Divergence-Aware Warp Scheduling

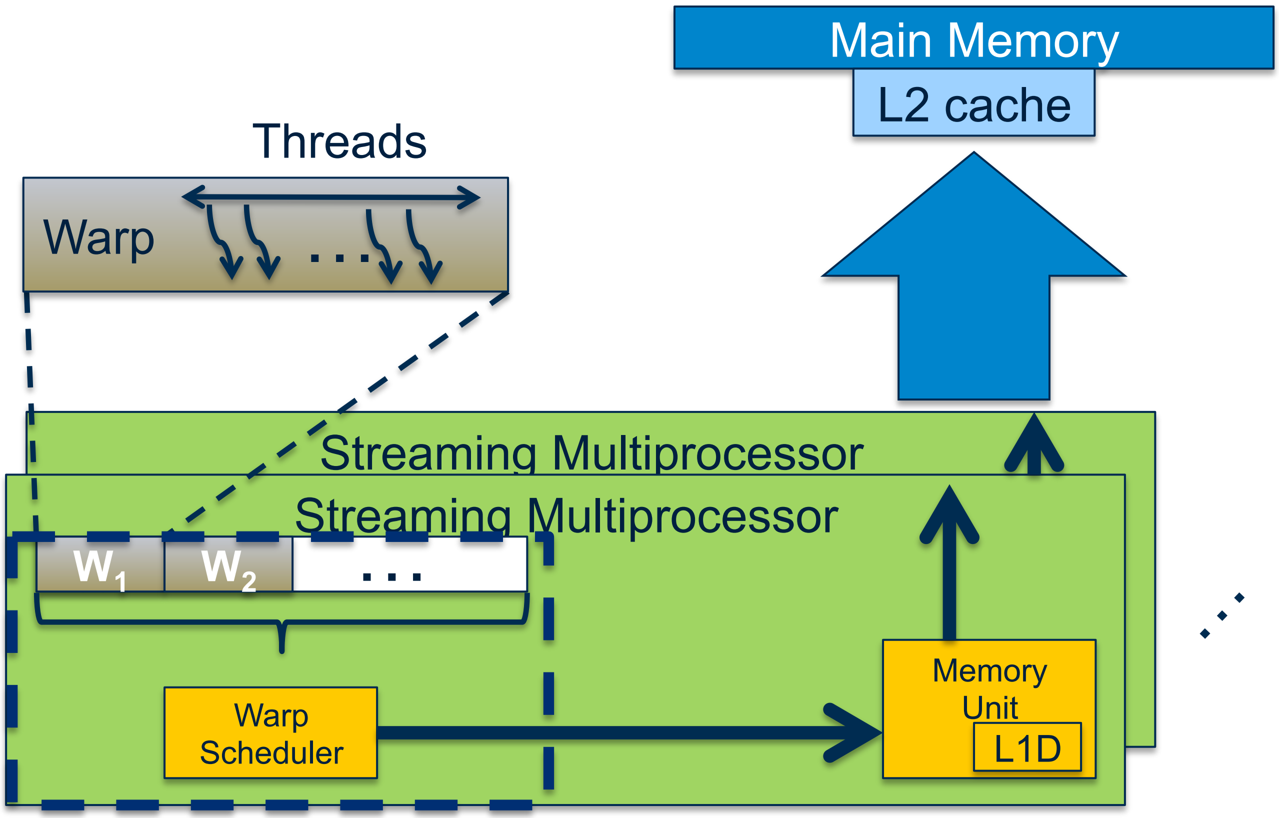
Timothy G. Rogers[1], Mike O'Connor[2], Tor M. Aamodt[1]

[1]The University of British Columbia
[2]NVIDIA Research

MICRO 2013 Davis, CA

GPU

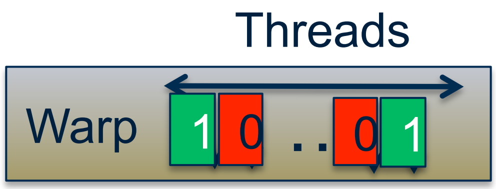
- 10000's concurrent threads
- Grouped into warps
- Scheduler picks warp to issue each cycle



2 Types of Divergence

Branch Divergence

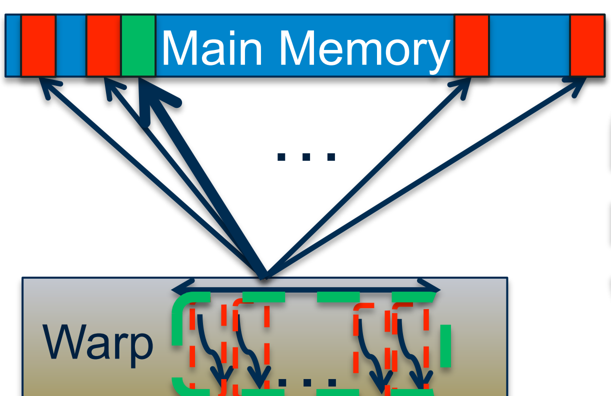
```
if(…) {
      …
}
```



Effects functional unit utilization

Aware of branch divergence

Memory Divergence



Can waste memory bandwidth

Aware of memory divergence

AND

Focus on improving performance

Motivation

- **Improve performance of programs with memory divergence**
  - Parallel irregular applications
  - Economically important (server computing, big data)

- **Transfer locality management from SW to HW**
  - Software solutions:
    - Complicate programming
    - Not always performance portable
    - Not guaranteed to improve performance
    - Sometimes impossible

Programmability Case Study

Sparse Vector-Matrix Multiply

2 versions from SHOC

Divergent Version

Example 1 Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Load
            int col = cols[j];
            t += val[j] * vecTexRe

        }
        out[myRow] = t;
    }
}
```

Divergence

Each thread has locality

GPU-Optimized Version

Example 2 GPU-Optimized SPMV-Vector Kernel

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

                       limiters[myRow];
                       miters[myRow+1];

        for (int j = warpStart + id;
             j < warpEnd; j += warpSize)
        {
            int col = cols[j];
            mySum += val[j] * vecTexReader(col);
        }
        partialSums[t] = mySum;

        // Reduce partial sums
        if (id < 16)
            partialSums[t] += partialSums[t+16];
        if (id <  8)
            partialSums[t] += partialSums[t+ 8];
        if (id <  4)
            partialSums[t] += partialSums[t+ 4];
        if (id <  2)
            partialSums[t] += partialSums[t+ 2];
        if (id <  1)
            partialSums[t] += partialSums[t+ 1];

        // Write result
        if (id == 0)
        {
            out[myRow] = partialSums[t];
        }
    }
}
```

Explicit Scratchpad Use

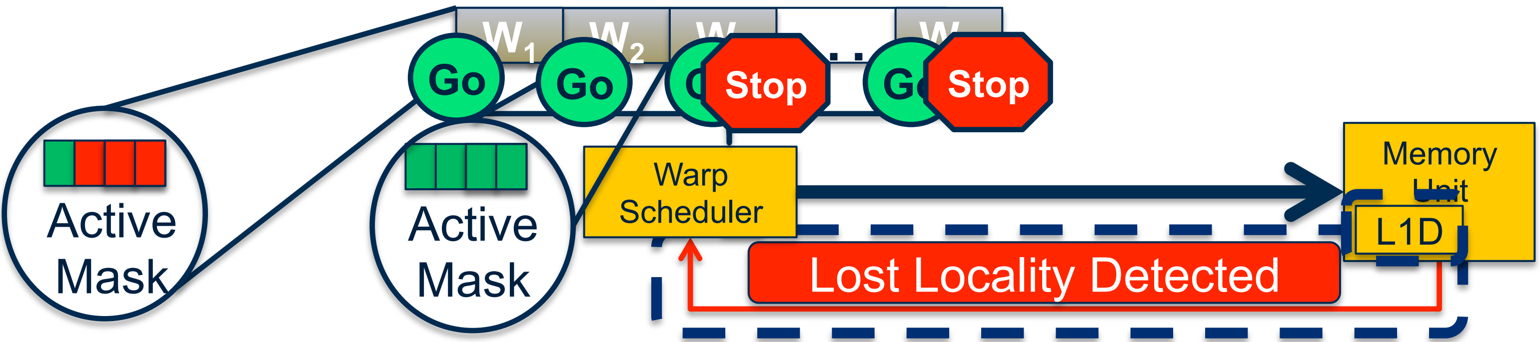
Dependent on Warp Size

Added Complication

Parallel Reduction

Previous Work

- **Scheduling used to capture intra-thread locality (MICRO 2012)**



Previous Work

Reactive
- Detects interference then throttles

Unaware of branch divergence
- All warps treated equally

Outperformed by profiled static throttling

Case Study: Divergent code 50% slowdown

Divergence-Aware Warp Scheduling

Predict and be Proactive

Adapt to branch divergence

Outperform static solution

Case Study: Divergent code <4% slowdown

Divergence-Aware Warp Scheduling

How to be proactive

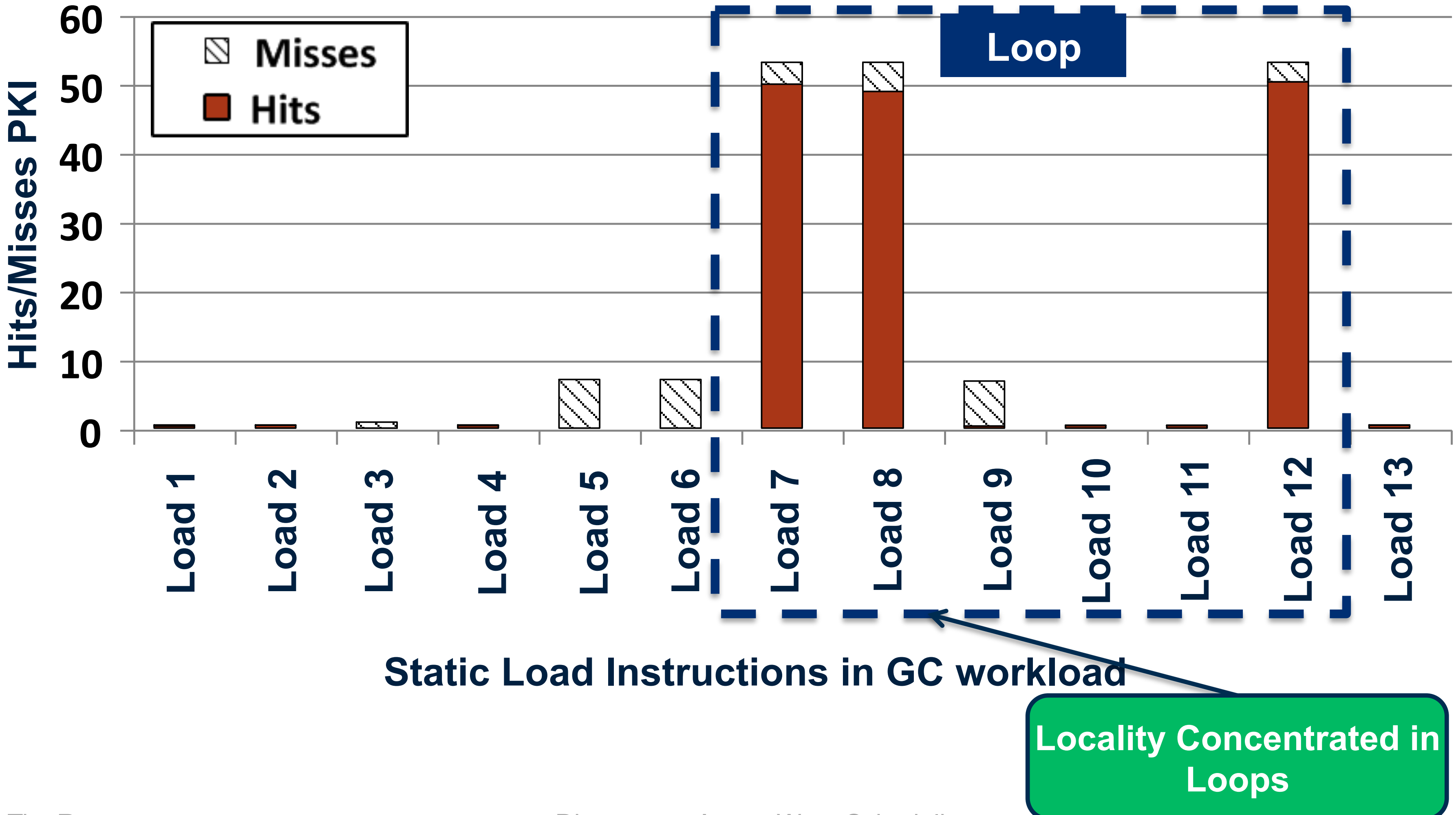
- Identify where locality exists
- Limit the number of warps executing in high locality regions

Adapt to branch divergence

- Create cache footprint prediction in high locality regions
- Account for number of active lanes to create **per-warp footprint prediction**.
- Change the prediction as branch divergence occurs.

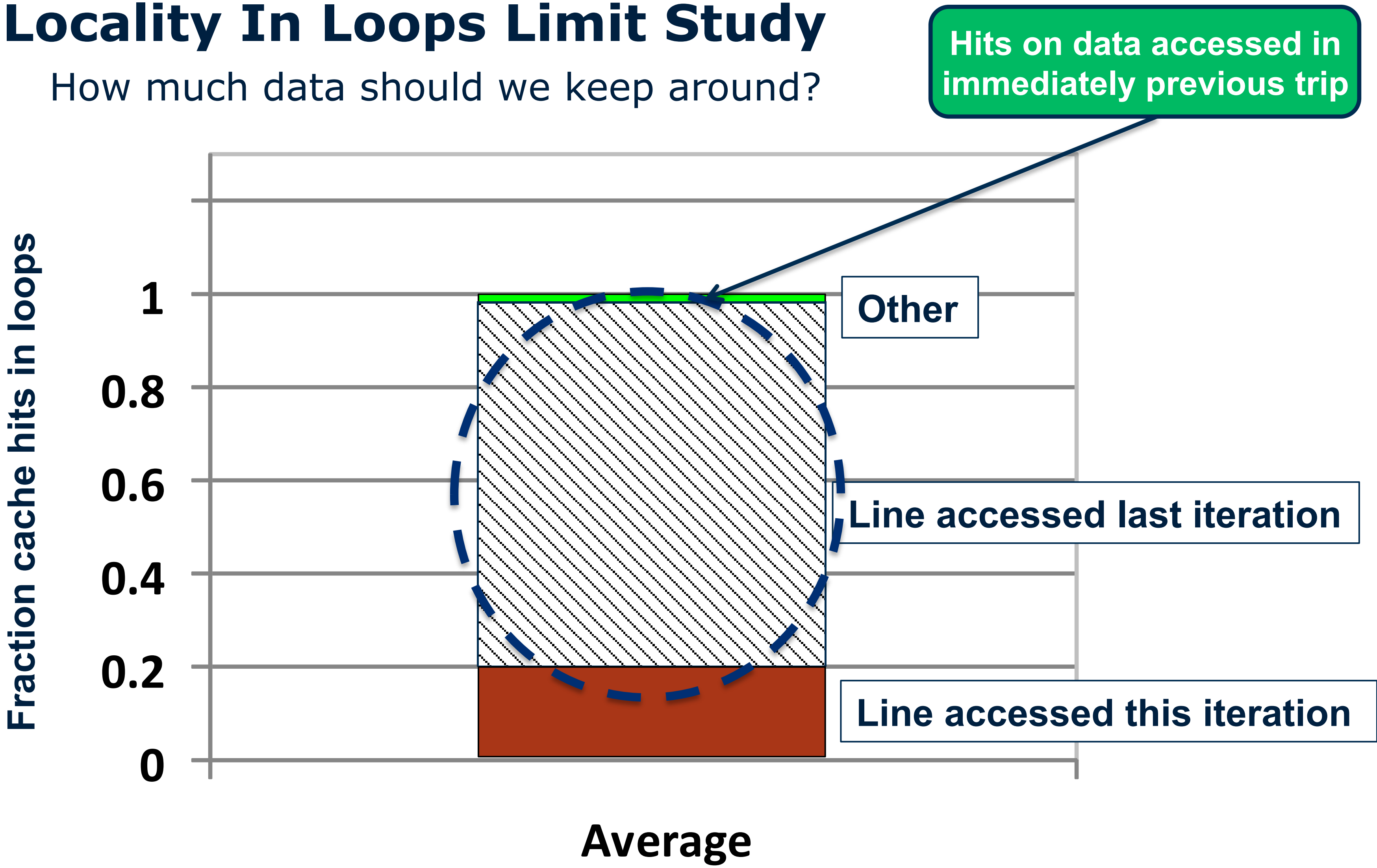
Where is the locality?

- **Examine every load instruction in program**



Locality In Loops Limit Study

How much data should we keep around?



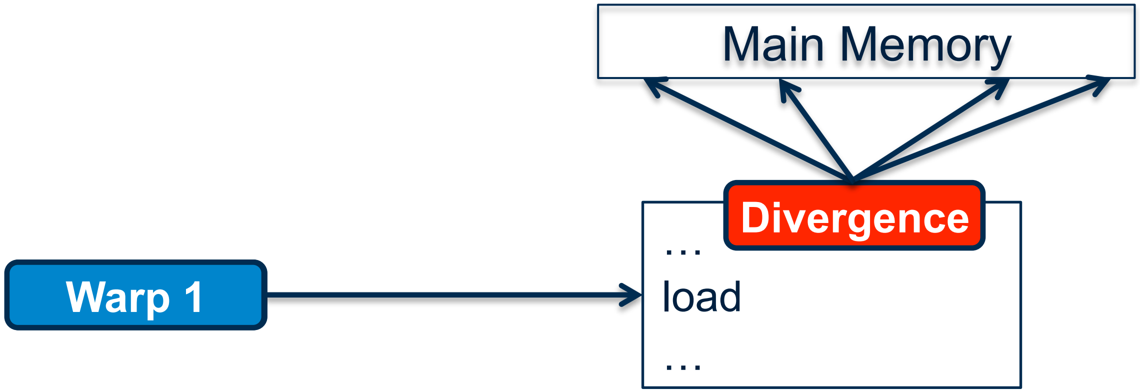
DAWS Objectives

1. **Predict the amount of data accessed by each warp in a loop iteration.**

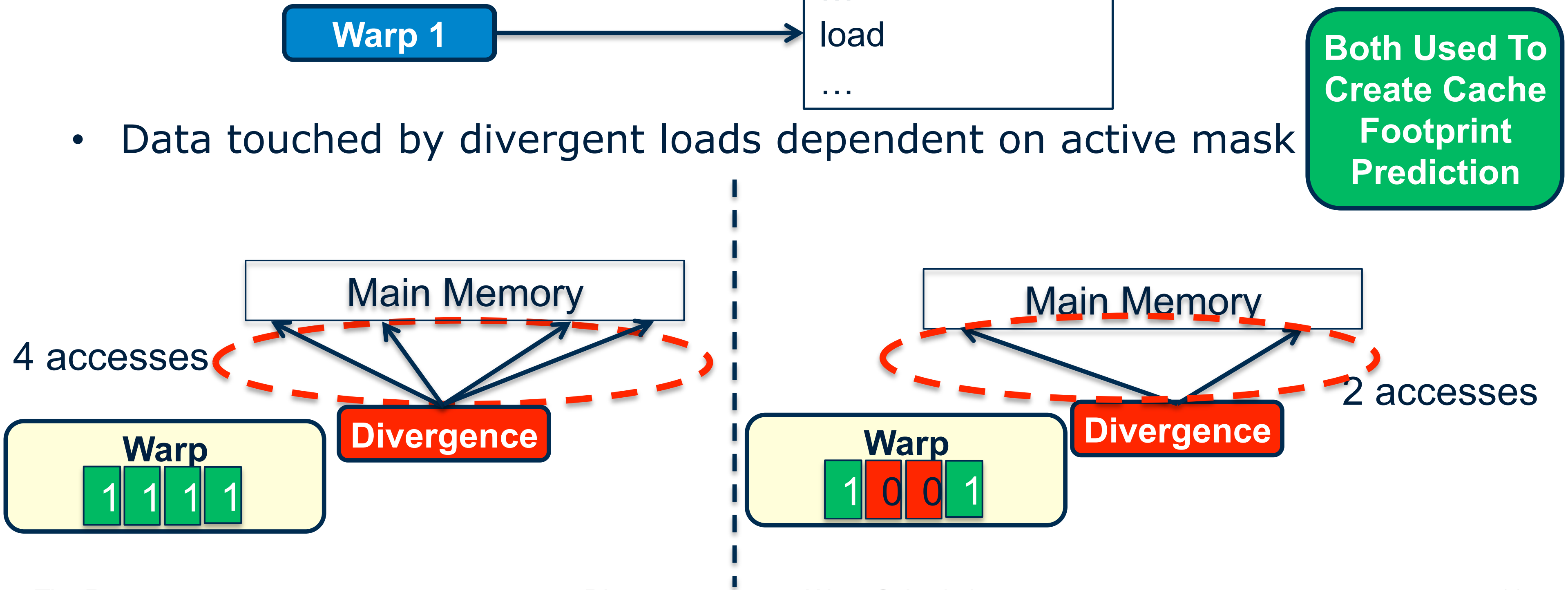
2. **Schedule warps in loops so that aggregate predicted footprint does not exceed L1D.**

Observations that enable prediction

- Memory divergence in static instructions is predictable



- Data touched by divergent loads dependent on active mask



Online characterization to create cache footprint prediction

1. Detect loops with locality

Some loops have locality

Some don't

Limit multithreading here

2. Classify loads in the loop

Loop with locality

```
while(…) {
        load 1    Diverged
        …
        load 2    Not Diverged
}
```

3. Compute footprint from active mask

Warp 0

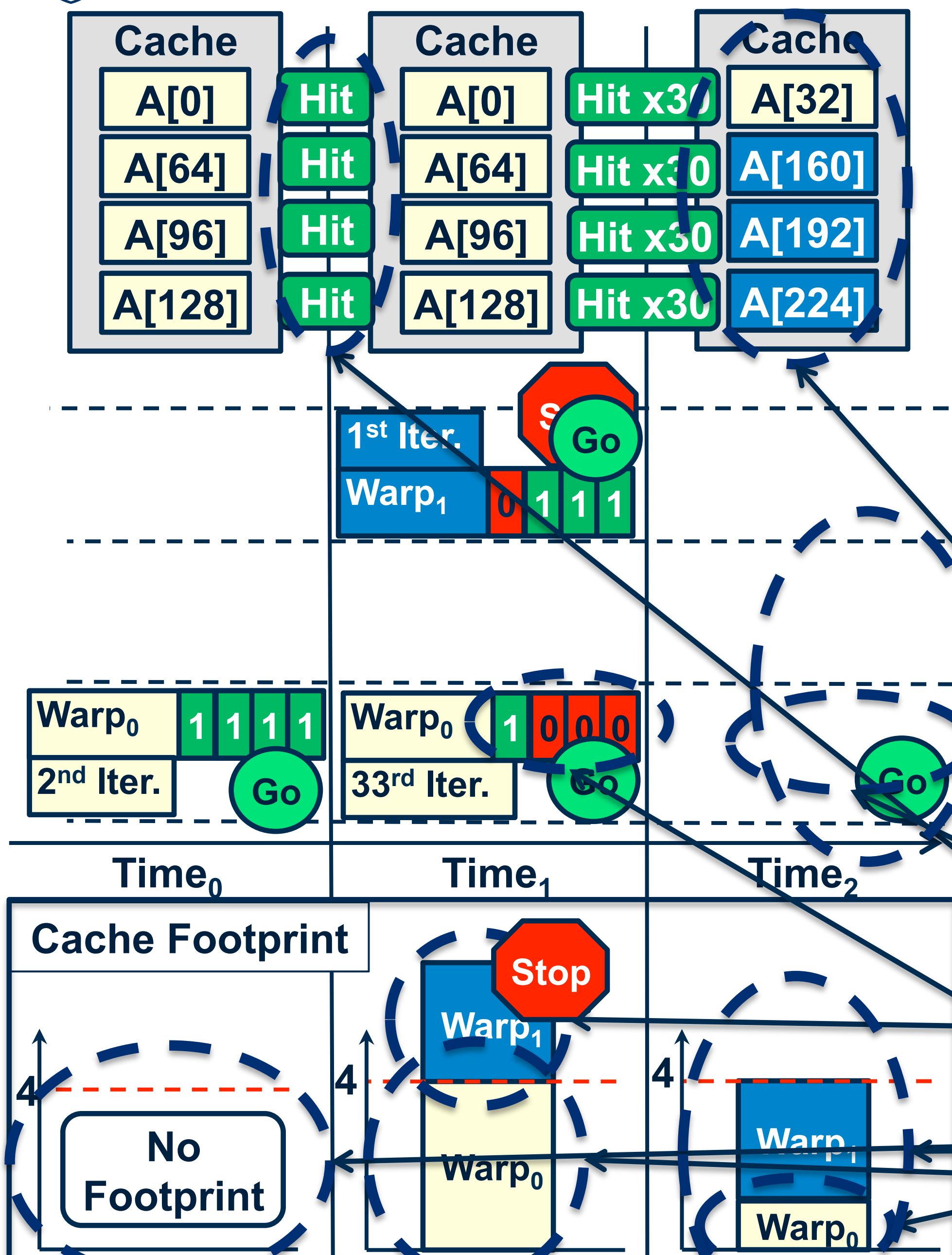
Loop with locality

```
while(…) {
        load 1    Diverged
        …
        load 2    Not Diverged
}
```

4 accesses
+
1 access

Warp 0's Footprint = 5 cache lines

DAWS Operation Example



Example Compressed Sparse Row Kernel

```
int C[]={0,64,96,128,160,160,192,224,256};
void sum_row_csr(float* A, …)  {
    float sum = 0;
    int i =C[tid];

  Loop
    while(i < C[tid+1]) {
        sum  += A[ i ];
        ++i;
    }
…
```

Divergent Branch

Memory Divergence

Methodology

GPGPU-Sim (version 3.1.0)

- 30 Streaming Multiprocessors
  - 32 warp contexts (1024 threads total)
- 32k L1D per streaming multiprocessor
- 1M L2 unified cache

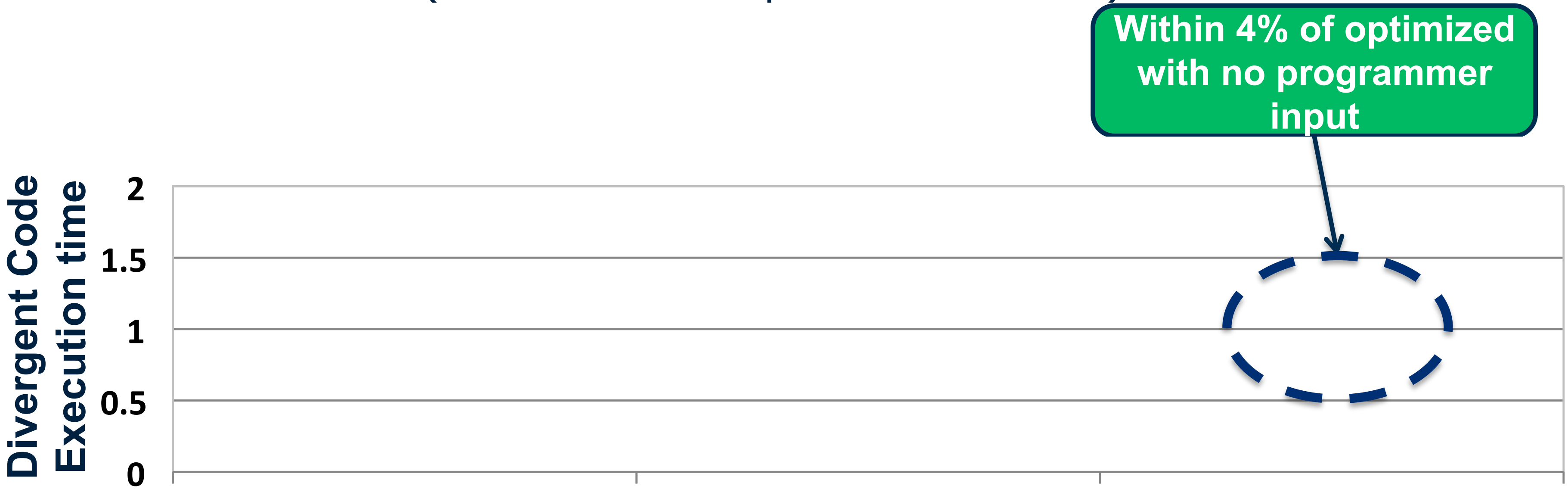
Compared Schedulers

- Cache-Conscious Wavefront Scheduling (CCWS)
- Profile based Best-SWL
- **Divergence-Aware Warp Scheduling (DAWS)**

More schedulers in paper

Sparse MM Case Study Results

- Performance (normalized to optimized version)



Sparse MM Case Study Results

- Properties (normalized to optimized version)

<20% increase in off-chip accesses

Divergent code issues 2.8x less instructions

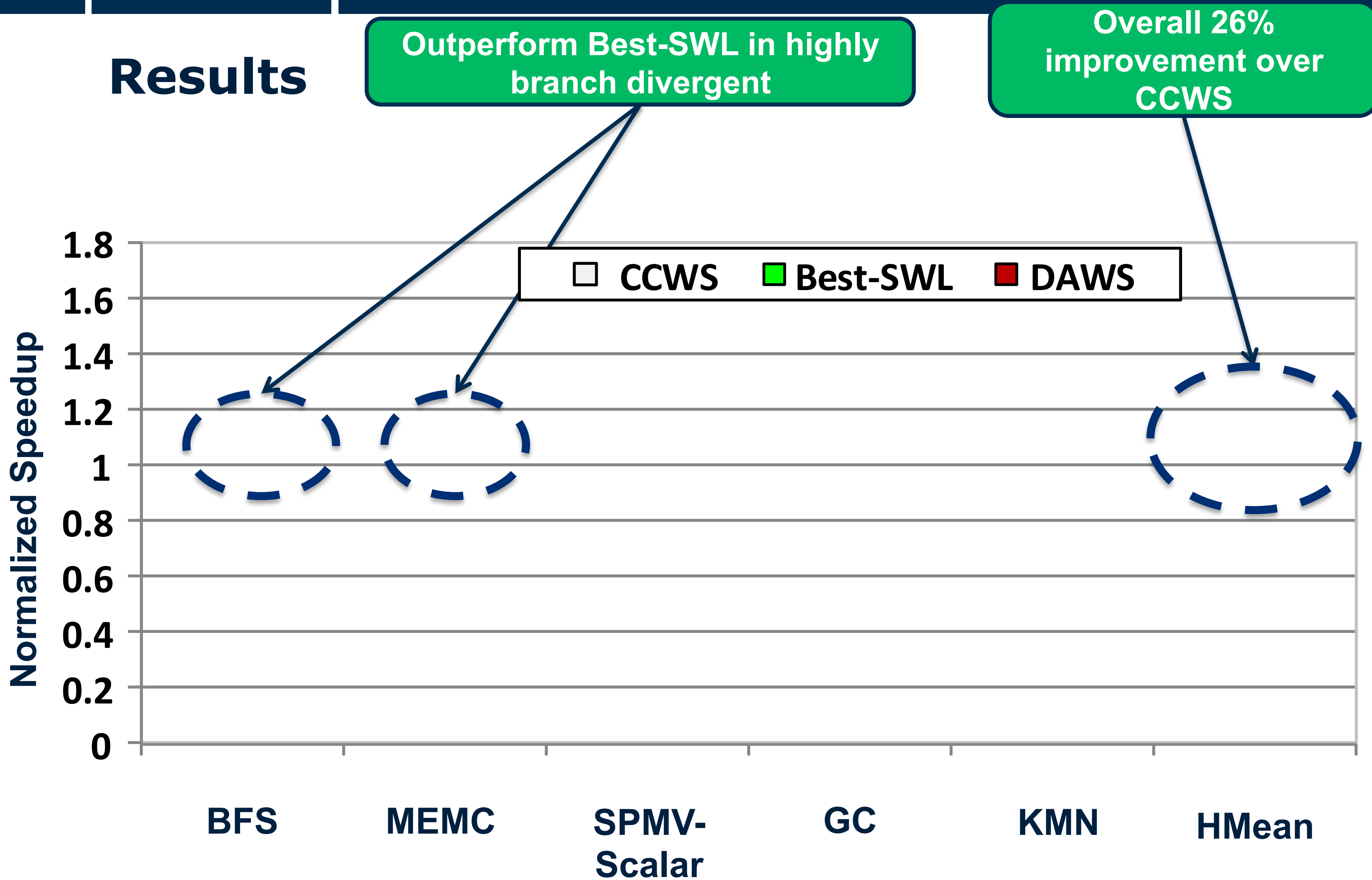
Divergent version now has potential energy advantages

Cache-Sensitive Applications

- Breadth First Search (BFS)
- Memcached-GPU (MEMC)
- Sparse Matrix-Vector Multiply (SPMV-Scalar)
- Garbage Collector (GC)
- K-Means Clustering (KMN)

Cache-Insensitive Applications in paper

Results


Outperform Best-SWL in highly branch divergent
Overall 26% improvement over CCWS
CCWS
Best-SWL
DAWS
Normalized Speedup
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
BFS
MEMC
SPMV-Scalar
GC
KMN
HMean

Summary

Questions?

Problem

Divergent loads in GPU programs.

- Software solutions complicate programming

Solution

DAWS

- Captures opportunities by accounting for divergence

Result

Overall **26%** performance improvement over CCWS

Case Study: Divergent code performs **within 4%** code optimized to minimize divergence