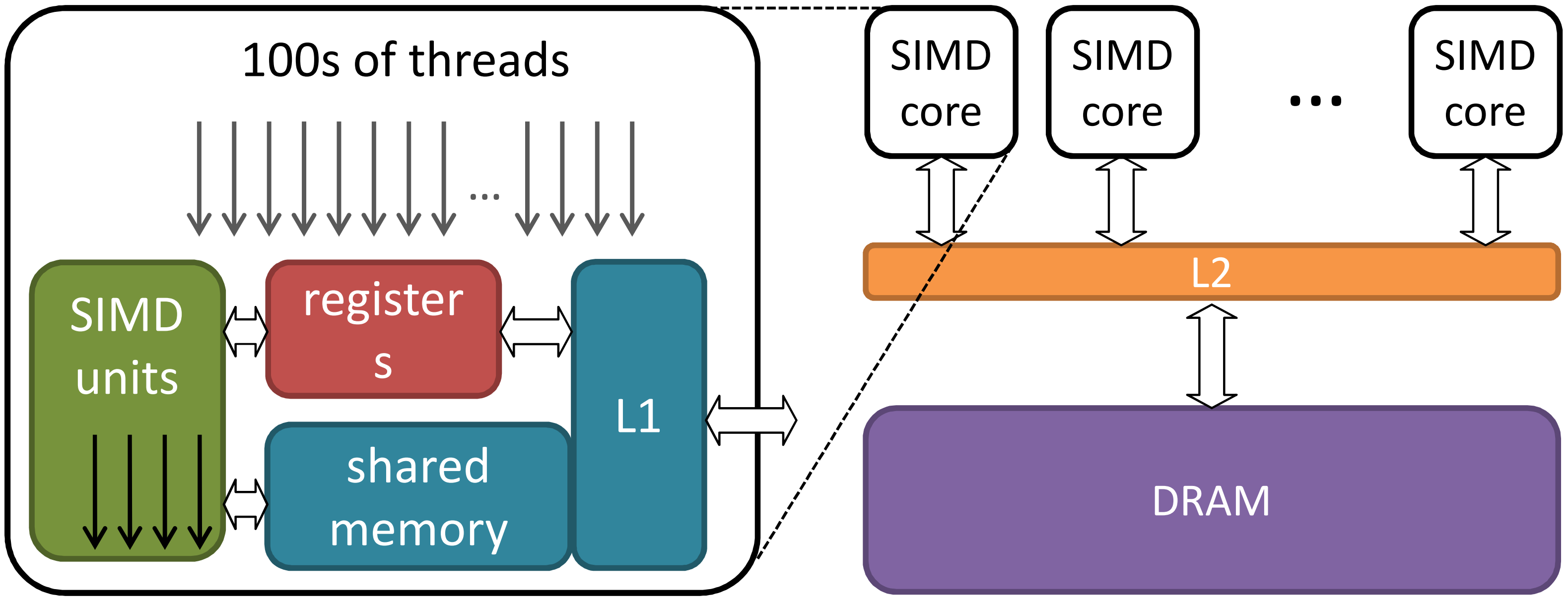
Characterizing and Improving the Use of Demand-Fetched Caches in GPUs

Wenhao Jia, Kelly A. Shaw*, Margaret Martonosi

Princeton University   University of Richmond*

GPU Caches: Challenges & Opportunities

- GPUs are particularly suitable for data parallel applications
- Newer GPUs have sizable demand-fetched caches
  - Up to 48 KB L1, around 1 MB L2
- NVIDIA GPU L1 caches have considerable configurability
  - Whole program on/off, size, per-instruction on/off

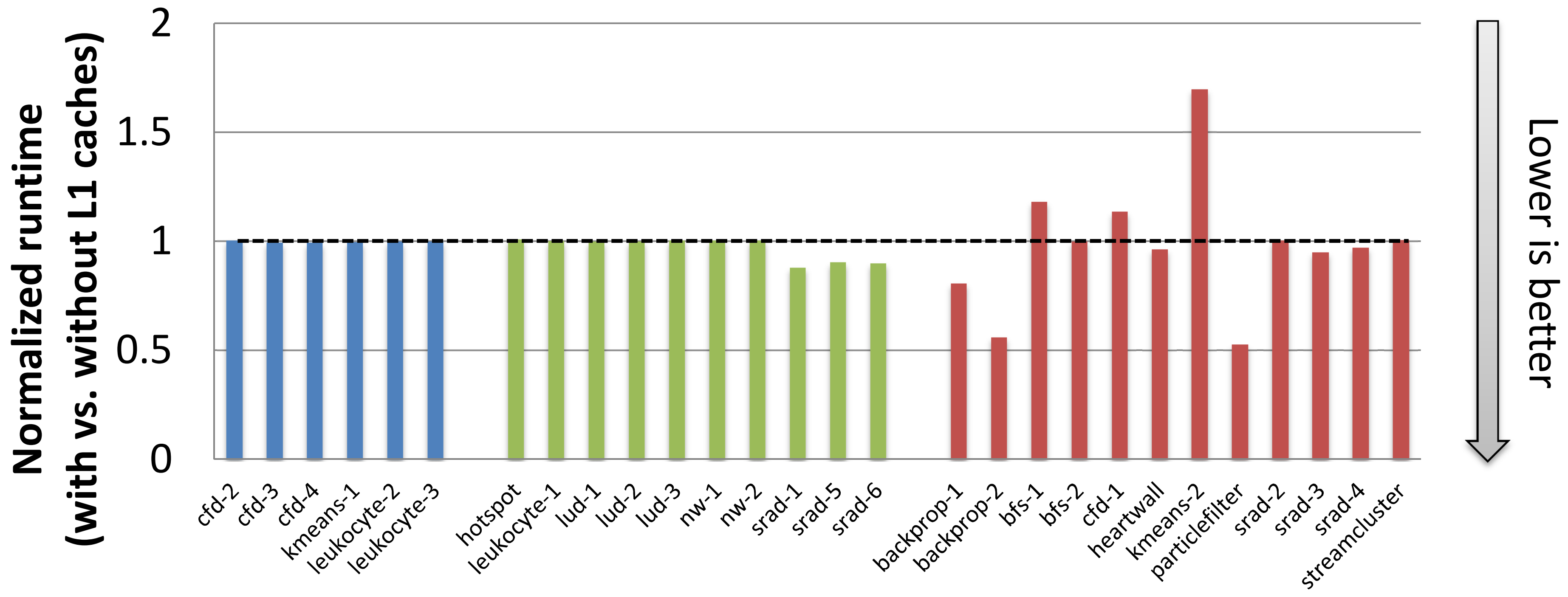


CPU Caches vs. GPU Caches

- Caches in CPUs
  - Well-studied, optimize hit-rates to reduce DRAM latency
  - 3C's: compulsory, capacity, conflict
- Caches in GPUs
  - New and less well-studied
  - GPU applications have unique characteristics: large number of threads, computational throughput-oriented, complex memory hierarchy, etc.
  - Need a new way to think about and characterize GPU cache utility

When Are GPU Caches Helpful?

Rodinia benchmark suite, NVIDIA Tesla C2070



Class I kernels: Texture/constant loads only, requests don't use L1 caches

Class II kernels: Mainly use shared memory, limited benefits from caching

Class III kernels: Use DRAM and thus caches frequently, but see <u>large and unpredictable</u> performance variations from caching

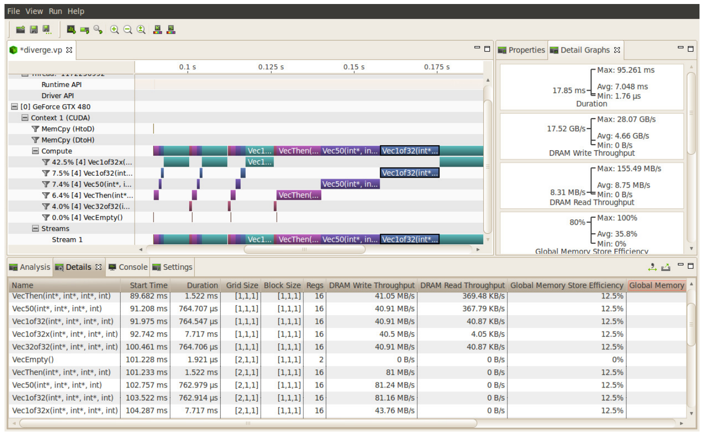
Research Goals

- Characterize performance impact of caches for GPU applications running on real systems
- Present a memory access locality taxonomy to help programmers and compilers analyze cache utility
- Propose and test an automatable, static cache utility analysis algorithm

Prior work has studied data layout and scratchpad optimization, but not caches or their configurability

Experimental Platform

- Real-system measurements
- NVIDIA Tesla C2070 GPU (Fermi) + Visual Profiler

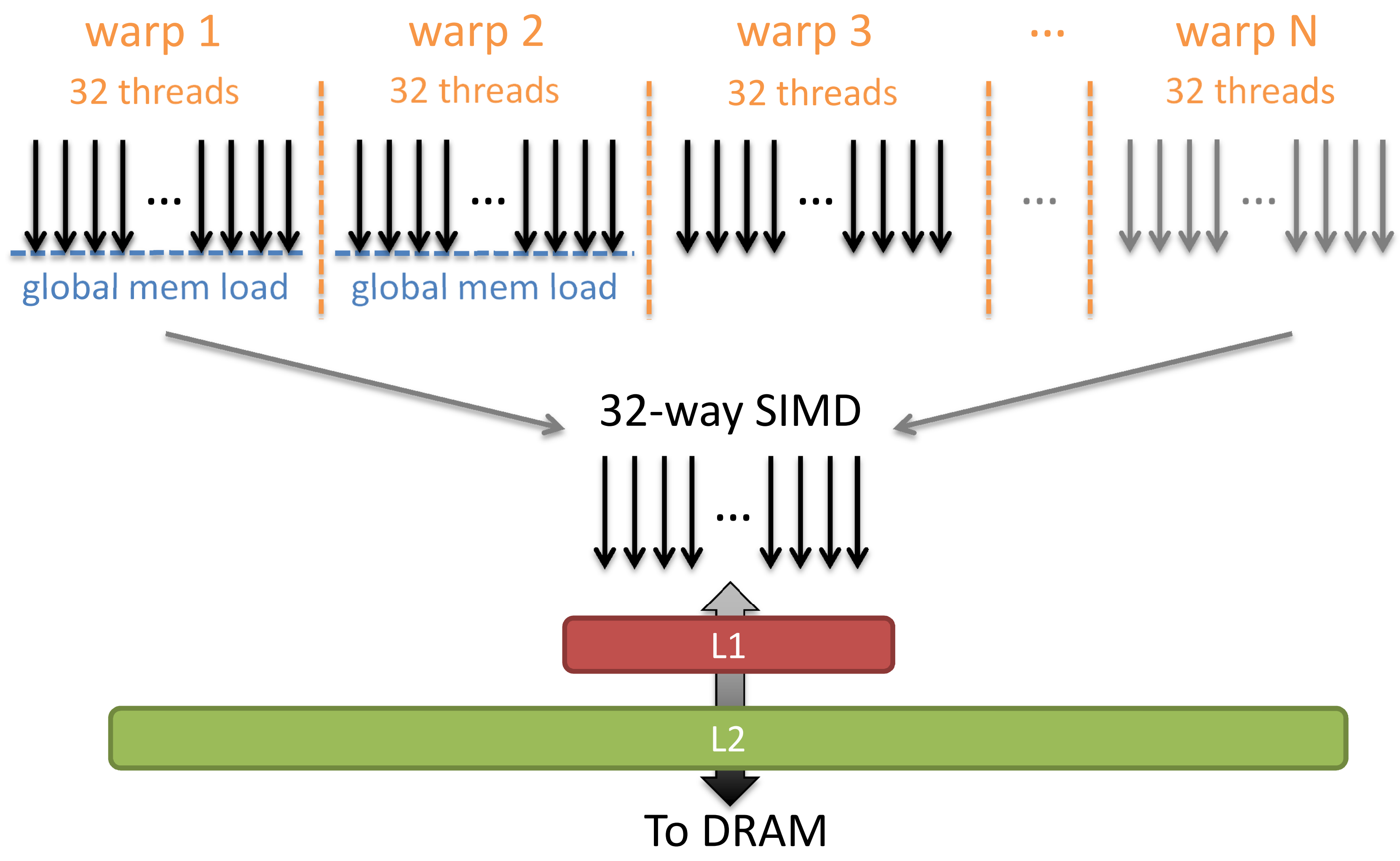


Name	Start Time	Duration	Grid Size	Block Size	Regs	DRAM Write Throughput	DRAM Read Throughput	Global Memory Store Efficiency	Global Memory
VecThen(int*, int*, int*, int)	89.682 ms	1.522 ms	[1,1,1]	[1,1,1]	16	41.05 MB/s	369.48 KB/s	12.5%	
Vec50(int*, int*, int*, int)	91.208 ms	764.707 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	367.79 KB/s	12.5%	
Vec1of32(int*, int*, int*, int)	91.975 ms	764.547 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
Vec1of32x(int*, int*, int*, int)	92.742 ms	7.717 ms	[1,1,1]	[1,1,1]	16	40.5 MB/s	4.05 KB/s	12.5%	
Vec32of32(int*, int*, int*, int)	100.461 ms	764.706 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
VecEmpty()	101.228 ms	1.921 µs	[2,1,1]	[1,1,1]	2	0 B/s	0 B/s	0%	
VecThen(int*, int*, int*, int)	101.233 ms	1.522 ms	[2,1,1]	[1,1,1]	16	81 MB/s	0 B/s	12.5%	
Vec50(int*, int*, int*, int)	102.757 ms	762.979 µs	[2,1,1]	[1,1,1]	16	81.24 MB/s	0 B/s	12.5%	
Vec1of32(int*, int*, int*, int)	103.522 ms	762.914 µs	[2,1,1]	[1,1,1]	16	81.16 MB/s	0 B/s	12.5%	
Vec1of32x(int*, int*, int*, int)	104.287 ms	7.717 ms	[2,1,1]	[1,1,1]	16	43.76 MB/s	0 B/s	12.5%	

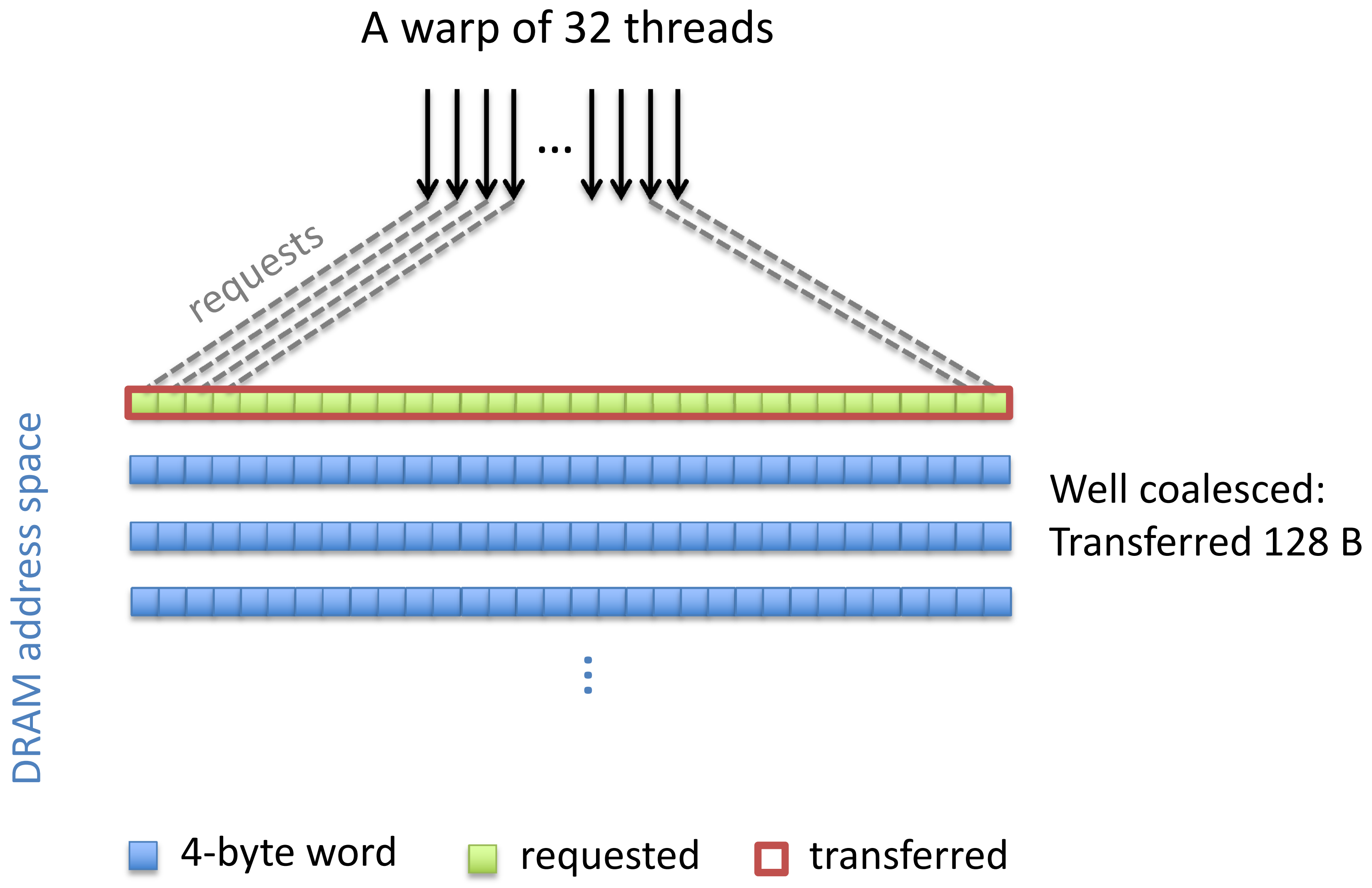
- Rodinia benchmark suite
  - 11 GPGPU applications with 28 kernels

Background: Warps Hide Memory Latency

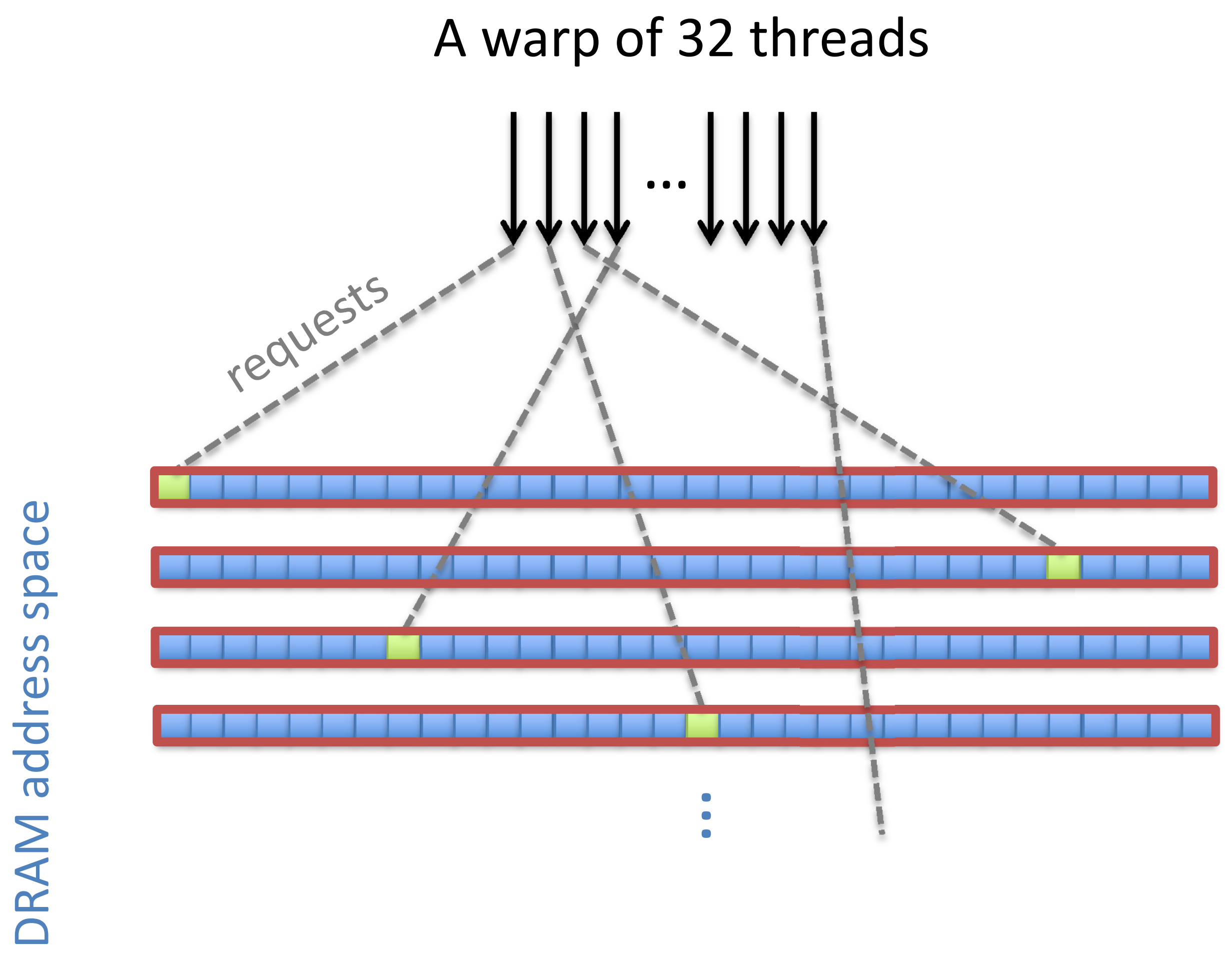
Hundreds of concurrent hardware threads in a thread block



Background: Effective Coalescing



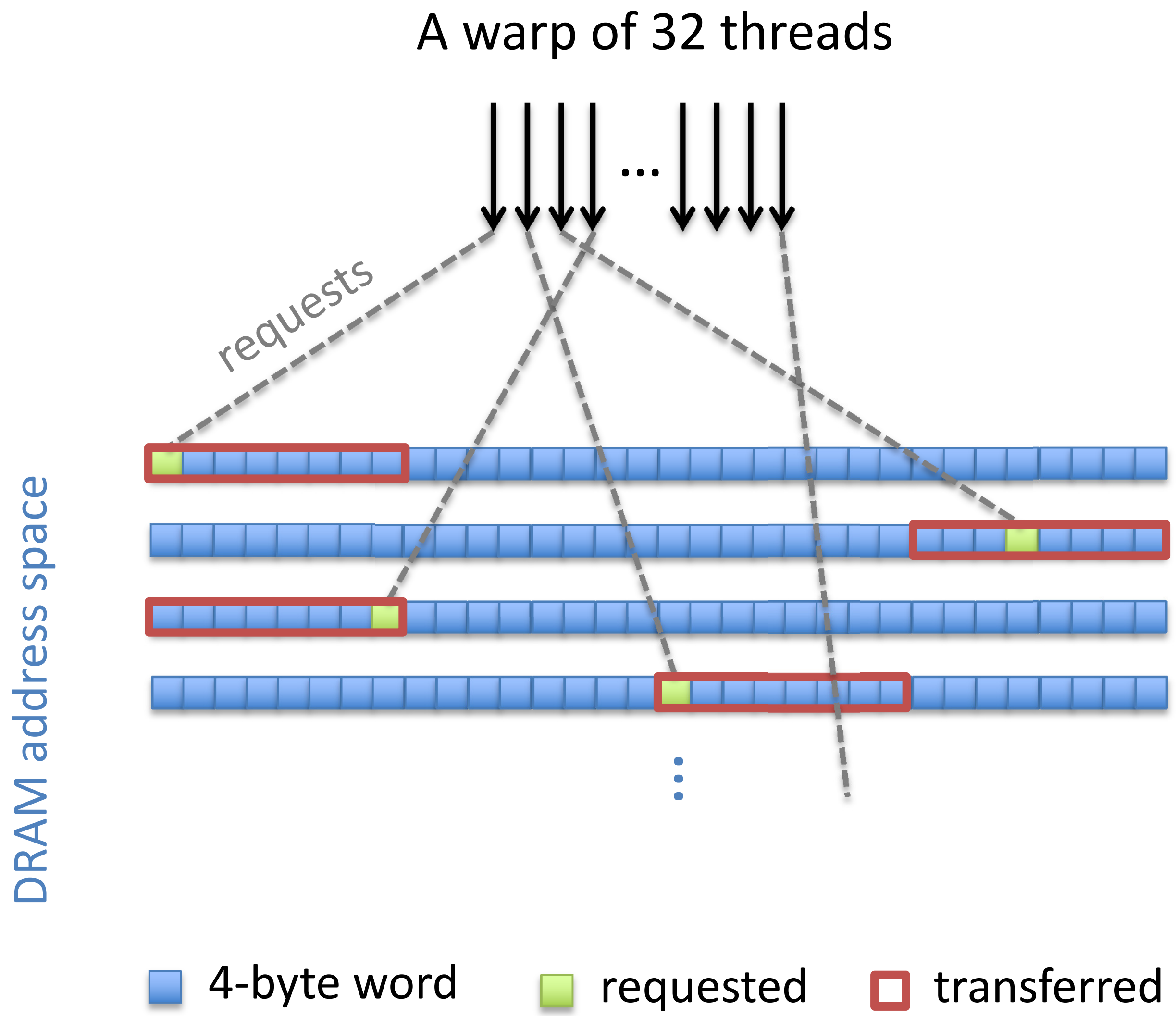
Background: Poor Coalescing with Caching On



Scattered accesses:
Transferred
128B × 32 = 4 KB

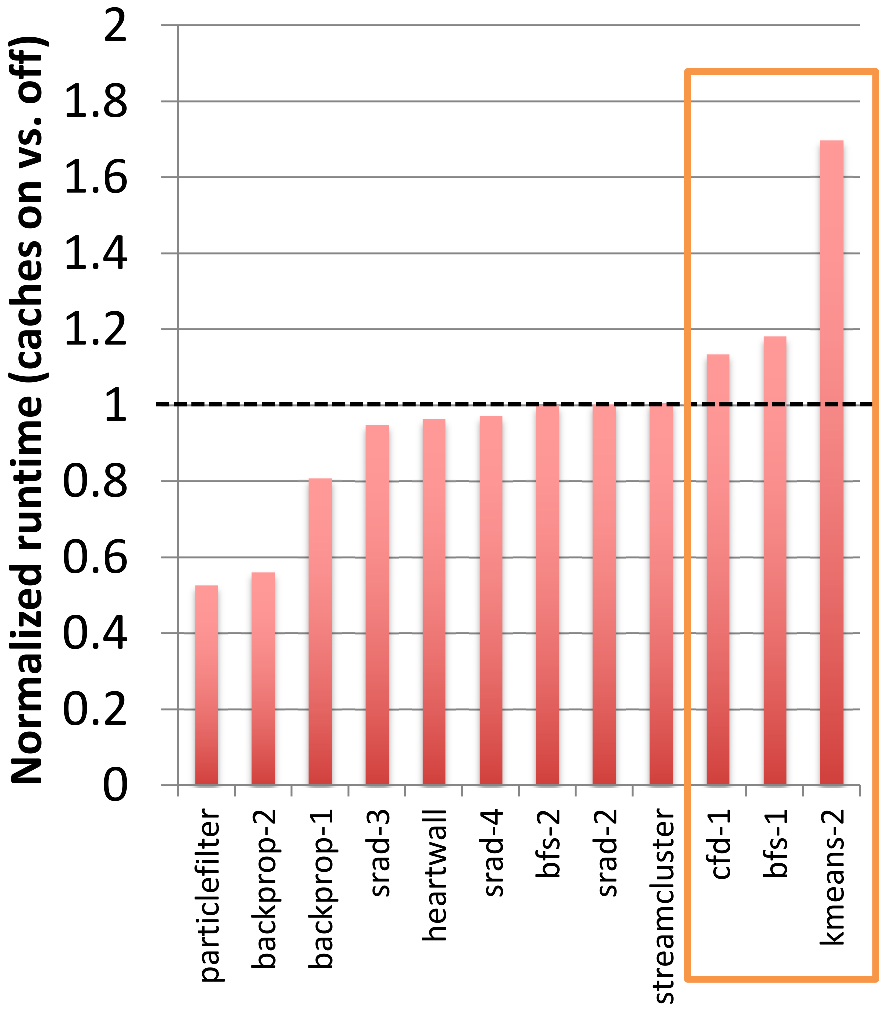
🟦 4-byte word 🟩 requested 🟥 transferred

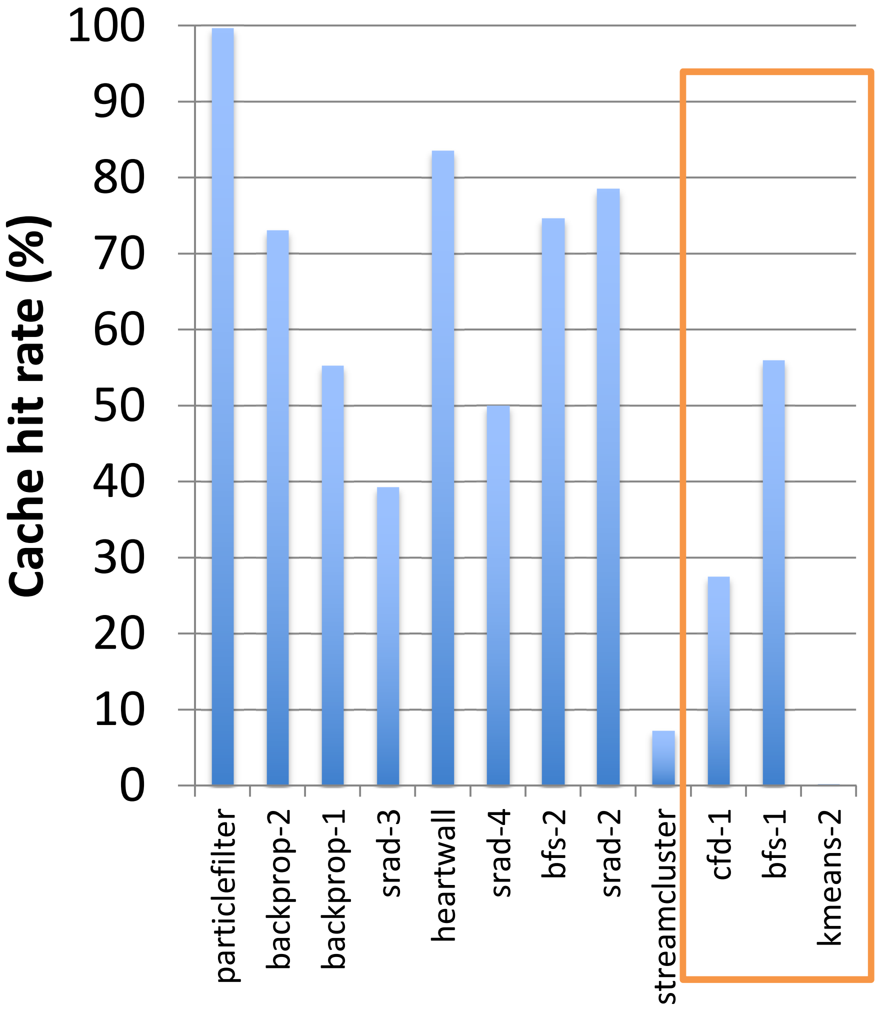
Background: Poor Coalescing with Caching Off



Turning caches off reduces memory request size:
Transferred
32B × 32 = 1 KB

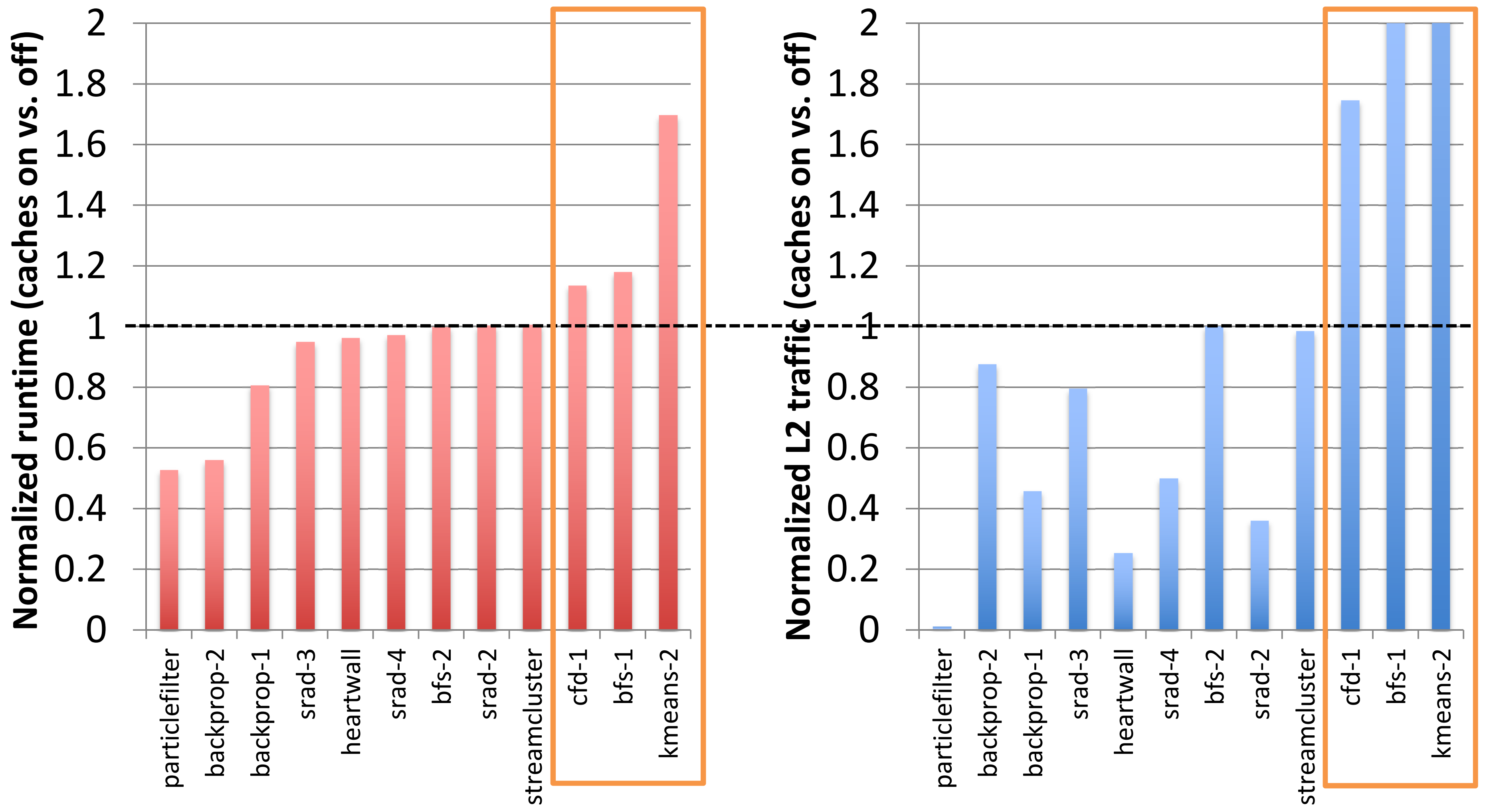
How to Predict Cache Utility?





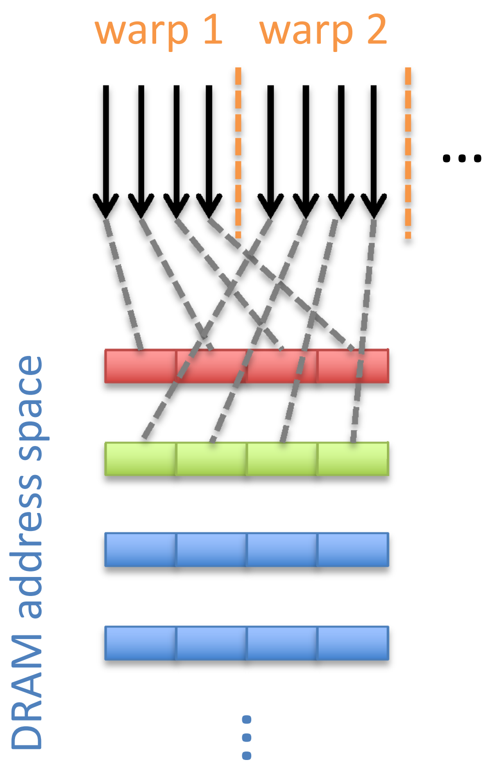
Performance and cache hit rate are poorly correlated

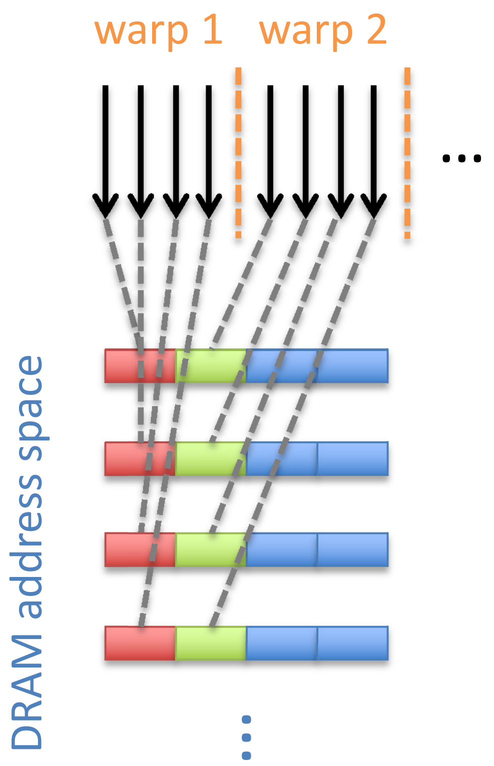
Global Memory Traffic Is a Better Indicator

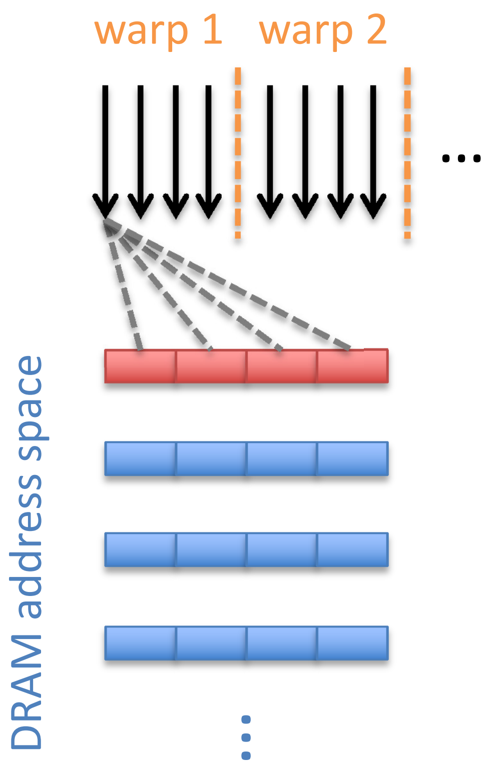


L2-to-L1 read traffic better correlates with runtime changes

Memory Access Locality Taxonomy







Within-warp locality

Threads in the same warp access consecutive locations; coalesced, no caching needed

Within-block locality

Threads in the same block access consecutive locations; short-term caching needed

Cross-instruction reuse

The same thread accesses consecutive locations at different points in time; longer-term, difficult to cache

Compiler Algorithm to Estimate Cache Utility

- Assumptions: Consider each instruction in isolation, control caching on a per-instruction basis, assume arrays have zero base addresses

- Step 1: Compute load addresses of load instructions
- Step 2: Estimate cache-on and cache-off traffic
- Step 3: Decide whether to cache for each instruction

Example: Compute Load Addresses

- Use the constant folding/propagation framework to represent each thread’s global memory load address as a function of its thread ID (tid), constants, and uncomputable (∞)

```
// thread block size: 256
void toy (float *A, int *B, float *C, float *D) {
    int index = threadIdx.x;
    float x = A[index / 32];
    int y = B[index * 32];
    float z = C[y];
    D[index] = x * z;
}
```

Results

index = tid

load (tid / 32)

load (tid * 32)

load (∞)

Example: Estimate Memory Traffic

- Goal: For an entire thread block, estimate the amount of global memory traffic when all threads execute one particular load instruction with and without caching

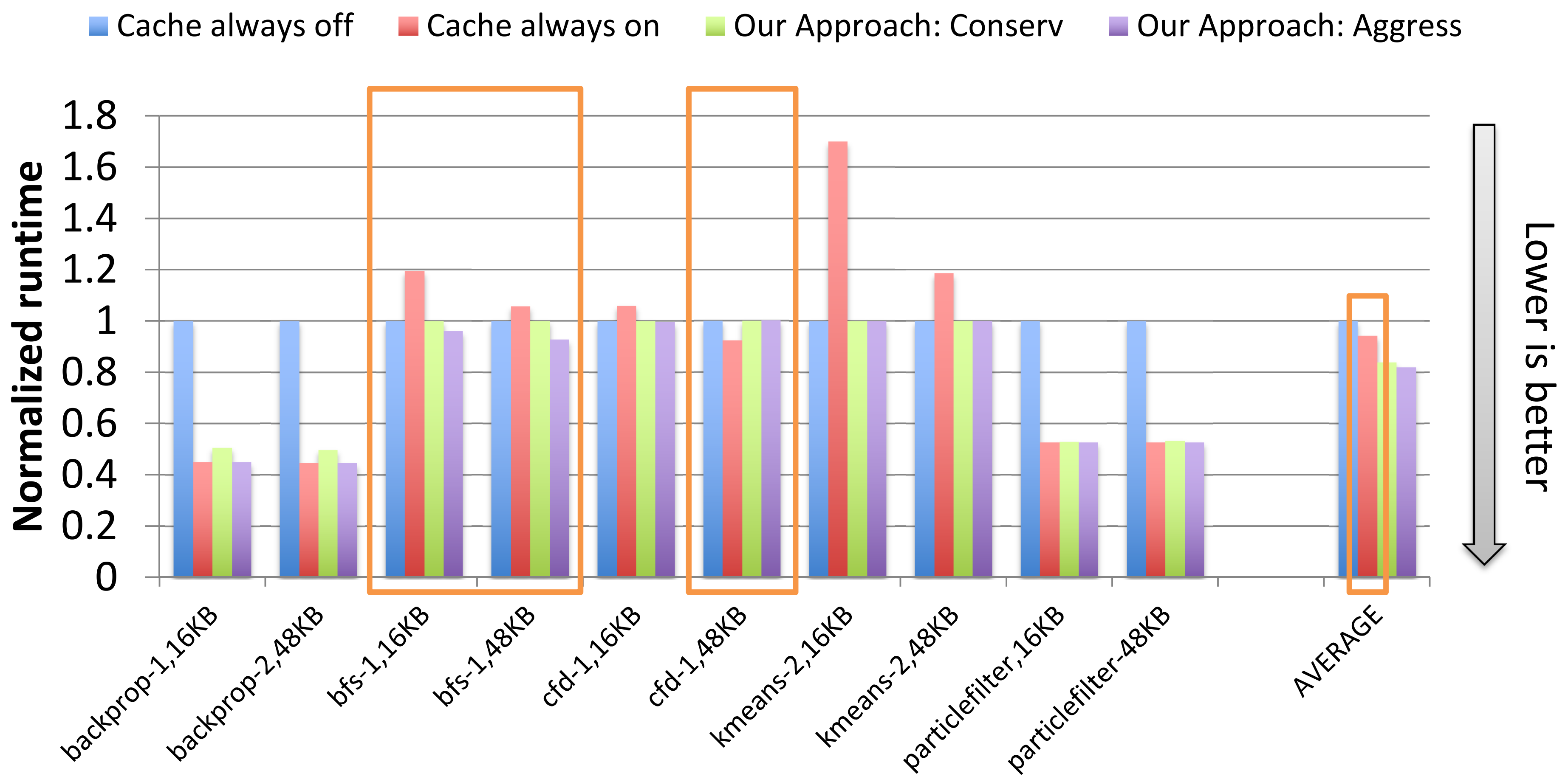
1 thread block

Load Inst	Expr	Requested Memory Addresses (Byte)				Cache-on Traffic	Cache-off Traffic	Cache?
		Warp 0	Warp 1	…	Warp 8			
A[index / 32]	tid / 32	0-3	4-7	…	28-31	128 B	32 × 8 = 256 B	Y
B[index * 32]	tid * 32	0-3 32-35 …	1024-1027 …	…	8196-8199 …	128 × 256 = 32 KB	32 × 256 = 8 KB	N
C[y]	∞							N

Caching Decisions

- For each load instruction, when an entire thread block executes it:
  - Cache-on traffic < cache-off traffic: cache for this load
  - Cache-on traffic > cache-off traffic: don't cache for this load
  - Cache-on traffic = cache-off traffic
    - Conservative strategy: turn caching off for this load
    - Aggressive strategy: turn caching on for this load
  - Unknown/uncomputable: likely to be scattered accesses based on our empirical observations, turn caching off for this load

Cache On/Off Algorithm: Results



In most cases, our compiler algorithm matches or improves on best all-or-nothing choice

Conclusions

- We should base GPU cache utility analysis on reducing global memory bandwidth requirement instead of hiding latency
- A memory access locality taxonomy helps analyze global memory traffic and consequently cache utility
- Automatable, static algorithm improves the average caching benefit from 5.8% to 18%

THANK YOU!