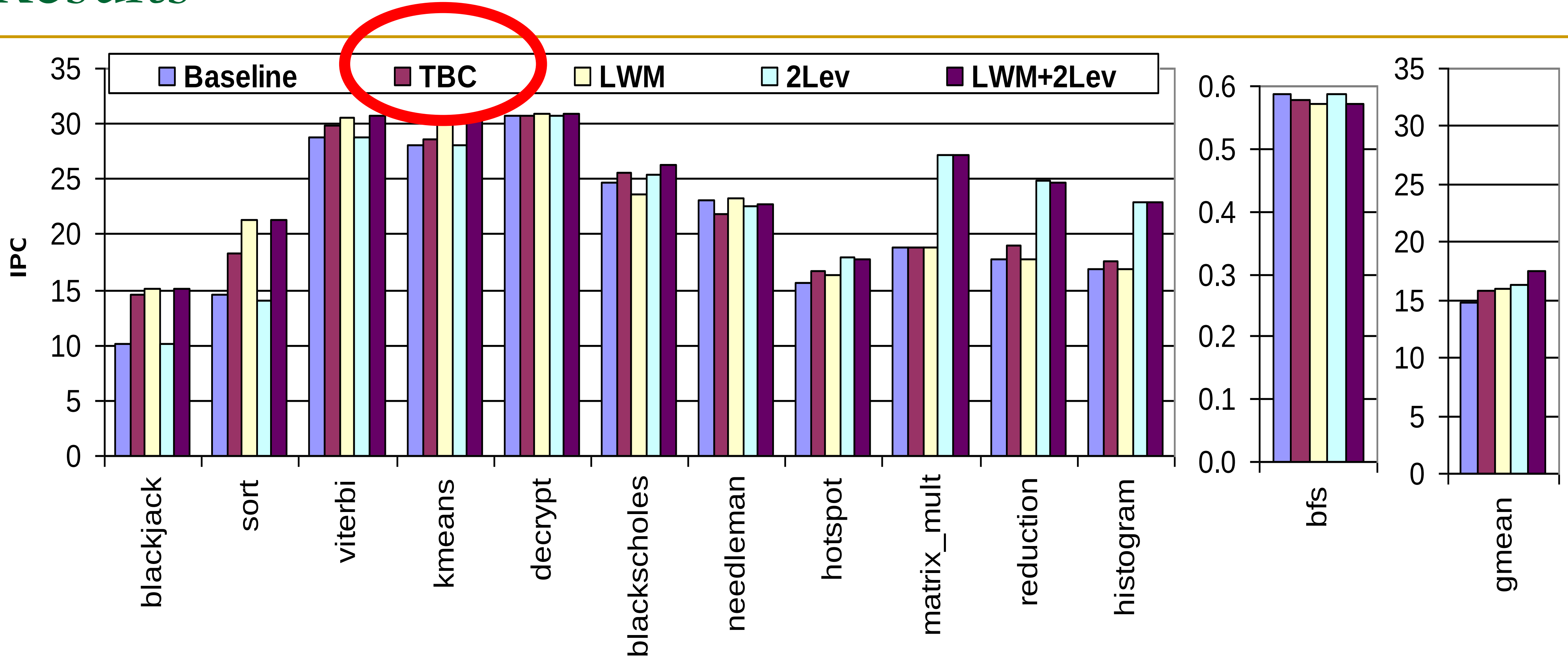
Simulate single GPU core with 1024 thread contexts divided into 32 warps each with 32 threads

Scalar Front End	1-wide fetch, decode 4KB single ported I-Cache Round-robin scheduling
SIMD Back End	In order, 5 stages, 32 parallel SIMD lanes
Register File and On Chip Memories	64KB Register File 128KB, 4-way, D-Cache with 128B line size 128KB, 32-banked private memory
Memory System	Open row, first-come first-serve scheduling 8 banks, 4KB row buffer per bank 100-cycle row hit latency, 300-cycle row conflict latency 32 GB/s memory bandwidth

Methodology - Benchmarks

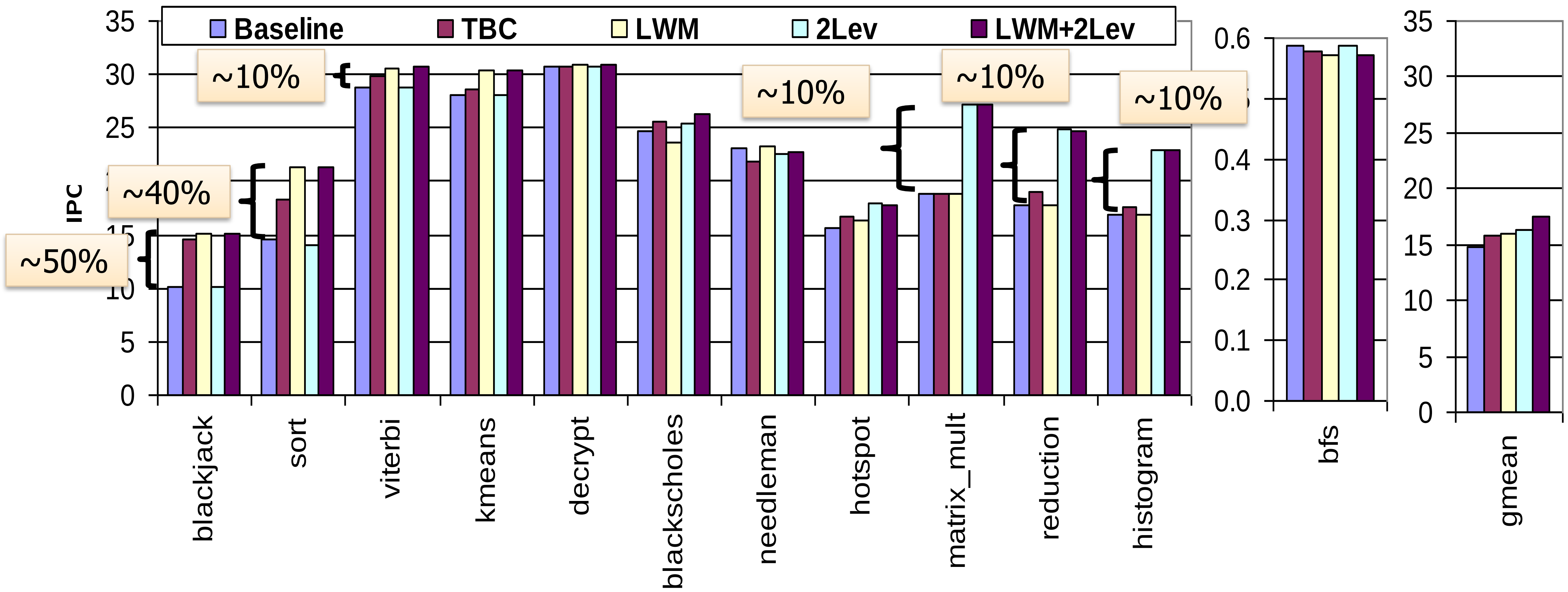
Benchmark	Description	Input Set
blackjack	Simulation of blackjack card game to compute house edge	Standard 52-card deck per thread
sort	Parallel bucket sort of a list of integers	1M random integers
viterbi	Viterbi algorithm for decoding convolutional codes	4M convolutionally encoded bits
kmeans	Partitioning based clustering algorithm	16K 1-dimensional 8-bit data points
decrypt	Advanced Encryption Standard decryption algorithm	256K AES encrypted bytes
blackscholes	Call/put option pricing using blackscholes equation	Initial values for 512K options
needleman	Calculate optimal alignment for 2 DNA sequences	2 DNA Sequences of length 2048
hotspot	Processor temperature simulation	512x512 grid of initial values
matrix_mult	Classic matrix multiplication kernel	2 256x256 integer matrices
reduction	Reduce input values into single sum	32M random boolean values
histogram	Binning ASCII characters into a histogram	16M ASCII characters
bfs	Breadth first search graph traversal	1 million node arbitrary graph

Results



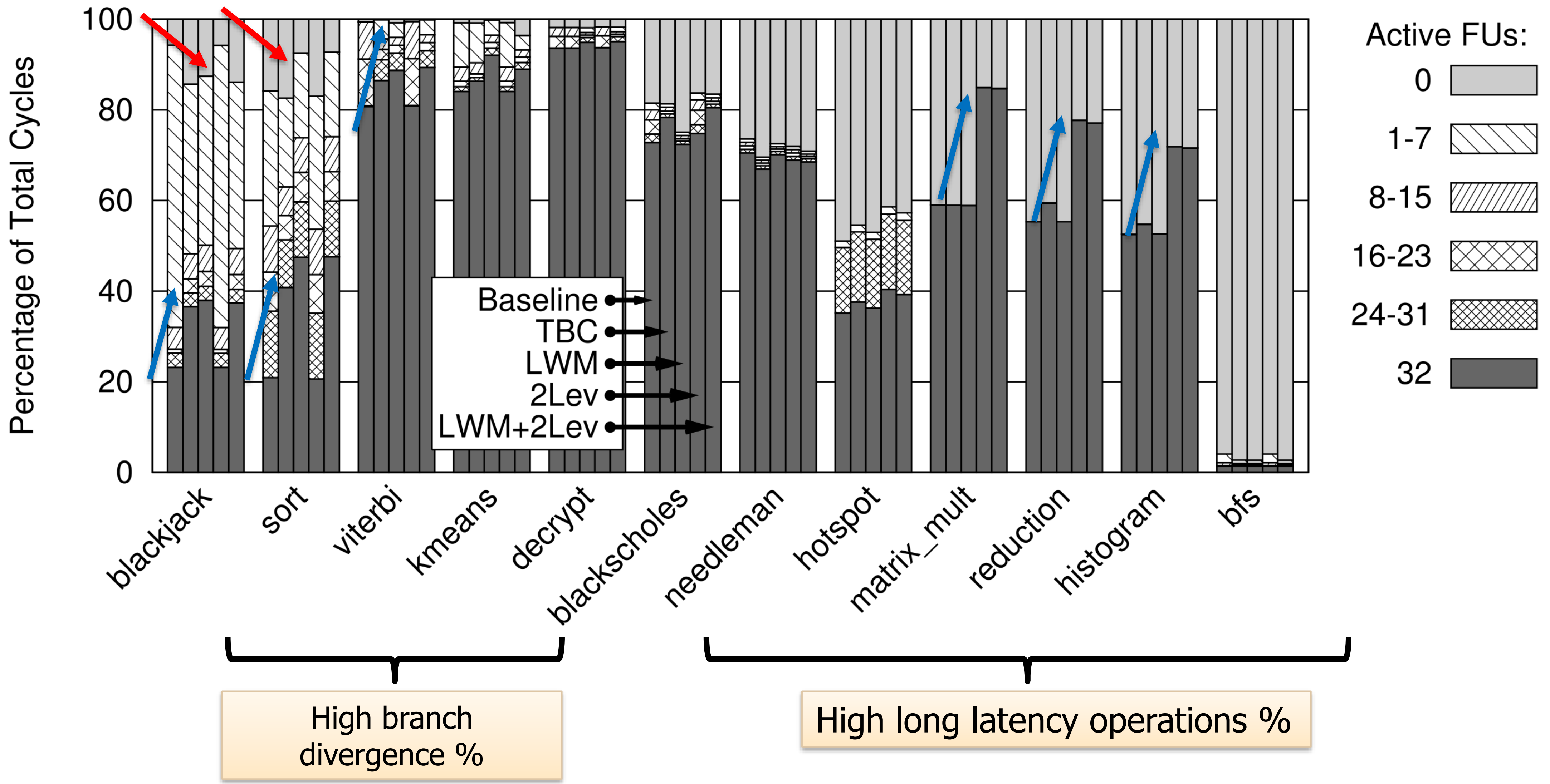
- TBC – Thread Block Compaction
  - Groups multiple regular-sized warps into a block and synchronizes them at every branch instruction

Results



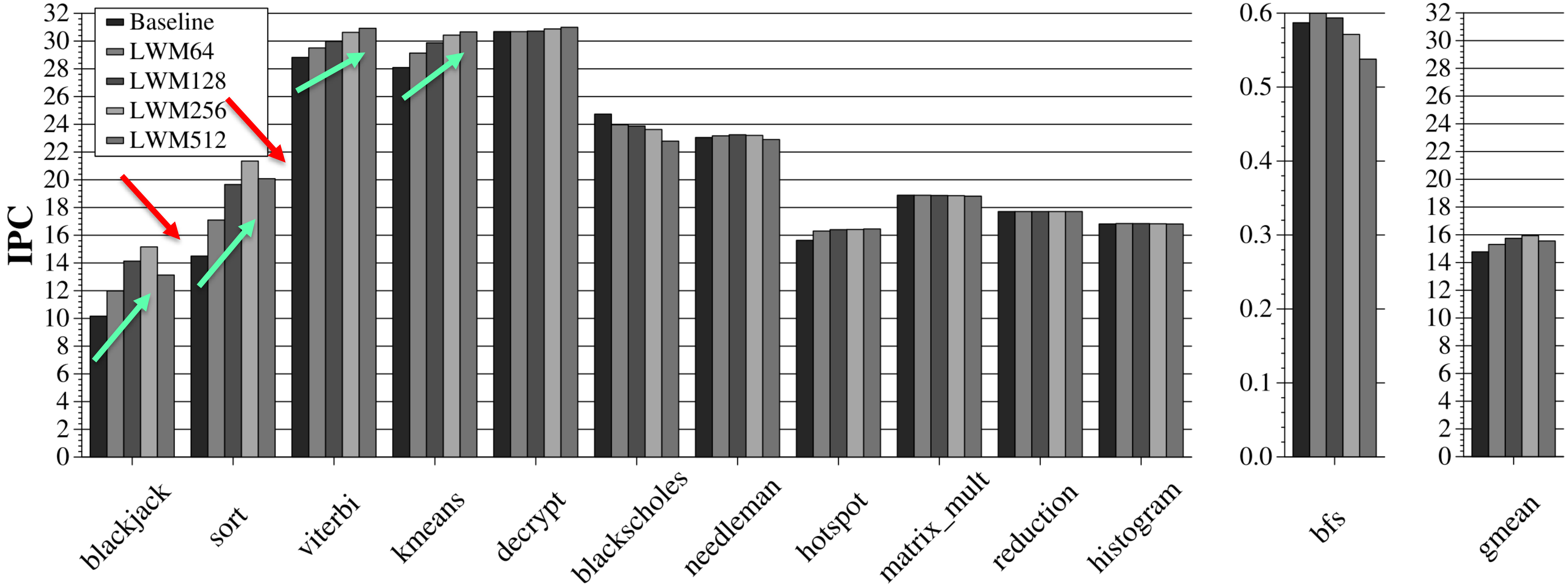
- LWM alone - 7.9% improvement
  - Good for branch-intensive applications
- 2Lev alone - 9.9% improvement
  - Good for long-latency stalls
  - Hit rate within 1.7% of traditional round robin

Results



Percentage of Total Cycles
100
80
60
40
20
0
Baseline
TBC
LWM
2Lev
LWM+2Lev
blackjack
sort
viterbi
kmeans
decrypt
blackscholes
needleman
hotspot
matrix_mult
reduction
histogram
bfs
Active FUs:
0
1-7
8-15
16-23
24-31
32
High branch divergence %
High long latency operations %

Large Warp Microachitecture Analysis

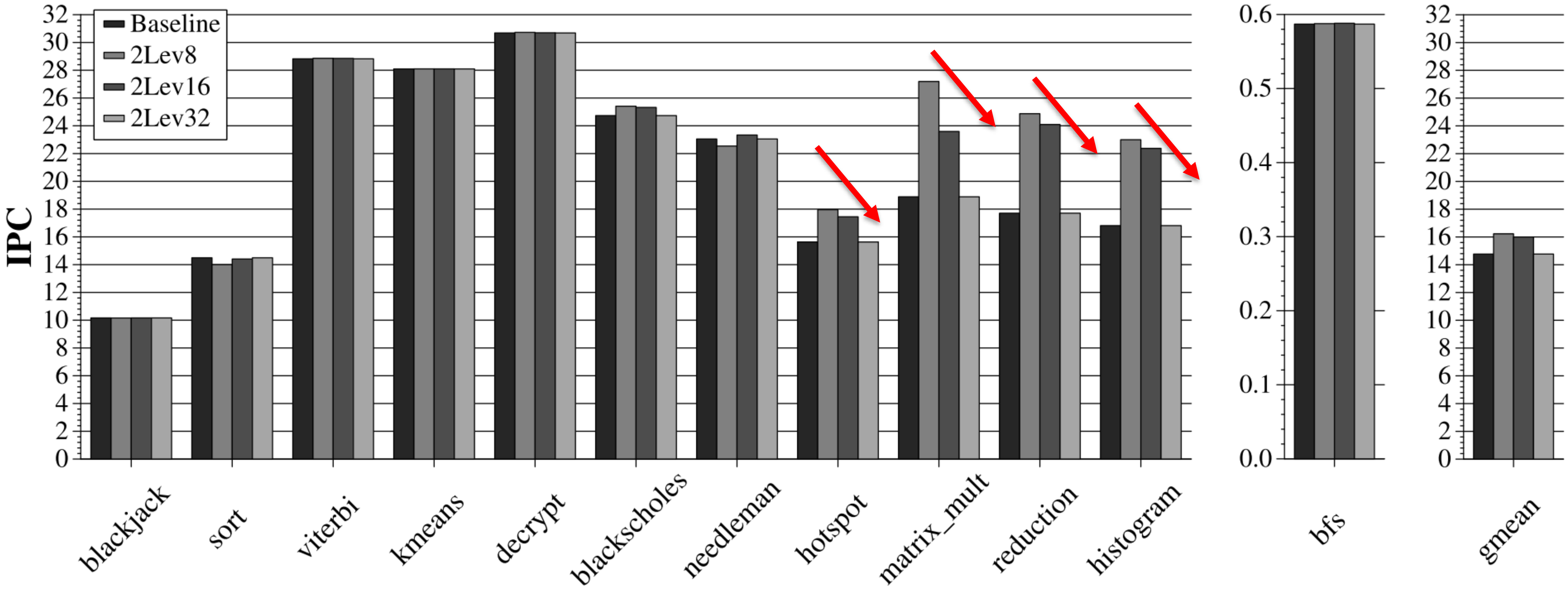


Effect of large warp size

- 256 threads per large warp best

- Larger warp sizes offer more potential for sub-warping
- Too large is inefficient when waiting for few divergent threads at re-convergence point

Two-Level Scheduling Analysis



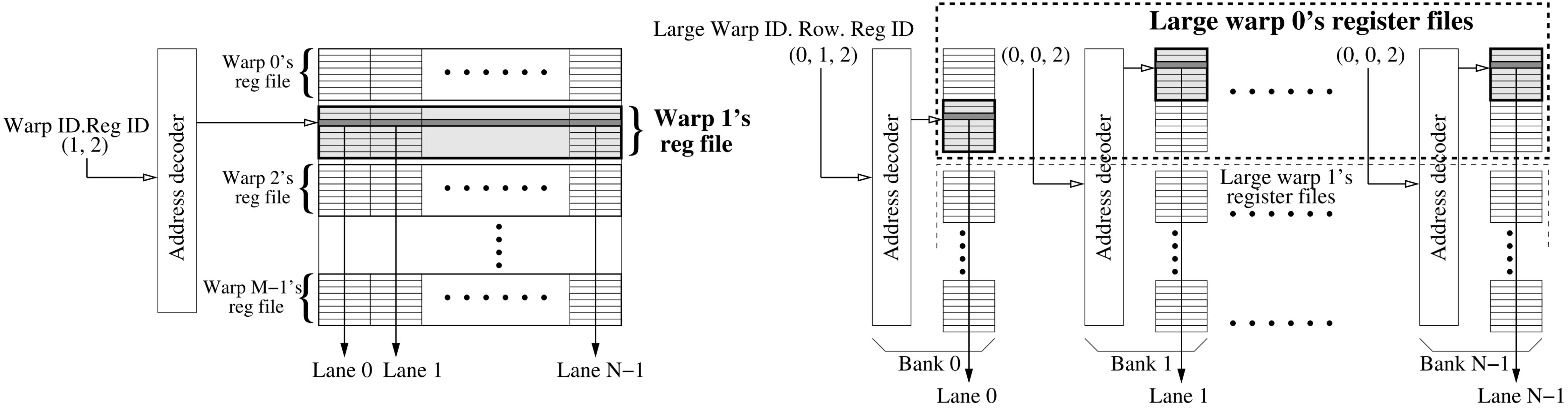
Effect of fetch group size on two-level scheduling

- 8 warps per fetch group optimal
- Larger sizes limits effectiveness

Hardware Overhead

- Restructuring of the register file for LWM
  - Instead of a single address decoder per GPU core, each SIMD lane requires one
  - Increase of 11% - 18% in register file area
  - About 2.5% increase of the total GPU area
  - Additional 224 bytes of storage for logic handling
- Two-level warp scheduling does not require any additional storage cost
  - Only a simple logic block is required for the additional level of round robin

Register File Design



(a) Baseline register files

(b) Large warp microarchitecture register files

Summary

Summary

- Two new mechanisms to improve GPU performance by better resource utilization

- **Large warp microarchitecture** alleviates the branch divergence penalty
  - Forming of fewer but larger warps
  - Dynamically created SIMD-width sized sub-warps from the active threads

- **Two-level round warp scheduling** policy to reduce idle execution cycles
  - Prevents warps from arriving at the same long latency operation at the same time

- These two mechanisms have experimentally shown an improvement in performance by **19.1%** on average

Strengths of the paper

- Simple ideas to maximize available resources
- Big GPU performance improvement
- Minimal hardware overhead
- Pure hardware mechanism requiring no programmer effort
- Paper easy to read and understand
- Cited 336 times [1]

[1] according to Google Scholar

Weaknesses

- Some hardware overhead still necessary

- More data and explanation about how data locality is preserved even with two levels of round robin would have been welcomed

Take-aways

- Still plenty of room for improvement even in classic hardware like GPU’s
- Even small adjustments can make big impact on overall performance

Additional Reading

- W. W. L. Fung et al. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. ACM TACO, 6(2):1–37, June 2009.
- U. Kapasi et al. Efficient conditional operations for data-parallel architectures. In MICRO-33, 2000.
- J. Meng et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In ISCA-37, 2010.
- N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on gpu performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU, 2010.*

Questions

Discussion starters

- Can you think of any potential issues with the proposed mechanisms?

Reference Locality

- Two level scheduler exploits inter-wavefront locality
  - Intra-wavefront accounts for more misses
- Round-robin nature of TLS technique causes the destruction of older wavefront's intra-wavefront locality

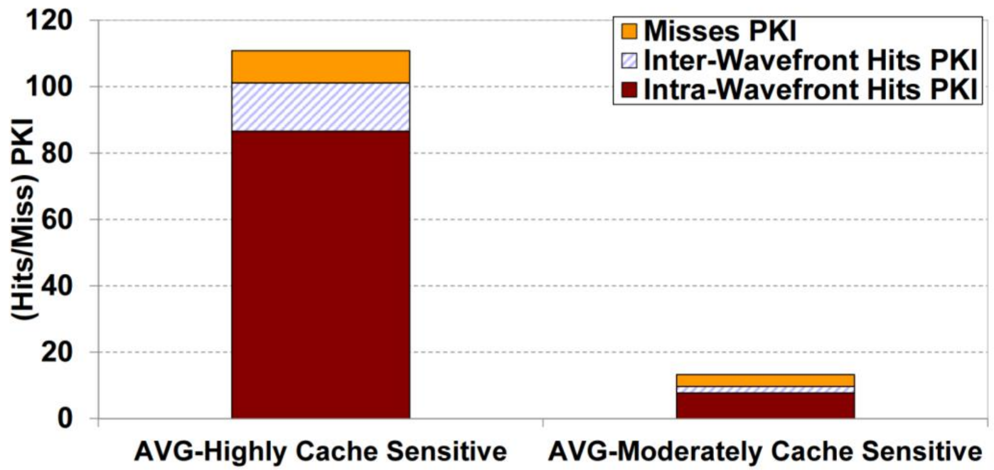
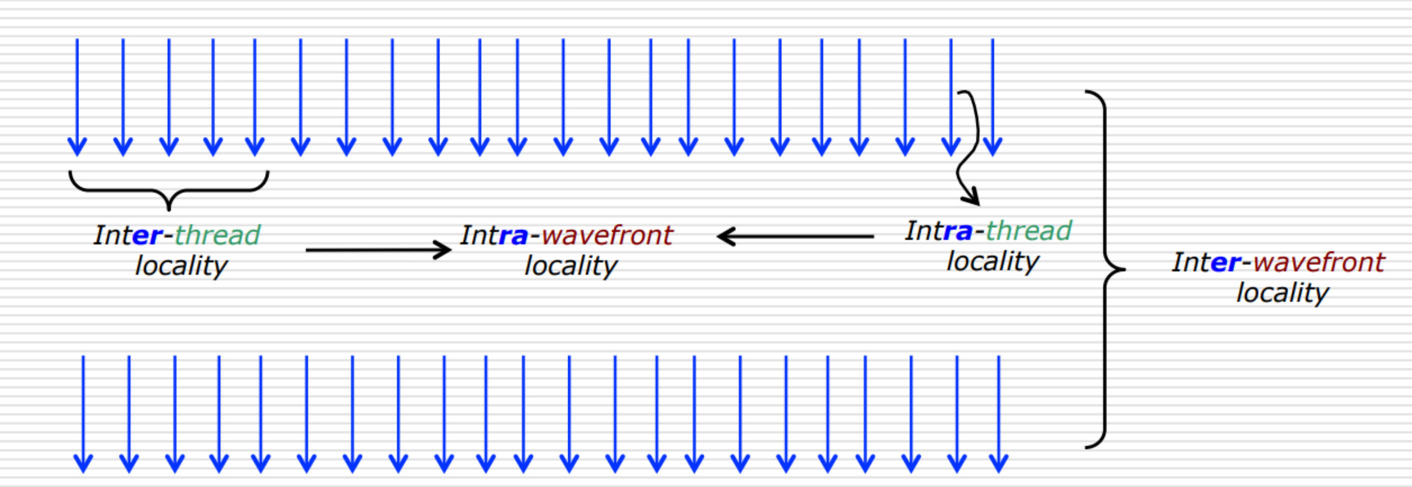


Figure 1: Average hits and misses *per thousand instructions* (PKI) using an unbounded L1 data cache (with 128B lines) on cache-sensitive benchmarks.



Figures from T. Rogers, M. O/Connor, T. Aamodt, "Cache Conscious Wavefront Scheduling," MICRO 2012

Discussion starters

- Can you think of any potential issues with the proposed mechanisms?
- Could you imagine any other alternative fetching policies?

Additional Mechanisms

- LRR – loose round robin scheduling
  - Wavefronts are prioritized for scheduling in round-robin order. However, if a wavefront cannot issue during its turn, the next wavefront in round-robin order is given the chance to issue.
- GTO – Greedy-then-oldest scheduler
  - GTO runs a single wavefront until it stalls then picks the oldest ready wavefront. The age of a wavefront is determined by the time it is assigned to the core.
- 2LVL – GTO
  - Combines Two Level Scheduling along with GTO
- CCWS – Cache-Conscious Wavefront Scheduling
  - Dynamically determines the number of wavefronts allowed to access the memory system and which wavefronts those should be
- SWL – Static Wavefront Limiting
  - Limits the number of wavefronts/ warps running at once to make sure it does not exceed L1D cache size

T. Rogers, M. O/Connor, T. Aamodt, “Cache Conscious Wavefront Scheduling,” MICRO 2012

Additional Mechanisms -Results

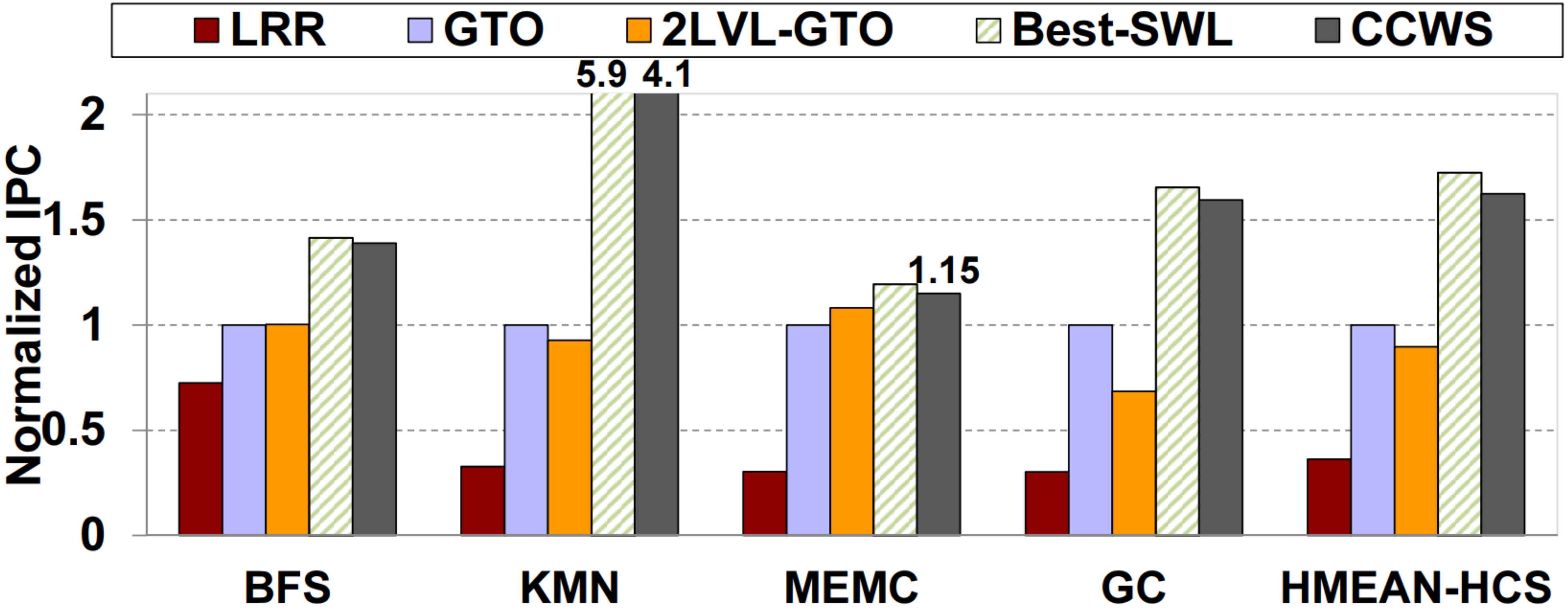


Figure 7: Performance of various schedulers and replacement policies for the highly cache-sensitive benchmarks. Normalized to the GTO scheduler.

Figures from T. Rogers, M. O/Connor, T. Aamodt, “Cache Conscious Wavefront Scheduling,” MICRO 2012

CCWS

- Dynamic tracking of relationship between wavefronts and working sets in the cache

- Modify scheduling decisions to minimize inference in the cache

T. Rogers, M. O/Connor, T. Aamodt, “Cache Conscious Wavefront Scheduling,” MICRO 2012

OWL - c(O)operative thread array a(W)are warp schedu(L)ing policy

- Takes advantage of characteristics of cooperative thread arrays (CTAs) to concurrently improve the GPGPU's
  - Cache hit rate
  - Latency hiding capability
  - DRAM bank parallelism
- Achieved by

  1) Selecting and prioritizing both L1 cache hit rates and latency tolerance

  2) Scheduling CTA groups thereby improving DRAM bank parallelism

  3) Employing opportunistic memory-side prefetching to take advantage of already open DRAM row
- OWL outperforms the proposed two-level scheduling policy by 19%

A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In ASPLOS, 2013.

Discussion starters

- Can you think of any potential issues with the proposed mechanisms?
- Could you imagine any other alternative fetching policies?
- Is IPC (instructions per cycle) the most important metric to judge GPUs by in today's world?
For example, wouldn't it be better to judge GPUs by performance per Watt?
- Could we implement these ideas in other pieces of hardware?
Or combine with other techniques and mechanisms?

Additional Slides

Memory Model

- Data from global memory is cached on chip in the data cache
- An entire cache line can be read (or written) in parallel in a single transaction

- If data accesses map to the same cache line …
  - all the threads in a warp access data in a <u>single transaction</u>
- If threads within a warp access different cache lines …
  - accesses will be serialized and pipeline <u>stalls</u>
- If a line not in the cache is required for at least one thread …
  - the warp stalls, is put aside, and other warps are allowed to continue

- Each thread has access to small amount of on-private memory
  - Stores private data of each thread like local variables
  - Cuts down on expensive global memory accesses

GPU Core Pipeline

- Fetch stage
  - scheduler selects a warp using a round-robin policy
- Each warp is associated with:
  - warp ID
  - active mask
    - a bit vector indicating whether a thread is active
  - a single program counter
- Instruction cache is accessed at the PC of the warp
- Fetched instruction is decoded
- Register values for all threads are read in parallel from the register file
- Values fed into SIMD backend and processed in parallel
- PC and active mask are updated after the final stage
- Warp is again considered for scheduling

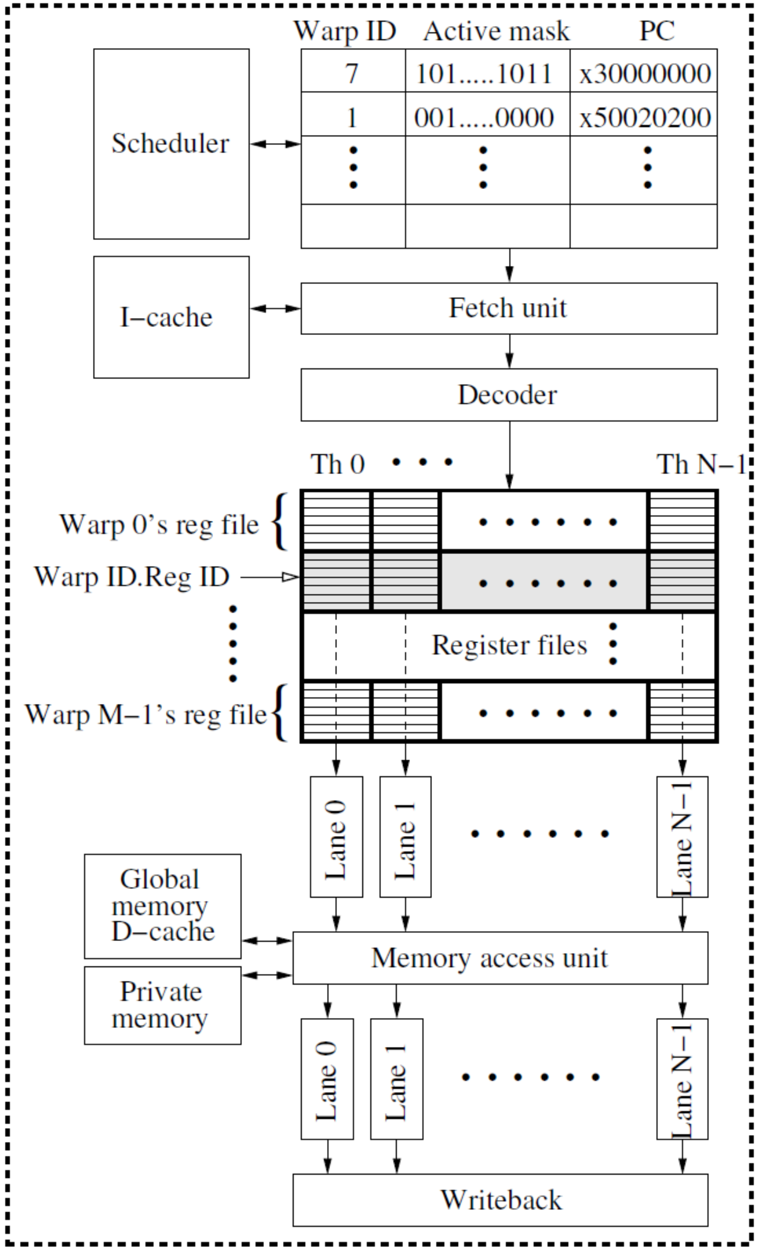
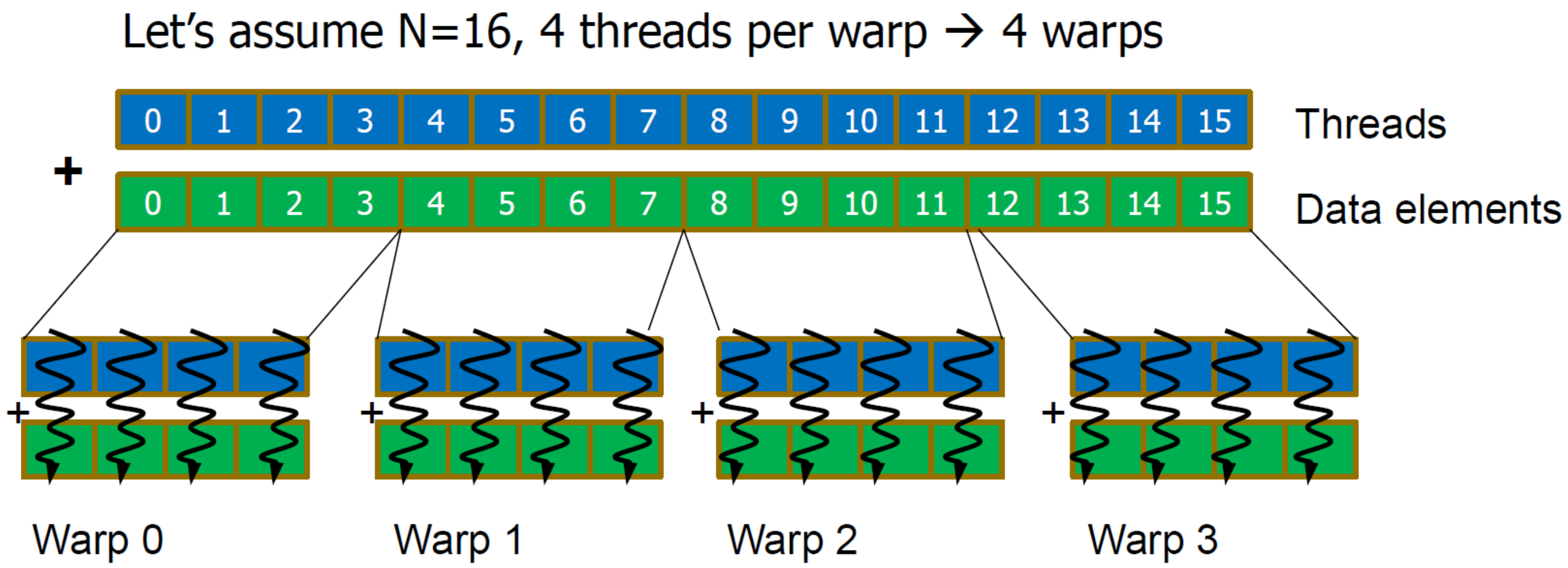


Figure 2: GPU core pipeline

SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements



Credit: Onur Mutlu – Design of Digital Circuits – ETH Spring 2017

LWM – Barrel Processing

- Standard GPU
  - Once a warp is selected by the scheduler in the fetch stage, it is not considered again for scheduling until the warp completes execution

- LWM Implementation – slight alteration
  - Once a large warp is selected, it is not reconsidered for scheduling until the first sub-warp completes execution
  - Single bit per thread to ensure thread not packed too soon
  - One exception: ***conditional branch instructions***
    - Large warp not refetched until *all* sub-warps are complete
    - Divergence not known until last sub-warp

LWM – Divergence and Reconvergence

- Divergence and re-convergence handled similarly to baseline warps.
- Except…
  - The new active mask and the active masks to be pushed on the divergence stack are buffered in temporary active mask buffers.
  - Once all subwarps complete execution, the current active mask and PC of the large warp are updated and divergence stack entries are pushed on the large warp's divergence stack (if large warp diverged)

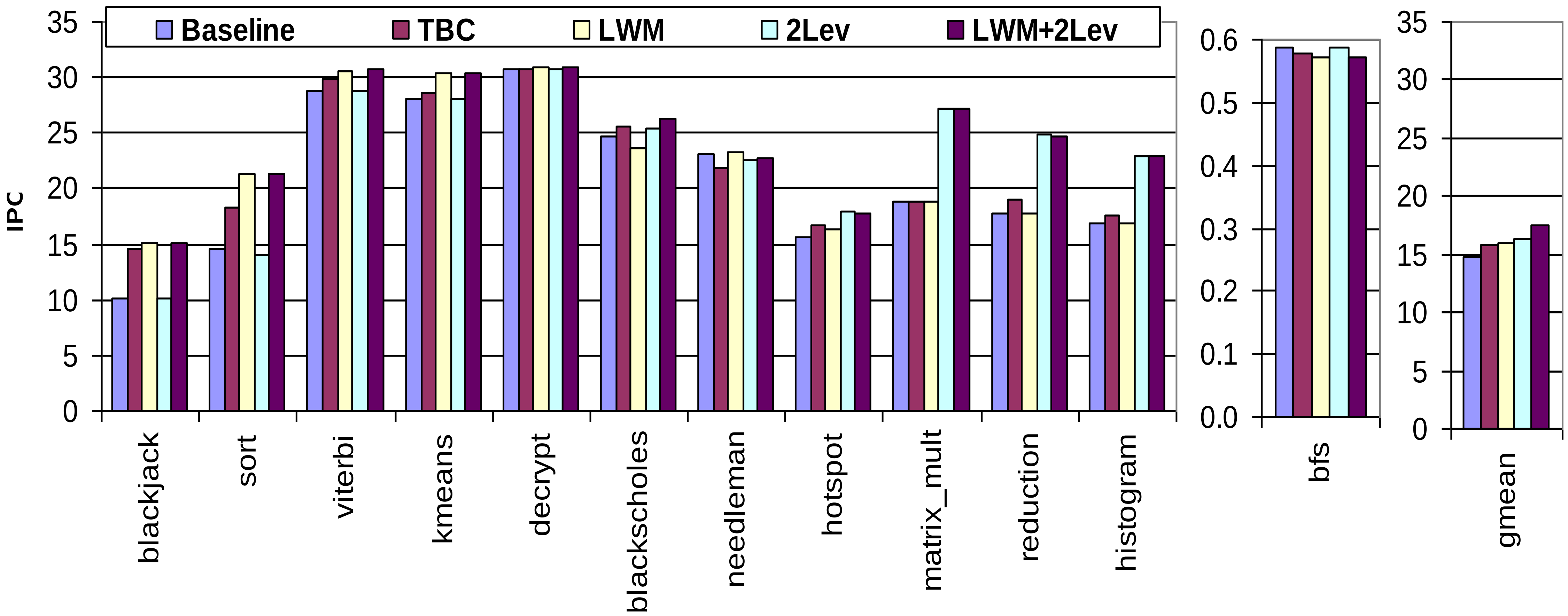
LWM – Unconditional Branch Optimization

- After an unconditional branch instruction (like a jump) is executed, only a single PC update is needed in standard GPU's

- Therefore, no multiple sub-warps are created as optimization

- Example
  - Warps with 256 threads with SIMD width of 32 can save 7 cycles with only one sub-warp (instead of previous 8) since only one cycle is needed

Results



- IPC – *Instructions per Cycle*
  - Warp size is the maximum possible value (ie 32)