OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance

Adwait Jog, Onur Kayiran- The Pennsylvania State University

Nachiappan Chidambaram Nachiappan - The Pennsylvania State University

Asit K. Mishra, Ravishankar Iyer – Intel Labs

Onur Mutlu – Carnegie Mellon University

Mahmut T. Lamdemir, Chita R. Das - The Pennsylvania State University

Overview

Introduction

Background

Challenges

Contribution

Summary

Current application agnostic GPU memory system drawbacks

- sub-optimal utilization of GPU performance
- significant imbalance in performance slowdowns across kernels
- Inefficiency of long current Warp scheduling policies in tolerating Memory latencies

Ways to solve this

- CTA aware scheduling Policy with minimal hardware changes
- Four different schemes which make a 33% average improvement in performance

Overview

Introduction

Background

Challenges

Contribution

Summary

Causes for GPU Idle time

- If CTA needs more Resources , then number of CTAs per core reduces, making inefficient utilization

- When threads in a Warp take different control flow paths ,the number of threads that can continue execution in Parallel Reduces.

- Low Efficiency of commonly used RR policy to hide Long memory fetch Latencies, caused due to limited off chip DRAM

- This poses a problem when we try to use Unified memory architecture for CPU and GPU

OWL

- A new Scheme called OWL – Co(O)perative Thread arrway aware (W)arp Schedu(L)ing Policy

- This is a proposed four-pronged concerted approach

CTA-aware two-level warp scheduler

- N CTAs per Core are grouped into n CTAs
- scheduling these small groups in RR fashion

Advantages

-Smaller groups of Warps, in RR reduces L1 chache contention
-Improves latency hiding and reduces inactive periods as not all warps read long latency operations around the same time
- 8% improved chache hit, 11% improved IPC performance

Locality aware warp scheduler

- This is a extension to two-level warp scheduler to reduce L1 cache contention further by always prioritizing a group of n CTAs over rest of the CTAs till the finish
- The major goal is to take advantage of locality between nearby threads and warps

Advantages

- 10% improved cache hit rate over previous scheme and resulting 11% improved IPC performance

Overview

Introduction

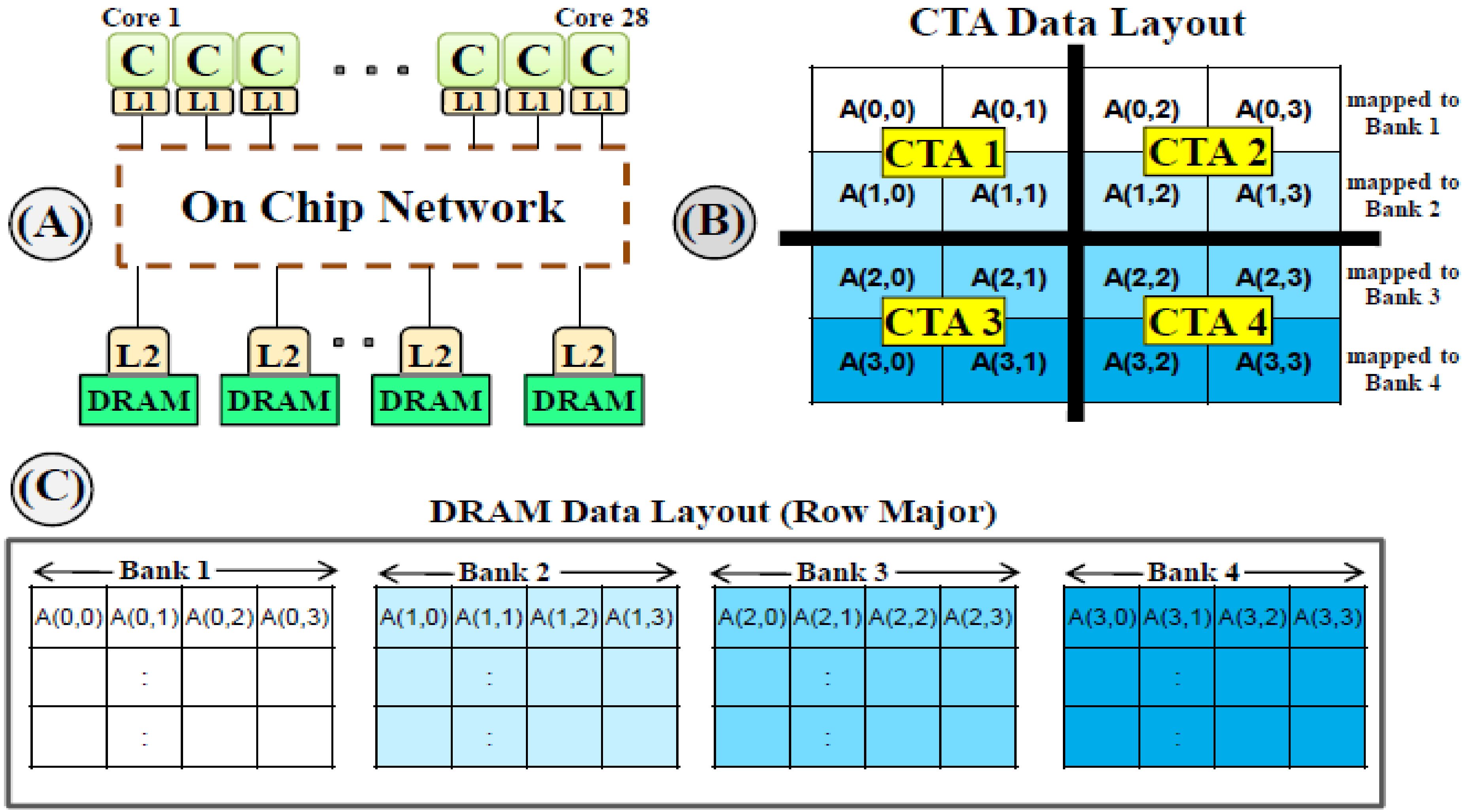
Background

Challenges

Contribution

Summary

Baseline GPU architecture



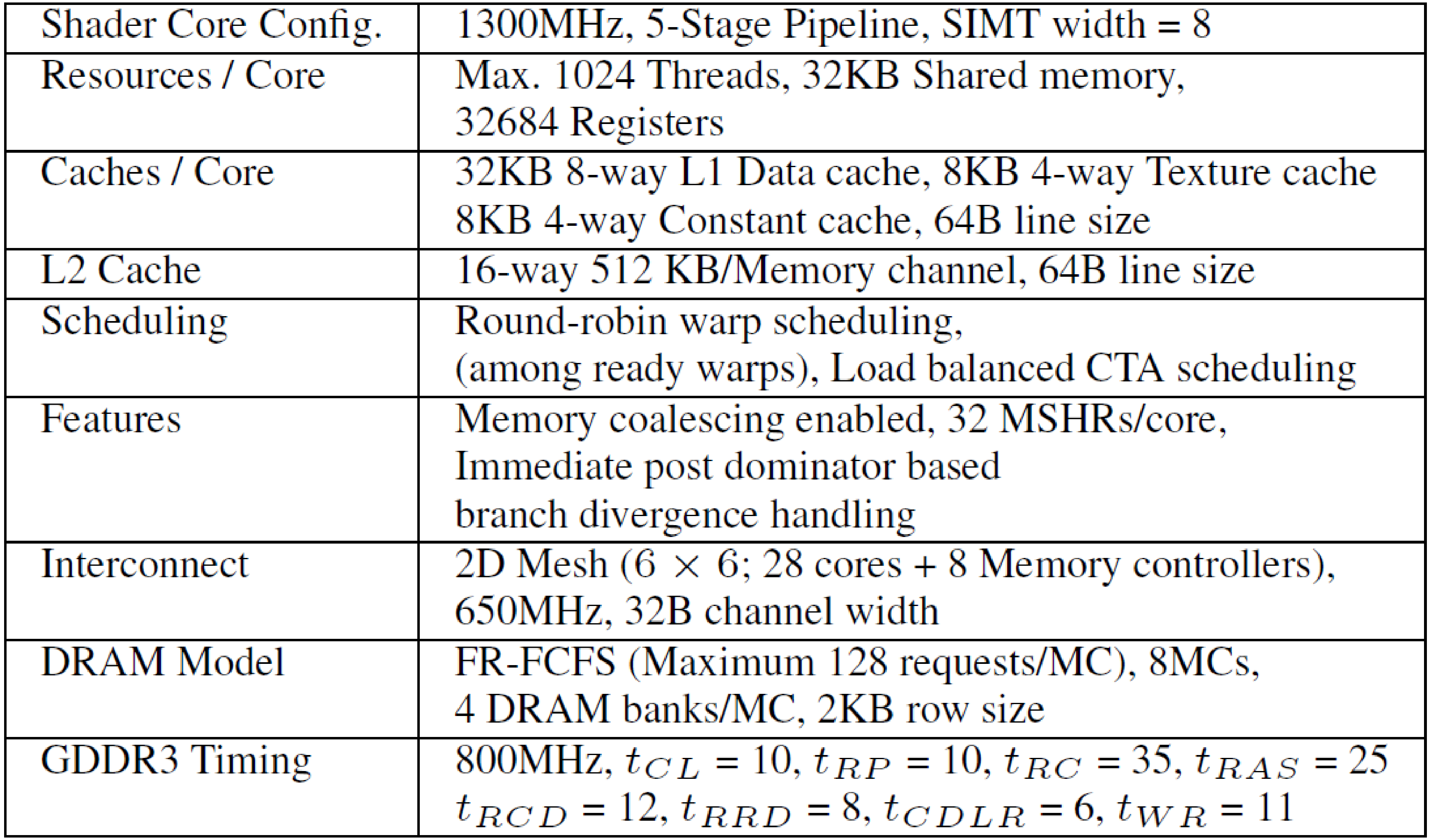
CTA Data Layout

A(0,0)	A(0,1)	A(0,2)	A(0,3)	mapped to Bank 1
A(1,0)	A(1,1)	A(1,2)	A(1,3)	mapped to Bank 2
A(2,0)	A(2,1)	A(2,2)	A(2,3)	mapped to Bank 3
A(3,0)	A(3,1)	A(3,2)	A(3,3)	mapped to Bank 4

DRAM Data Layout (Row Major)

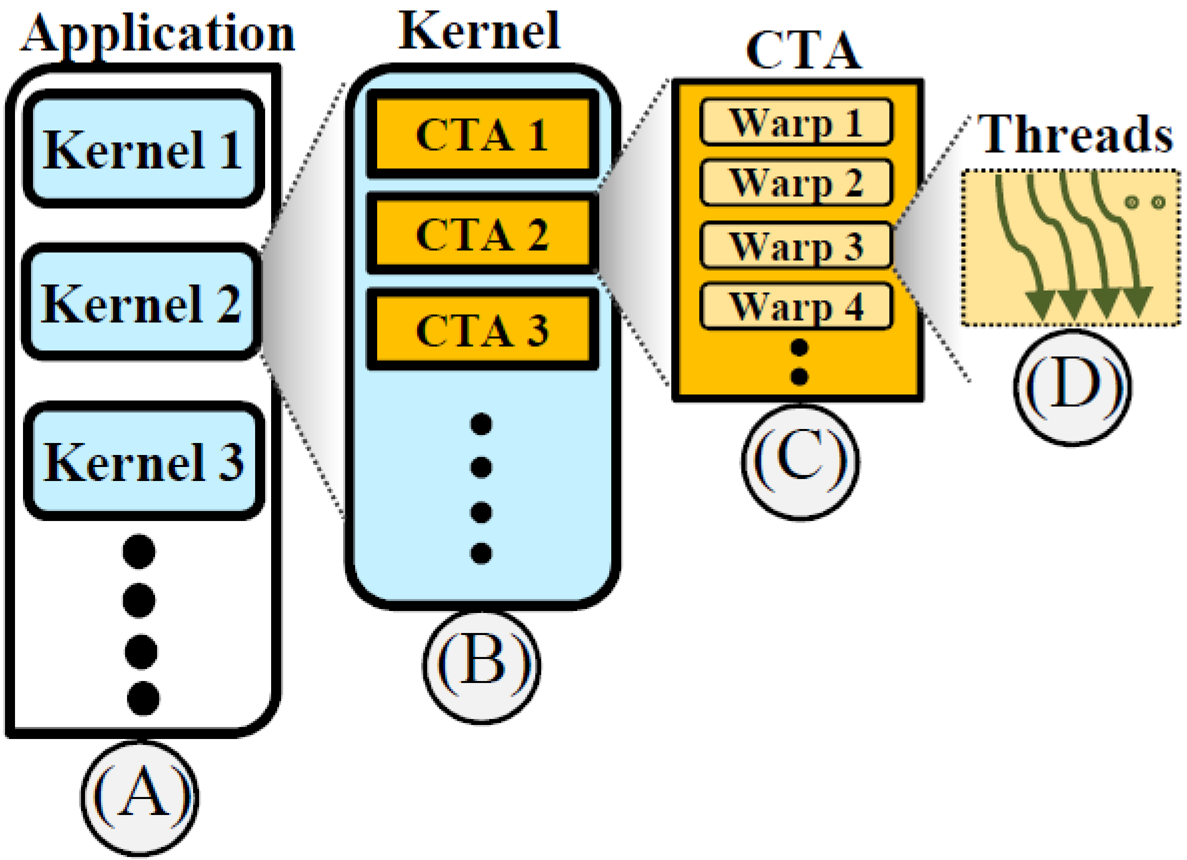
Bank 1				Bank 2				Bank 3				Bank 4			
A(0,0)	A(0,1)	A(0,2)	A(0,3)	A(1,0)	A(1,1)	A(1,2)	A(1,3)	A(2,0)	A(2,1)	A(2,2)	A(2,3)	A(3,0)	A(3,1)	A(3,2)	A(3,3)
	:				:				:				:		
	:				:				:				:		

Baseline Configuration



Shader Core Config.	1300MHz, 5-Stage Pipeline, SIMT width = 8
Resources / Core	Max. 1024 Threads, 32KB Shared memory, 32684 Registers
Caches / Core	32KB 8-way L1 Data cache, 8KB 4-way Texture cache 8KB 4-way Constant cache, 64B line size
L2 Cache	16-way 512 KB/Memory channel, 64B line size
Scheduling	Round-robin warp scheduling, (among ready warps), Load balanced CTA scheduling
Features	Memory coalescing enabled, 32 MSHRs/core, Immediate post dominator based branch divergence handling
Interconnect	2D Mesh (6 × 6; 28 cores + 8 Memory controllers), 650MHz, 32B channel width
DRAM Model	FR-FCFS (Maximum 128 requests/MC), 8MCs, 4 DRAM banks/MC, 2KB row size
GDDR3 Timing	800MHz, t_{CL} = 10, t_{RP} = 10, t_{RC} = 35, t_{RAS} = 25 t_{RCD} = 12, t_{RRD} = 8, t_{CDLR} = 6, t_{WR} = 11

GPU application Hierarchy



Overview

Introduction

Background

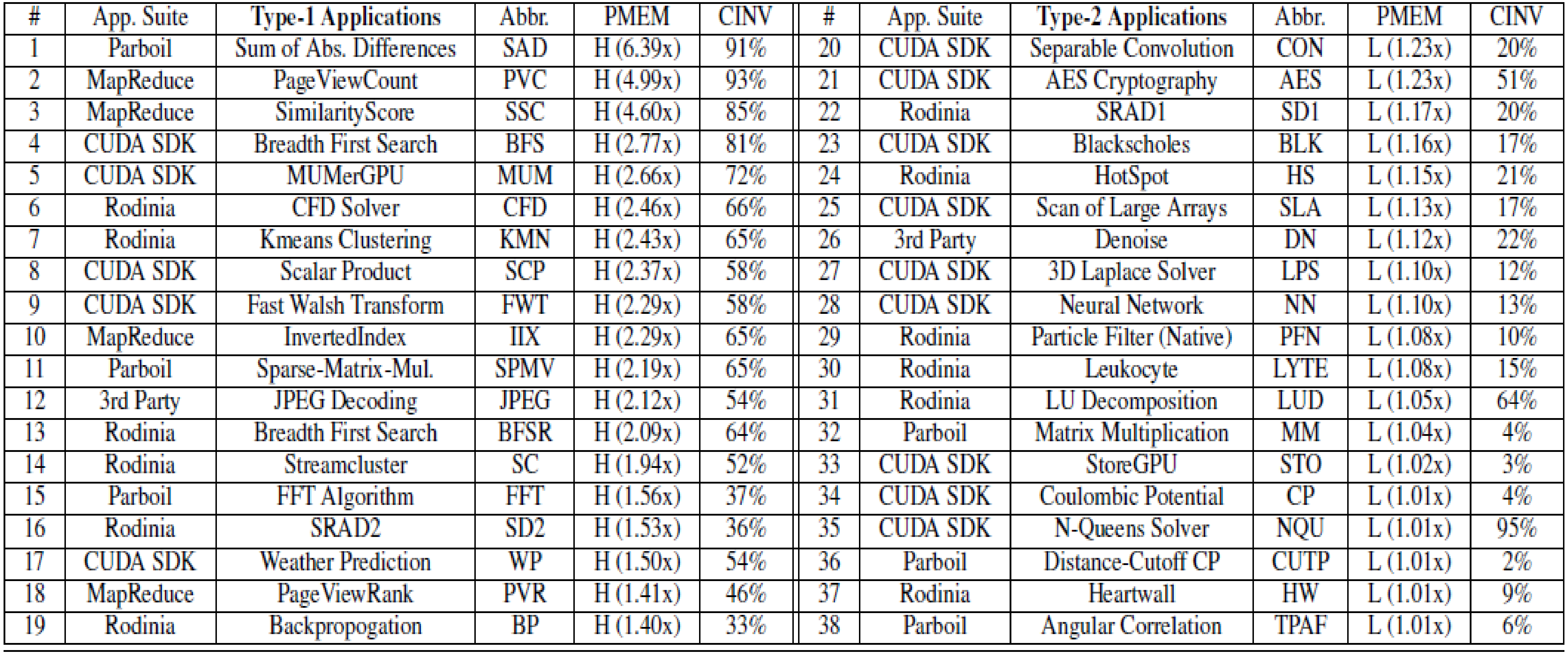
Challenges

Contribution

Summary

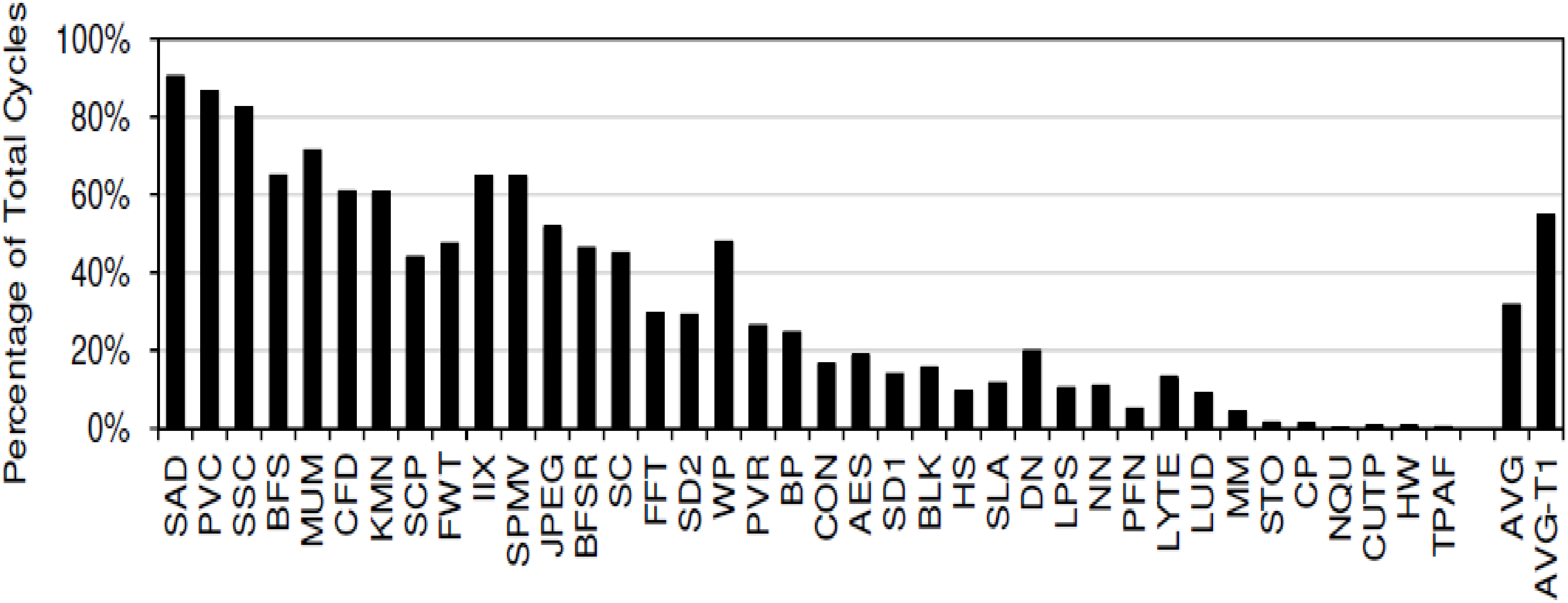
Motivation and Workload Analysis

- CINV (Core inactive Time)
- PMEM : all memory requests magically hit in L1 Cache



#	App. Suite	Type-1 Applications	Abbr.	PMEM	CINV	#	App. Suite	Type-2 Applications	Abbr.	PMEM	CINV
1	Parboil	Sum of Abs. Differences	SAD	H (6.39x)	91%	20	CUDA SDK	Separable Convolution	CON	L (1.23x)	20%
2	MapReduce	PageViewCount	PVC	H (4.99x)	93%	21	CUDA SDK	AES Cryptography	AES	L (1.23x)	51%
3	MapReduce	SimilarityScore	SSC	H (4.60x)	85%	22	Rodinia	SRAD1	SD1	L (1.17x)	20%
4	CUDA SDK	Breadth First Search	BFS	H (2.77x)	81%	23	CUDA SDK	Blackscholes	BLK	L (1.16x)	17%
5	CUDA SDK	MUMerGPU	MUM	H (2.66x)	72%	24	Rodinia	HotSpot	HS	L (1.15x)	21%
6	Rodinia	CFD Solver	CFD	H (2.46x)	66%	25	CUDA SDK	Scan of Large Arrays	SLA	L (1.13x)	17%
7	Rodinia	Kmeans Clustering	KMN	H (2.43x)	65%	26	3rd Party	Denoise	DN	L (1.12x)	22%
8	CUDA SDK	Scalar Product	SCP	H (2.37x)	58%	27	CUDA SDK	3D Laplace Solver	LPS	L (1.10x)	12%
9	CUDA SDK	Fast Walsh Transform	FWT	H (2.29x)	58%	28	CUDA SDK	Neural Network	NN	L (1.10x)	13%
10	MapReduce	InvertedIndex	IIX	H (2.29x)	65%	29	Rodinia	Particle Filter (Native)	PFN	L (1.08x)	10%
11	Parboil	Sparse-Matrix-Mul.	SPMV	H (2.19x)	65%	30	Rodinia	Leukocyte	LYTE	L (1.08x)	15%
12	3rd Party	JPEG Decoding	JPEG	H (2.12x)	54%	31	Rodinia	LU Decomposition	LUD	L (1.05x)	64%
13	Rodinia	Breadth First Search	BFSR	H (2.09x)	64%	32	Parboil	Matrix Multiplication	MM	L (1.04x)	4%
14	Rodinia	Streamcluster	SC	H (1.94x)	52%	33	CUDA SDK	StoreGPU	STO	L (1.02x)	3%
15	Parboil	FFT Algorithm	FFT	H (1.56x)	37%	34	CUDA SDK	Coulombic Potential	CP	L (1.01x)	4%
16	Rodinia	SRAD2	SD2	H (1.53x)	36%	35	CUDA SDK	N-Queens Solver	NQU	L (1.01x)	95%
17	CUDA SDK	Weather Prediction	WP	H (1.50x)	54%	36	Parboil	Distance-Cutoff CP	CUTP	L (1.01x)	2%
18	MapReduce	PageViewRank	PVR	H (1.41x)	46%	37	Rodinia	Heartwall	HW	L (1.01x)	9%
19	Rodinia	Backpropogation	BP	H (1.40x)	33%	38	Parboil	Angular Correlation	TPAF	L (1.01x)	6%

Percentage of total cycles spent waiting



Type 1: PMEM>=1.4x(Average CINV - 67%)
Type 2: PMEM<1.4x

- MemoryBlockCycles(70%): This indicates that even if there are Warps in the core , they are waiting for their memory requests to return back from L2.
- NoWarpCycles: This is idle time of core caused may be because of less CTAs in kernel or so called CTA load imbalance phenomenon, where some cores execute assigned CTAs earlier than others.

The Proposed OWL Scheduler

CTA-Aware: CTA-aware two-level warp scheduling

- We divide the whole N CTAs per core into (n) and (N-n)
- if k is the number of warps per CTA then (n*k) warps in 1st group, and ((N-n)*k) warps in another group.
- We need to make sure that minimum number of warps per group is satisfied(normally equal to the number of pipeline stages)
- Ex: if N=10, k=2 warps/CTA and (n*k) >=5(pipeline stages), we need to have n=3 in 1st group, similarly 3 in 2nd group, and if we choose 3 in 3rd group we cant satisfy minimum 5 warps for 4th group so we add n=4 for the 3rd group.

-

CTA-Aware: CTA-aware two-level warp scheduling(2)

- Once all the warps associated with the first selected group are blocked due to the unavailability of data, a group switch occurs giving opportunity to the next CTA group for execution

- We call this scheme CTA-aware two-level scheduling(*CTA-Aware*), as the groups are formed taking CTA boundaries into consideration and a two-level scheduling policy is employed, where scheduling within a group (level 1) and switching among different groups (level 2) are both done in a round-robin fashion.

Disadvantage : it takes decisions agnostic to inter-CTA row sharing properties

CTA-Aware-Locality: Locality aware warp scheduling

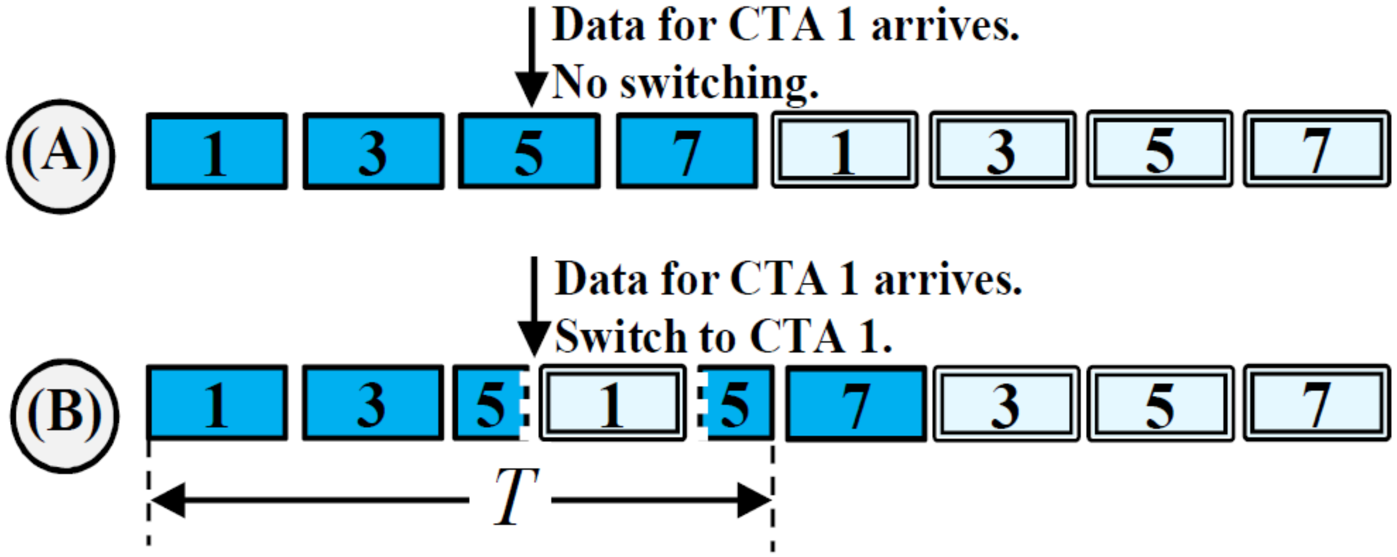
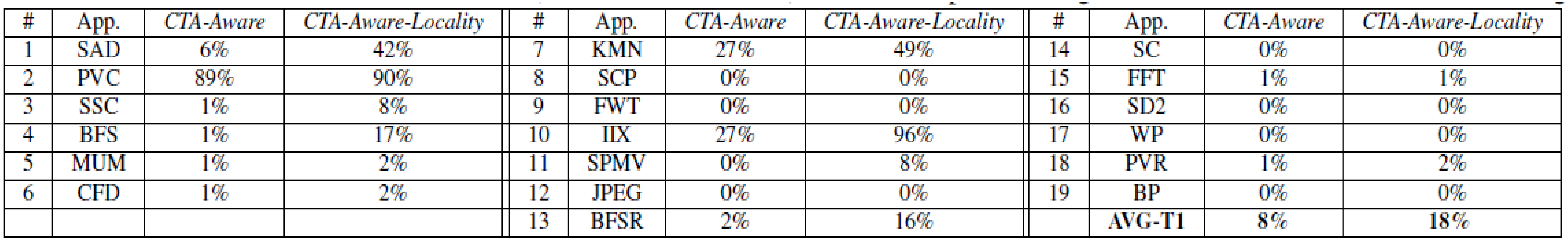


Figure 4. An illustrative example showing the working of (A) CTA-aware two-level warp scheduling (*CTA-Aware*) (B) Locality aware warp scheduling (*CTA-Aware-Locality*). Label in each box refers to the corresponding CTA number.

CTA-Aware-Locality: Locality aware warp scheduling

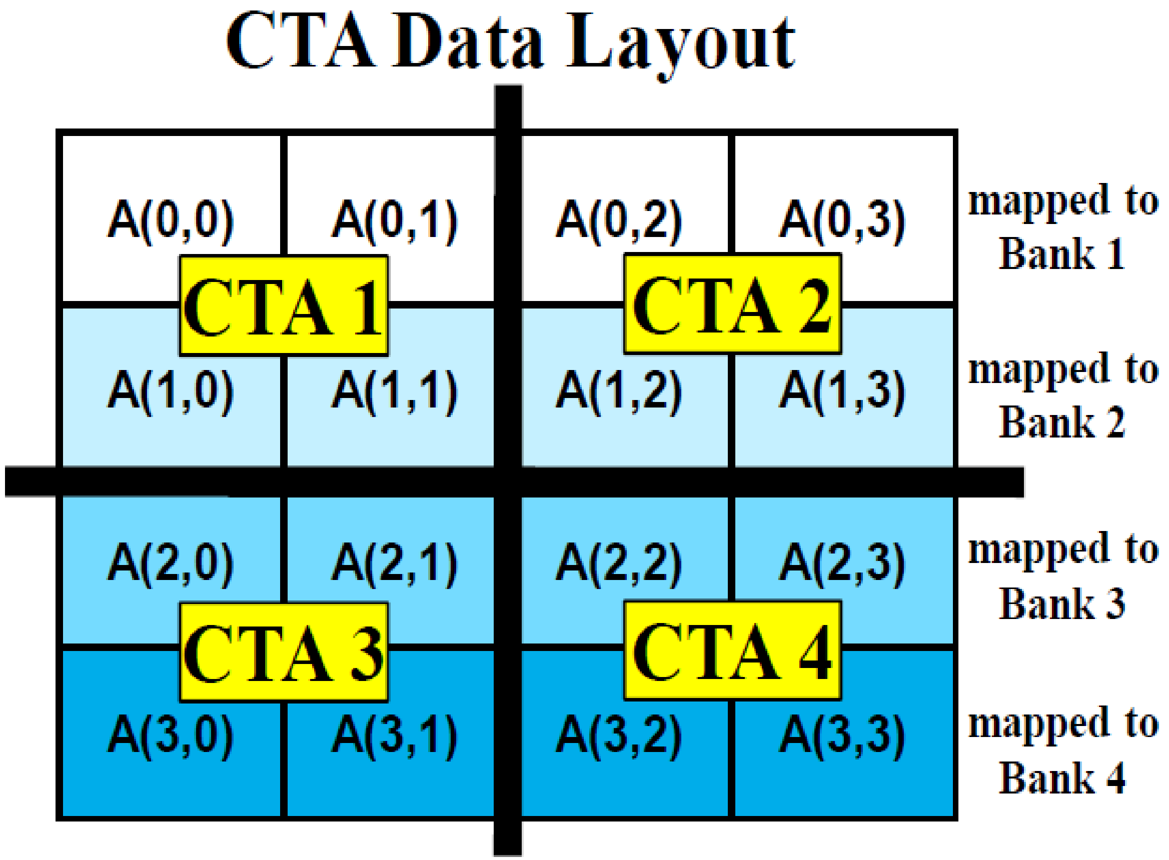
- Grouping takes place same as earlier two level warp scheduling, but the scheduling is not strictly round robin, if data appears on cache for a previously blocked CTA, that CTA is picked up and executed in order to avoid L1 misses.

-*CTA-Aware* reduces the overall miss rate by 8%. *CTA-Aware-Locality* is further able to reduce the overall miss rate (by 10%) by scheduling warps as soon as the data arrives for them, rather than waiting for their turn, thereby reducing the number of CTAs currently taking advantage of the L1 caches



#	App.	*CTA-Aware*	*CTA-Aware-Locality*
1	SAD	6%	42%
2	PVC	89%	90%
3	SSC	1%	8%
4	BFS	1%	17%
5	MUM	1%	2%
6	CFD	1%	2%
7	KMN	27%	49%
8	SCP	0%	0%
9	FWT	0%	0%
10	IIX	27%	96%
11	SPMV	0%	8%
12	JPEG	0%	0%
13	BFSR	2%	16%
14	SC	0%	0%
15	FFT	1%	1%
16	SD2	0%	0%
17	WP	0%	0%
18	PVR	1%	2%
19	BP	0%	0%
	AVG-T1	**8%**	**18%**

CTA-Aware-Locality-BLP: BLP aware warp scheduling



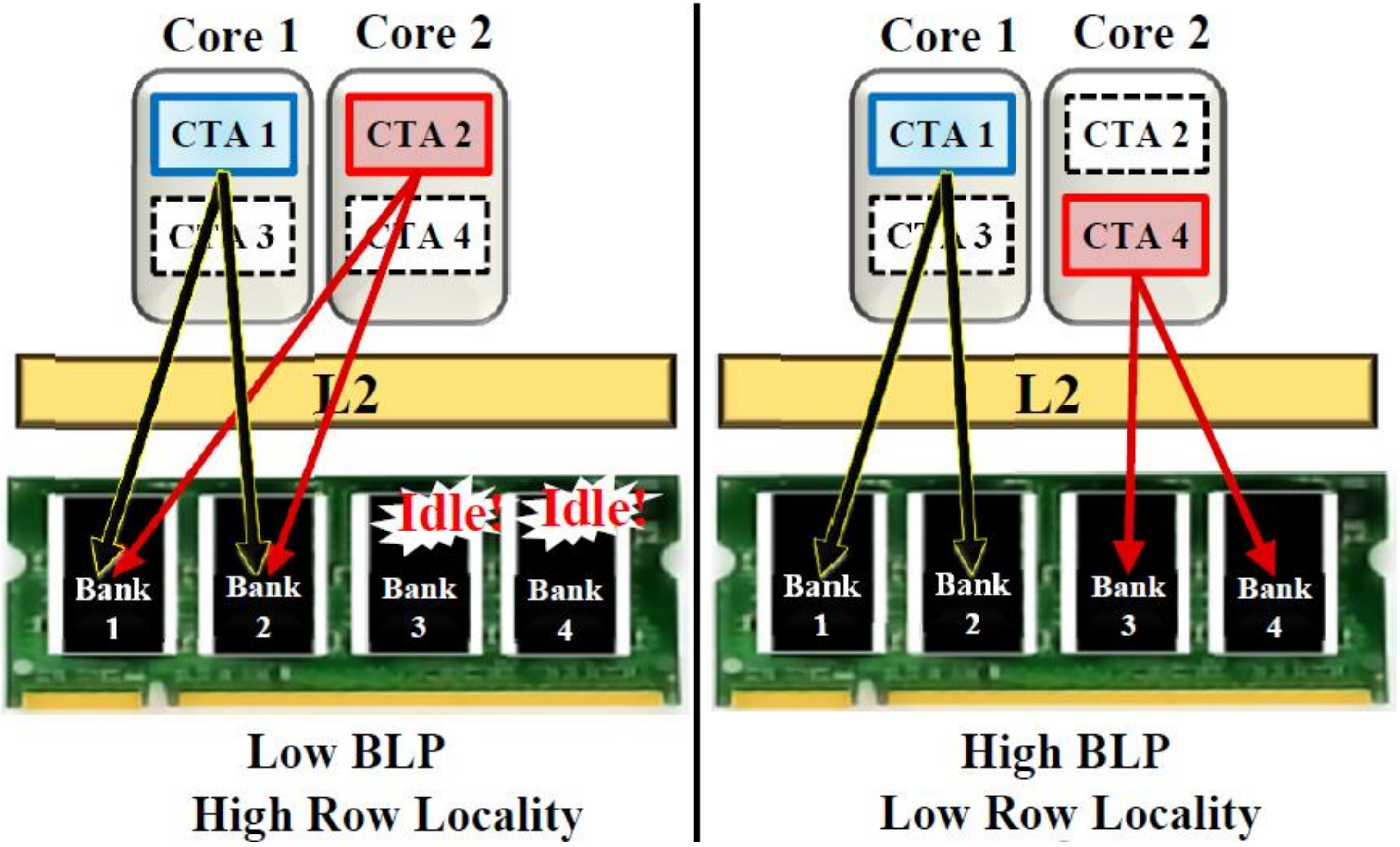
CTA Data Layout

A(0,0)	A(0,1)	A(0,2)	A(0,3)	mapped to Bank 1
A(1,0)	A(1,1)	A(1,2)	A(1,3)	mapped to Bank 2
A(2,0)	A(2,1)	A(2,2)	A(2,3)	mapped to Bank 3
A(3,0)	A(3,1)	A(3,2)	A(3,3)	mapped to Bank 4

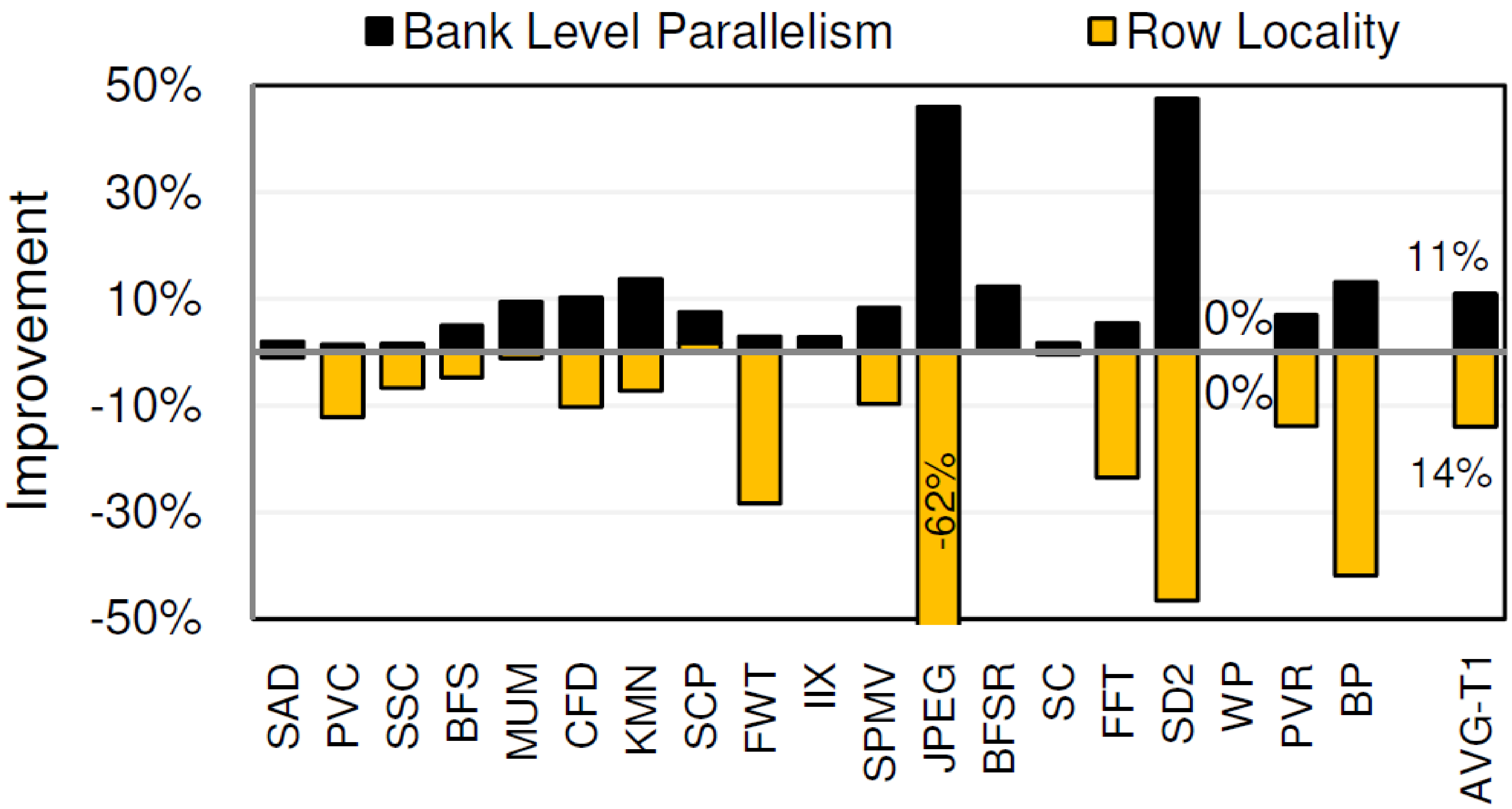
CTA-Aware-Locality-BLP: BLP aware warp scheduling

- Ensure that ***nonconsecutive CTAs*** (i.e., CTAs that do not share rows) are *always* prioritized in different cores

- Hence, prioritizing non-consecutive CTAs in different cores leads to improved BLP.

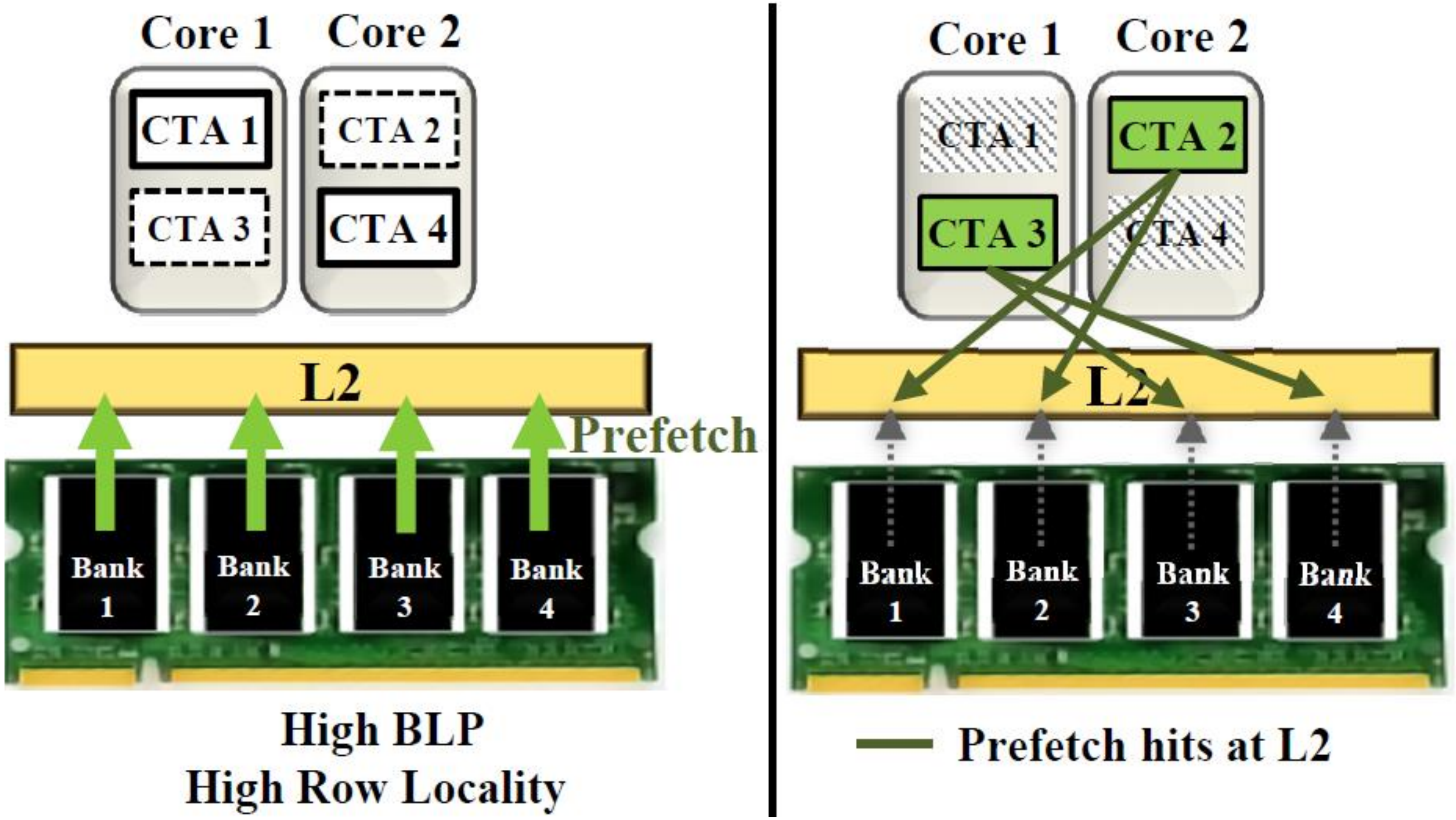


CTA-Aware-Locality-BLP: BLP aware warp scheduling



Opportunistic Prefetching

- Our goal is to restore row buffer locality (and hence efficiently utilize an open row as much as possible) while keeping the benefits of BLP



Opportunistic Prefetching

- We observe that prefetching cache blocks of an *already open row* can achieve this goal: if the prefetched cache blocks are later needed by other CTAs, these CTAs will find the prefetched data in the cache and hence do not need to access DRAM.
- The key idea of *opportunistic prefetching* is to prefetch the so-far-unfetched cache lines in an already open row into the L2 caches, just before the row is closed.
- There are two key design decisions in our opportunistic prefetcher: *what cache lines to prefetch* and *when to stop prefetching*.

Opportunistic Prefetching

- **What to prefetch?** The prefetcher we evaluate starts prefetching when there are no more demand requests to an open row.
- **When to stop opportunistic prefetching?**
  - In the first scheme, the prefetcher stops immediately after a demand request to a *different* row arrives.(demand imp. than prefetch)
- second scheme prefetches at least a minimum number of cache lines (*C*) regardless of whether or not a demand arrives. The value of *C* is set to a value *lower* initially. The prefetcher continuously monitors the number of demand requests at the memory controller queue. If that number is less than a *threshold*, the value of *C* is set to a value *higher*.(8 < *C* < 16)

Hardware Overheads

- CTA aware scheduling : 65nm TSMC , 28 core system has 57mW power overhead , 0.18mm^2

Optimistic prefetch

- 4MCs, each with 4 banks, 32 rows of Cache blocks (4*4*32 bits) has a overhead of 512bits

Overview

Introduction

Background

Challenges

Contribution

Summary

Experimental Results

-On average (arithmetic mean), ***CTA-Aware*** provides 14% IPC improvement, with 9% reduction in memory waiting time over RR.

-We observe significant IPC improvements in PVC (2.5×) and IIX (1.22×) applications, as the miss rate drastically reduces by 89% and 27%, respectively. As expected, we do not observe significant performance improvements in SD2, WP, and SPMV as there is no reduction in miss rate compared to RR

Effect of CTA-Aware-Locality

-The primary advantage of ***CTA-Aware-Locality*** is the reduced number of memory requests due to better cache utilization, and as a result of this, we also observe an improvement in DRAM bandwidth utilization due to reduced contention in DRAM banks

-The main advantage of this scheme is further reduced L1 miss rates. We observe 11% average IPC improvement (6% decrease in *MemoryBlockCycles*) over *CTA-Aware*, and 25% (17% hmean, 21% gmean) over RR. We observe 81% IPC improvement in IIX, primarily because of 69% in L1 miss rates

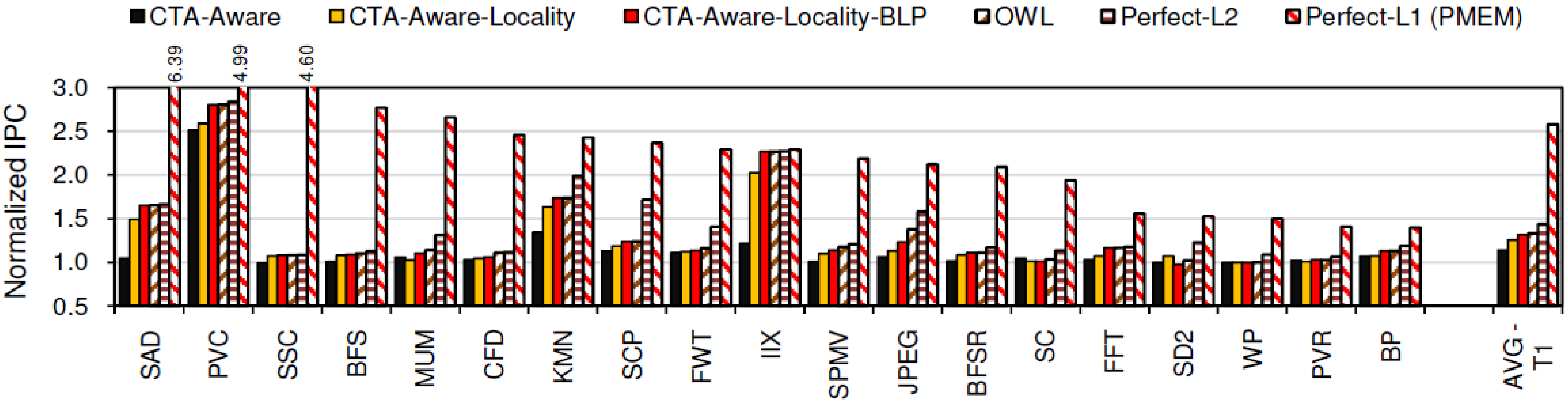
Effect of CTA-Aware-Locality-BLP

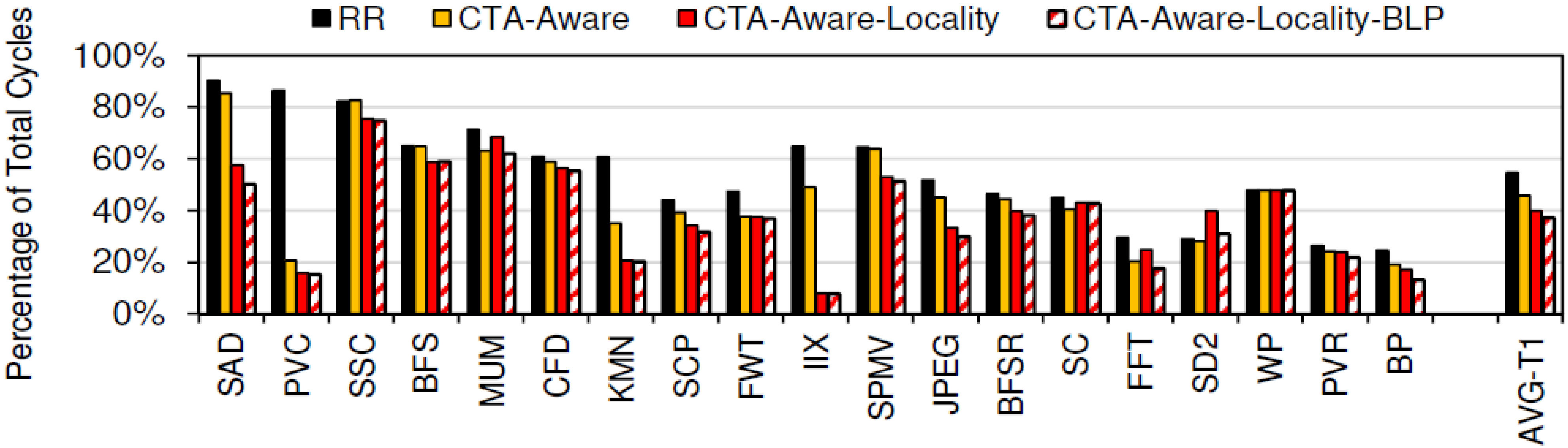
-In this scheme, we strive to achieve better BLP at the cost of row locality.

-Using this scheme, on average, we observe 6% IPC (4% hmean, 4% gmean) improvement, and 3% decrease in *MemoryBlockCycles* over ***CTA-Aware- Locality***.

-The effects of the 62% reduction in row locality is outweighed by the 46% increase in BLP, yielding a 10% IPC improvement over ***CTA-Aware-Locality***

Combined Effect of OWL





Sensitivity to group size

-The minimum number of warps in a group should be at least equal to the number of pipeline stages

-We observe that when minimum group size is 8 warps, we get the best IPC improvements

- We observe that as the number of banks increases, the effectiveness of *CTA-Aware-Locality-BLP* increases.

Overview

Introduction

Background

Challenges

Contribution

Summary

Summary

The key idea in OWL is to take advantage of characteristics of cooperative thread arrays (CTAs) to concurrently improve cache hit rate, latency hiding capability, and DRAM bank parallelism in GPGPUs. OWL achieves these benefits by

1) selecting and prioritizing a group of CTAs scheduled on a core, thereby improving both L1 cache hit rates and latency tolerance

2) scheduling CTA groups that likely do not access the same memory banks on different cores, thereby improving DRAM bank parallelism

3) employing opportunistic memory-side prefetching to take advantage of already open DRAM rows, thereby improving both DRAM row locality and cache hit rates.

Any Questions