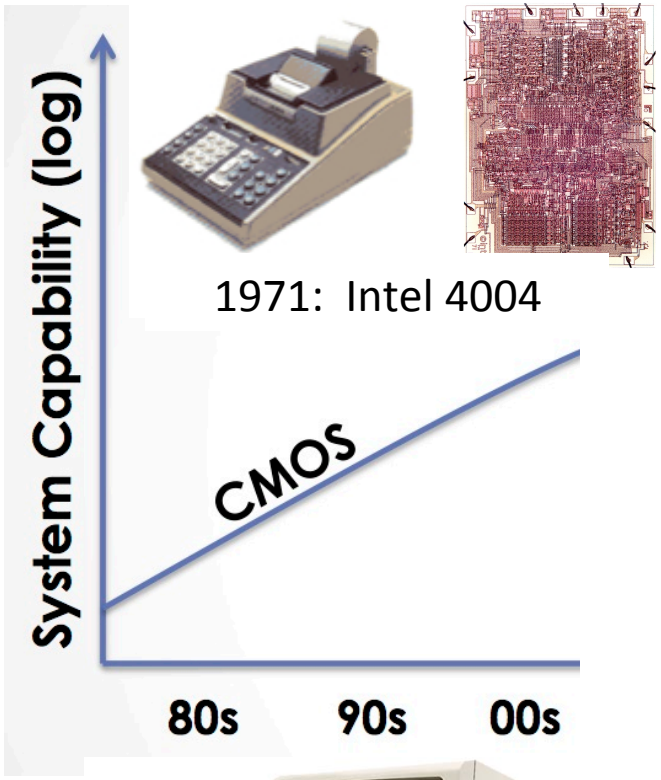
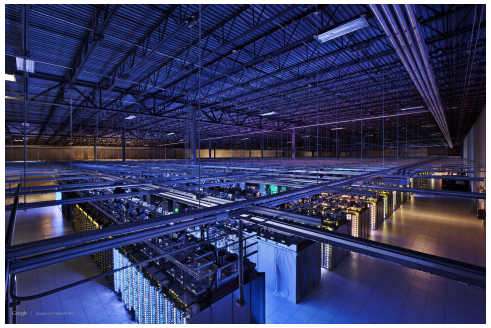
Part 3: Research Directions

Decreasing cost per unit computation



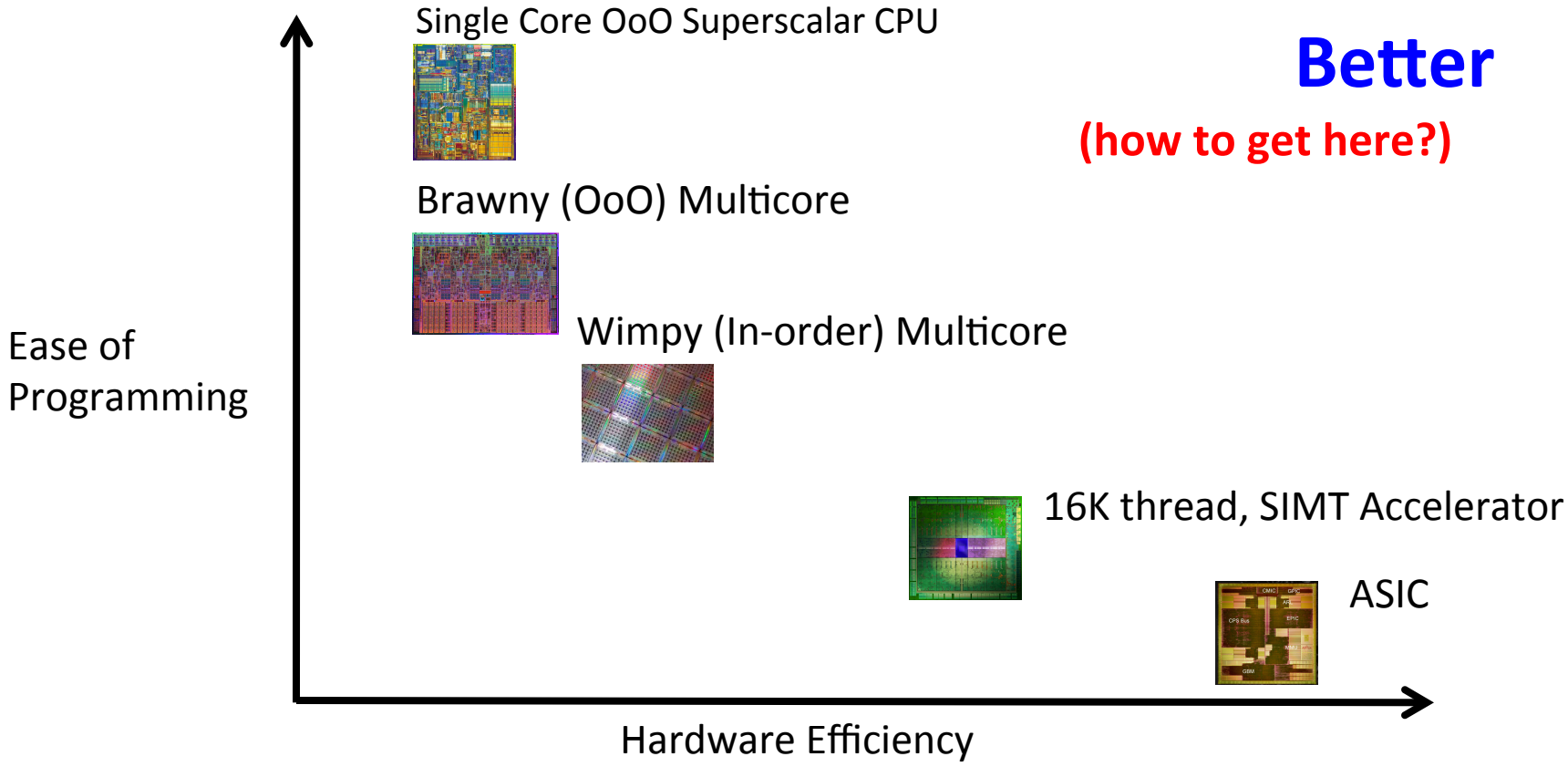
1971: Intel 4004

2007: iPhone



2012: Google datacenter

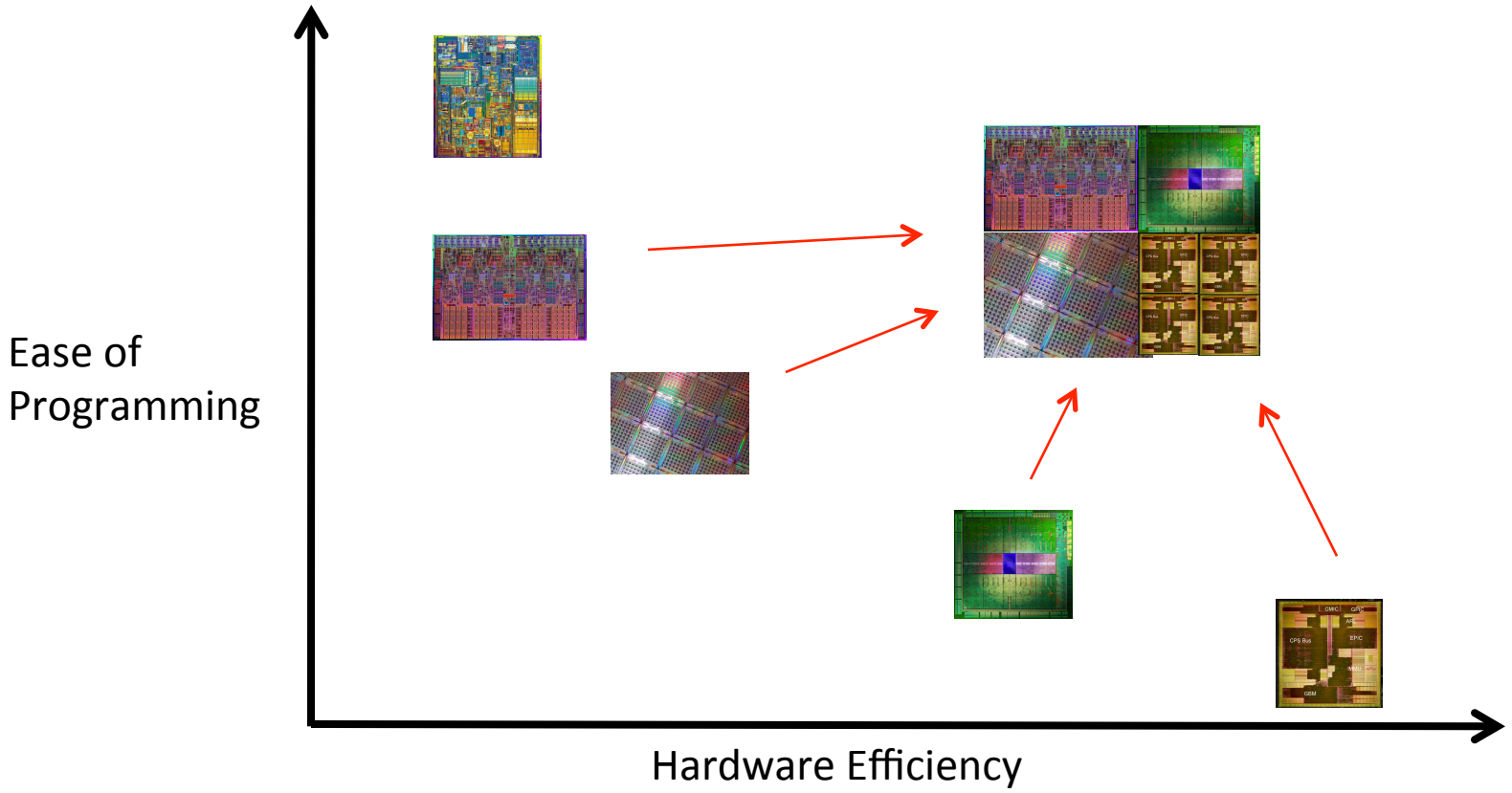
1981: IBM 5150



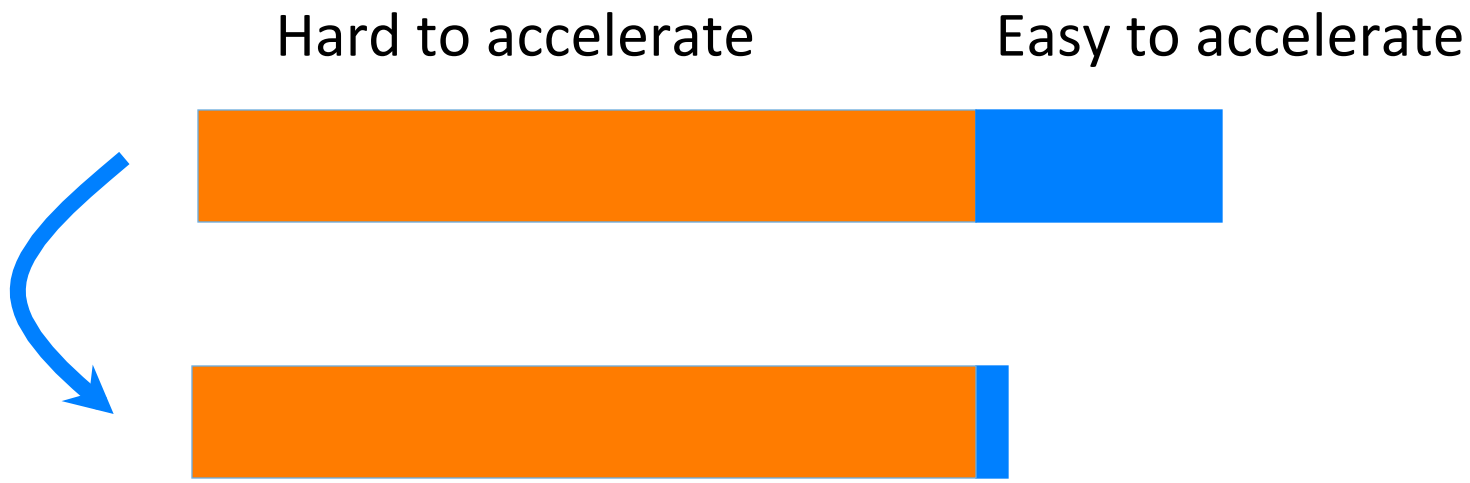
Better

(how to get here?)

Start by using right tool for each job…

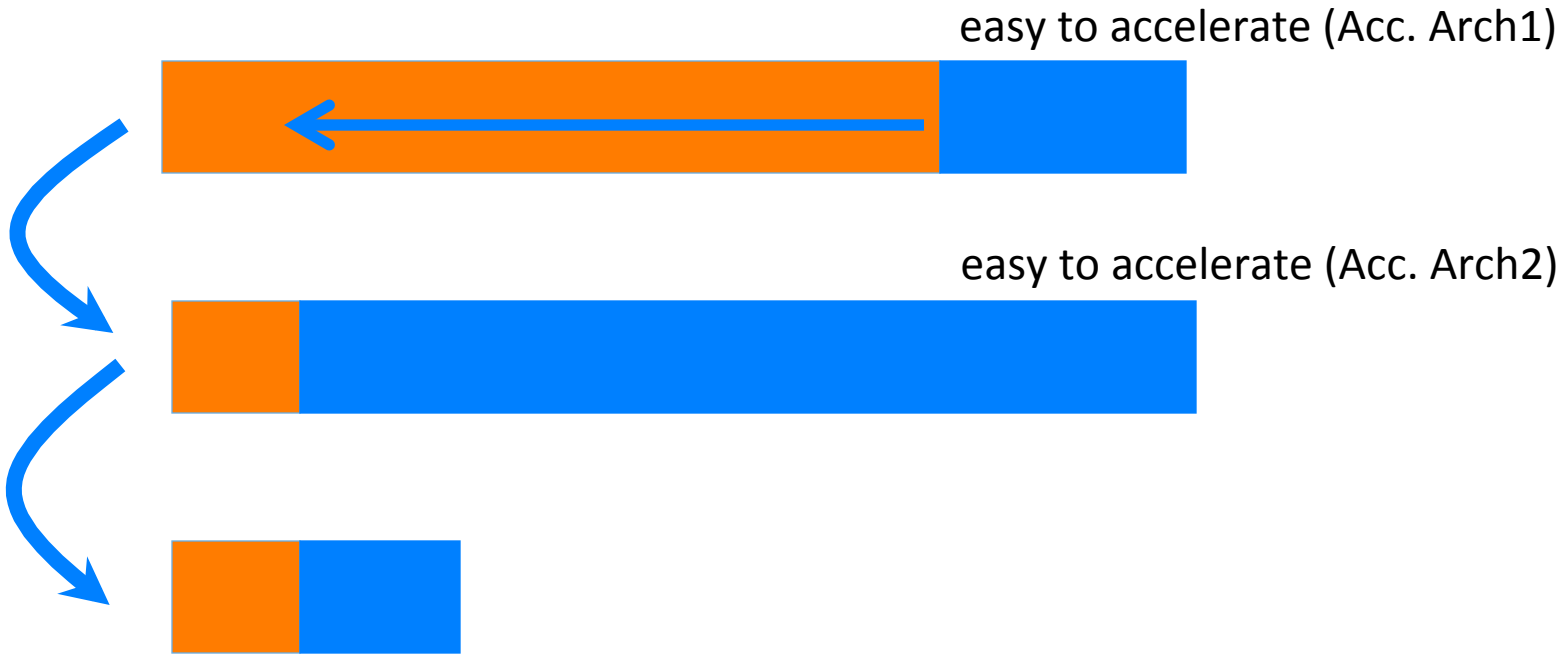


Amdahl's Law Limits this Approach



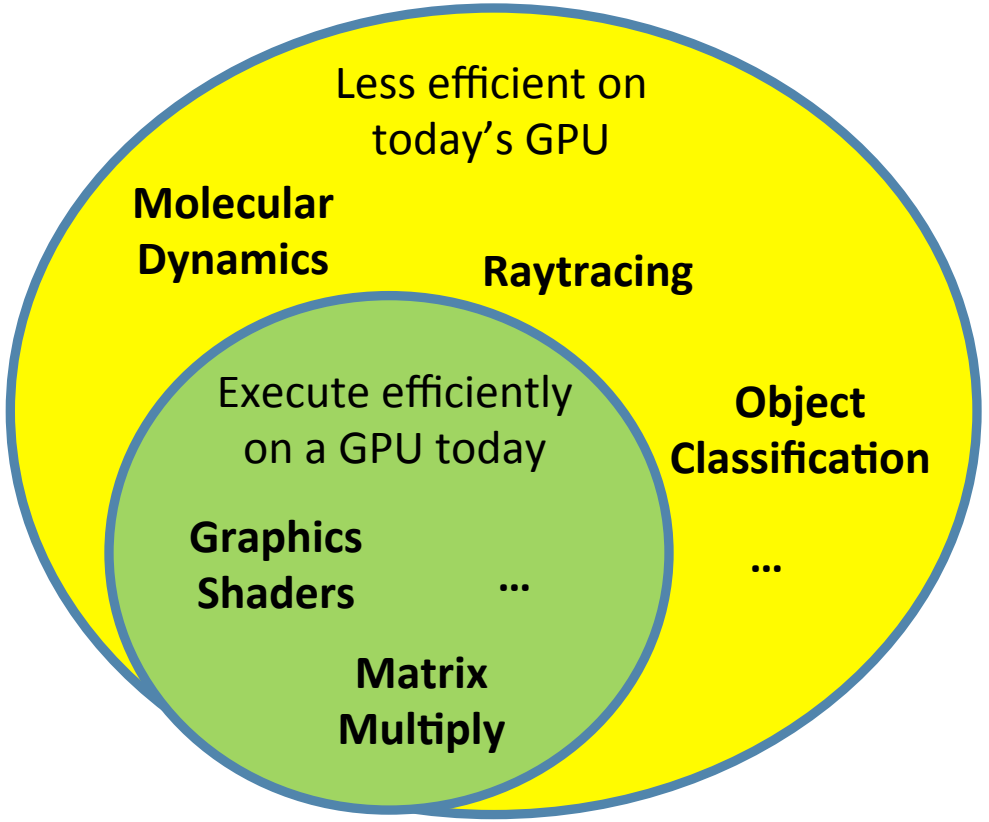
$$\text{Improvement}_{\text{overall}} = \frac{1}{\text{Fraction}_{\text{hard}} + \frac{1 - \text{Fraction}_{\text{hard}}}{\text{Improvement}_{\text{easy}}}}$$

Question: Can dividing line be moved?



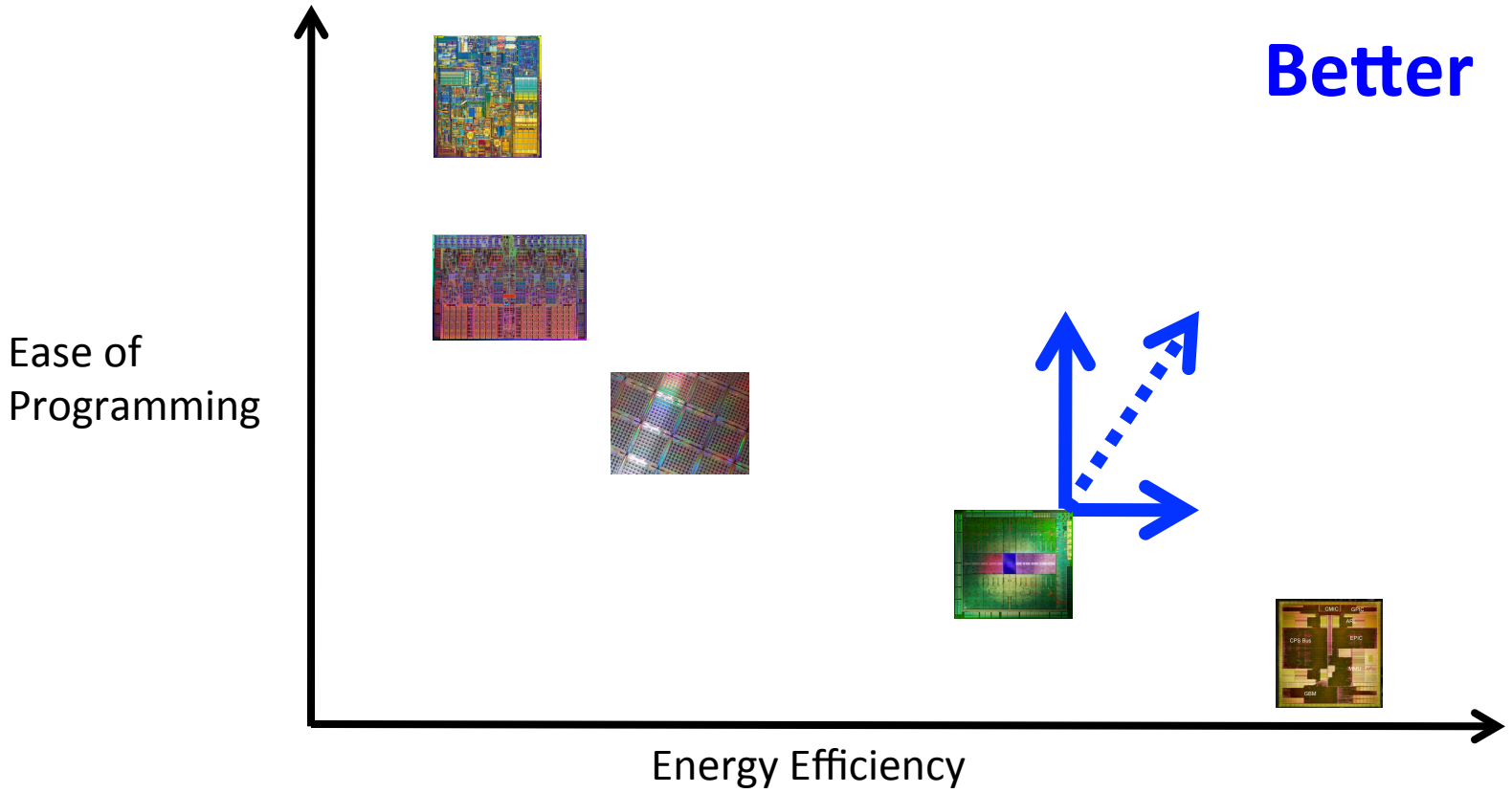
Forward-Looking GPU Software

- Still Massively Parallel
- Less Structured
  - Memory access and control flow patterns are less predictable



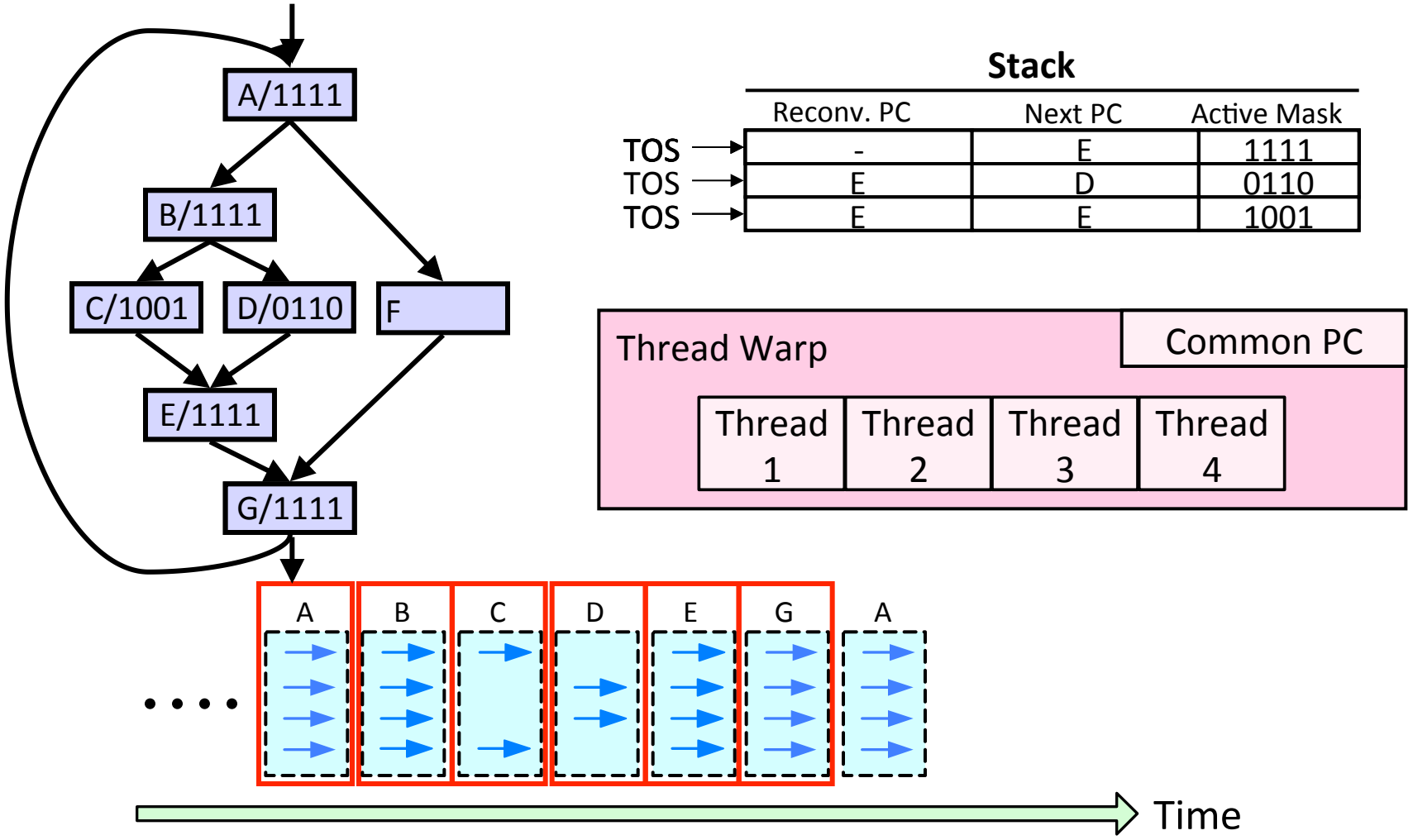
[Tim Rogers]

Two Routes to "Better"



Research Direction 1:
Mitigating SIMT Control Divergence

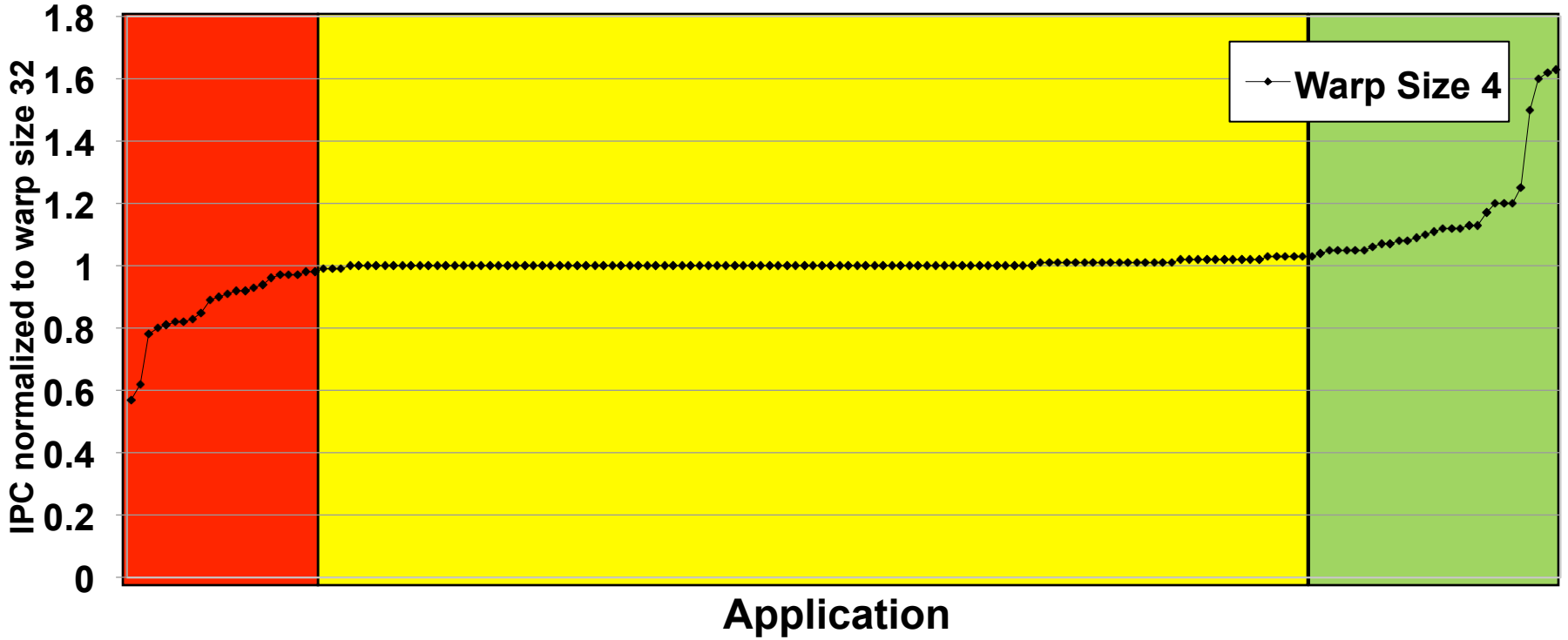
Recall: SIMT Hardware Stack



Potential for significant loss of throughput when control flow diverged!

Performance vs. Warp Size

- 165 Applications



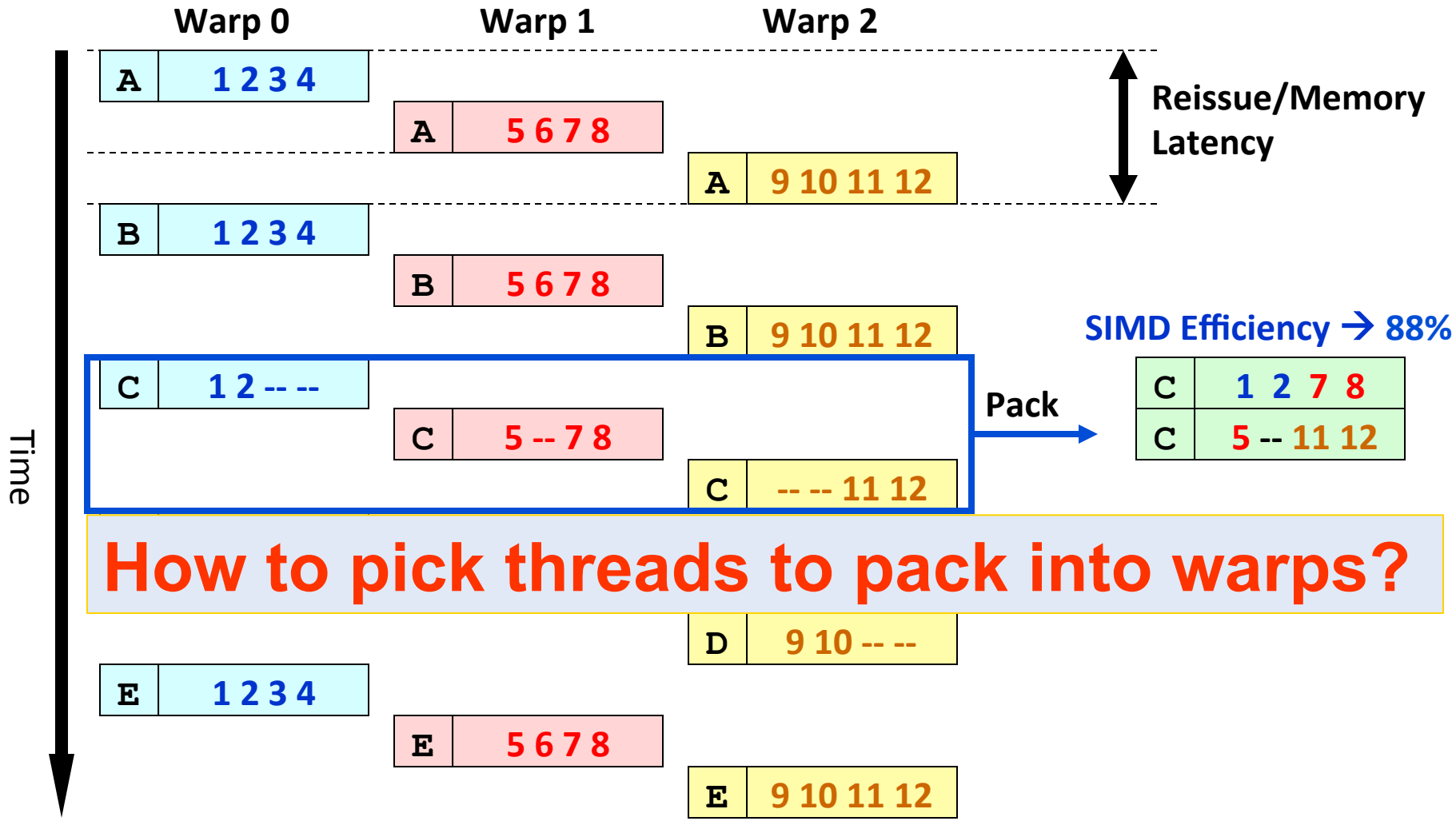
Convergent Applications

Warp-Size Insensitive Applications

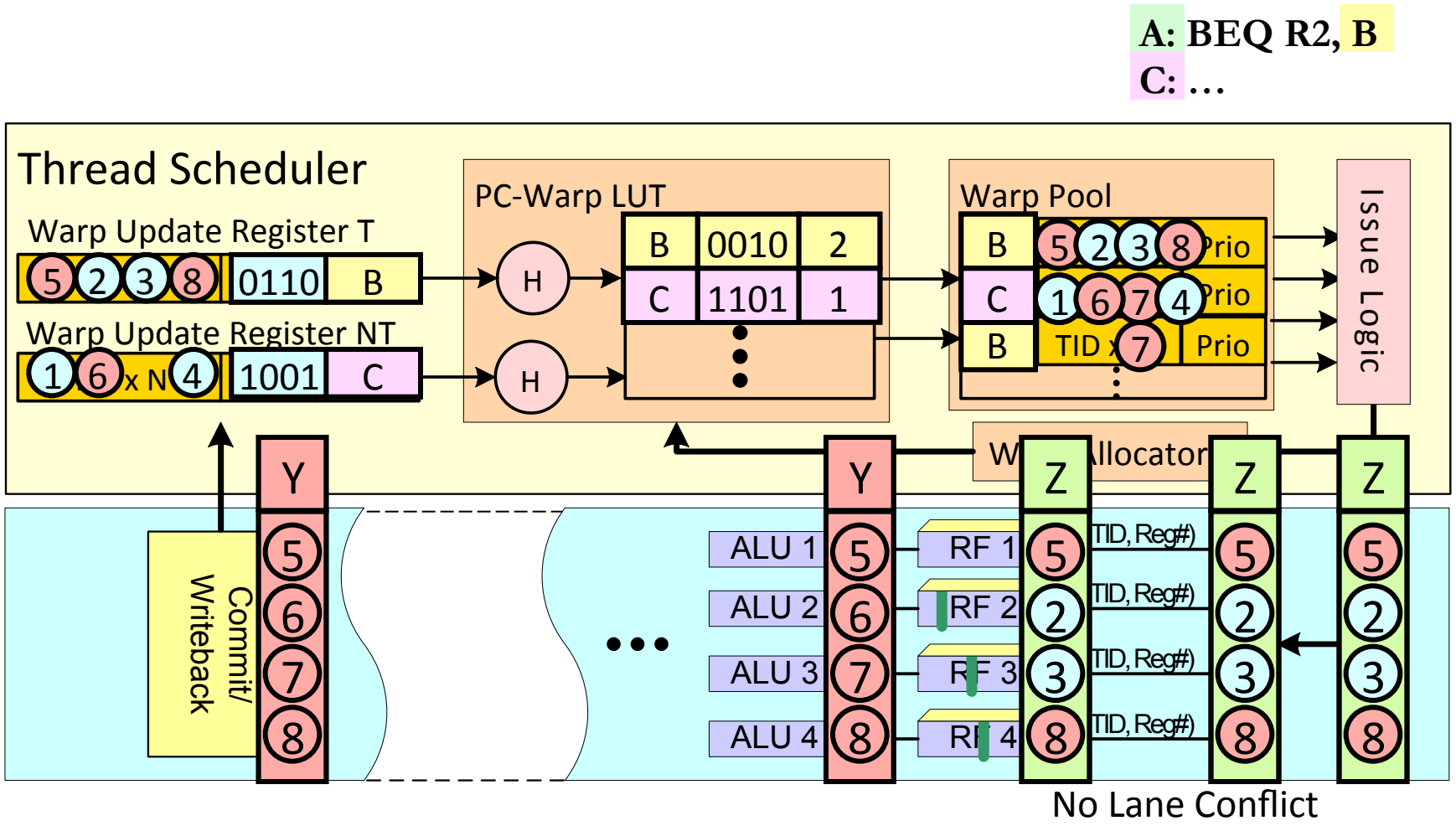
Divergent Applications

Dynamic Warp Formation

(Fung MICRO'07)



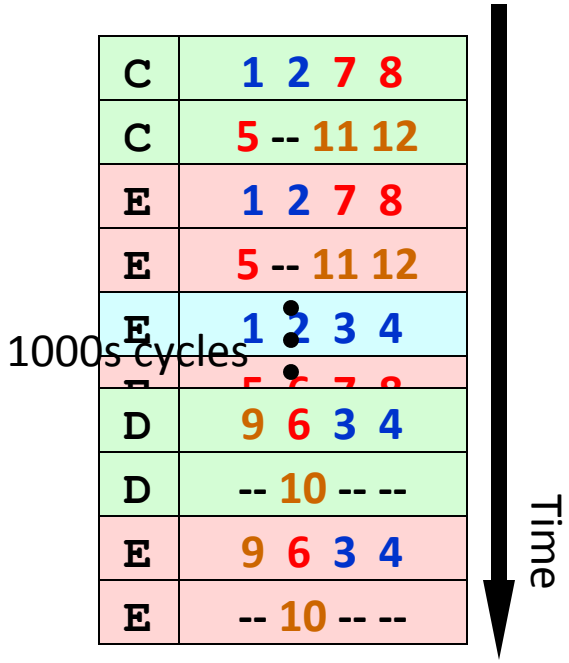
Dynamic Warp Formation: Hardware Implementation



DWF Pathologies: Starvation

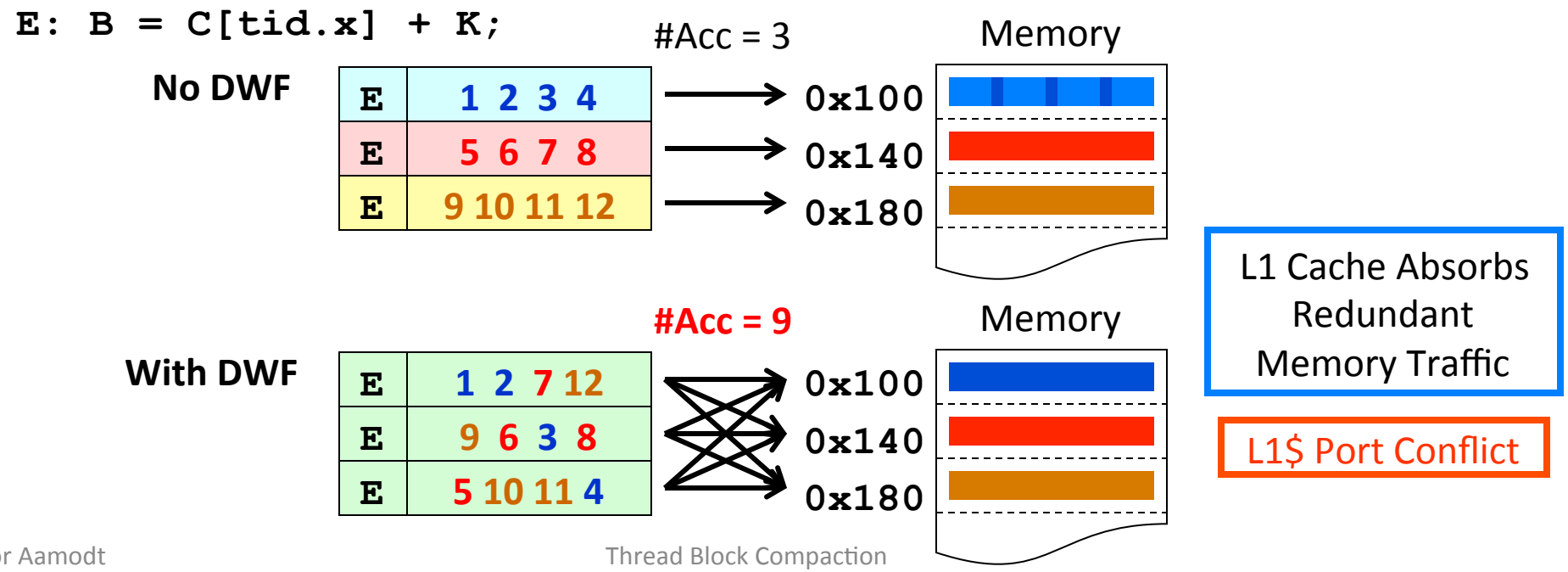
- Majority Scheduling
  - Best Performing
  - Prioritize largest group of threads with same PC
- ***Starvation***
  - LOWER SIMD Efficiency!
- Other Warp Scheduler?
  - Tricky: Variable Memory Latency

```
B: if (K > 10)
C:    K = 10;
   else
D:    K = 0;
E: B = C[tid.x] + K;
```



DWF Pathologies: Extra Uncoalesced Accesses

- Coalesced Memory Access = Memory SIMD
  - 1st Order CUDA Programmer Optimization
- Not preserved by DWF



DWF Pathologies: Implicit Warp Sync.

- Some CUDA applications depend on the lockstep execution of "static warps"

Warp 0	Thread 0 ... 31
Warp 1	Thread 32 ... 63
Warp 2	Thread 64 ... 95

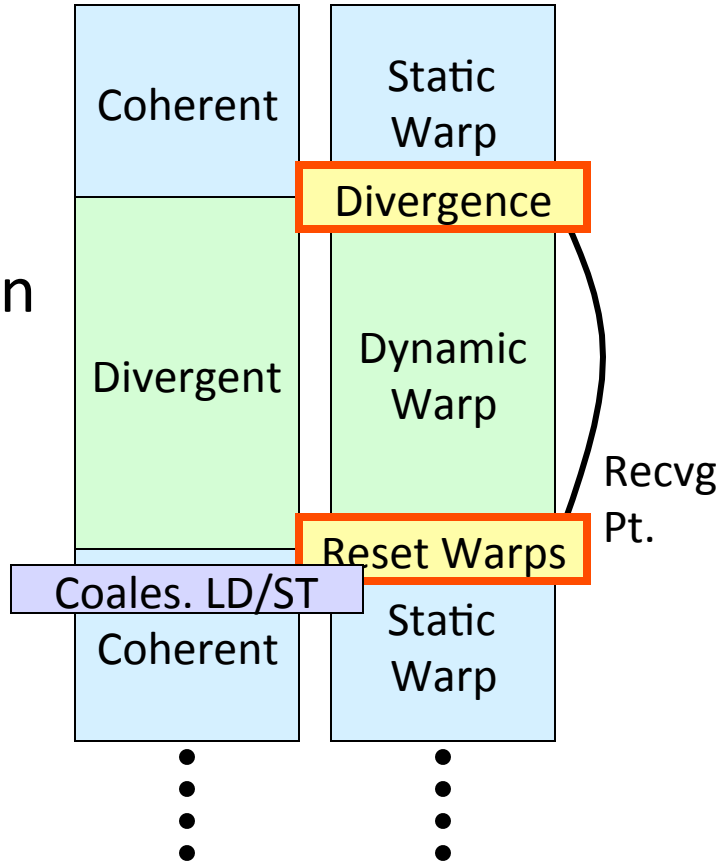
  - E.g. Task Queue in Ray Tracing

```
int wid = tid.x / 32;
if (tid.x % 32 == 0) {
  sharedTaskID[wid] = atomicAdd(g_TaskID, 32);
}
my_TaskID = sharedTaskID[wid] + tid.x % 32;
ProcessTask(my_TaskID);
```

Implicit Warp Sync.

Observation

- Compute kernels usually contain divergent and non-divergent (coherent) code segments
- Coalesced memory access usually in coherent code segments
  - DWF no benefit there



Thread Block Compaction

- Run a thread block like a warp
  - Whole block move between coherent/divergent code
  - Block-wide stack to track exec. paths reconvg.
- Barrier @ Branch/reconverge pt.
  - All avail. threads arrive at branch
  - Insensitive to warp scheduling
- Warp compaction
  - Regrouping with all avail. threads
  - If no divergence, gives static warp arrangement

✓ **Implicit Warp Sync.**

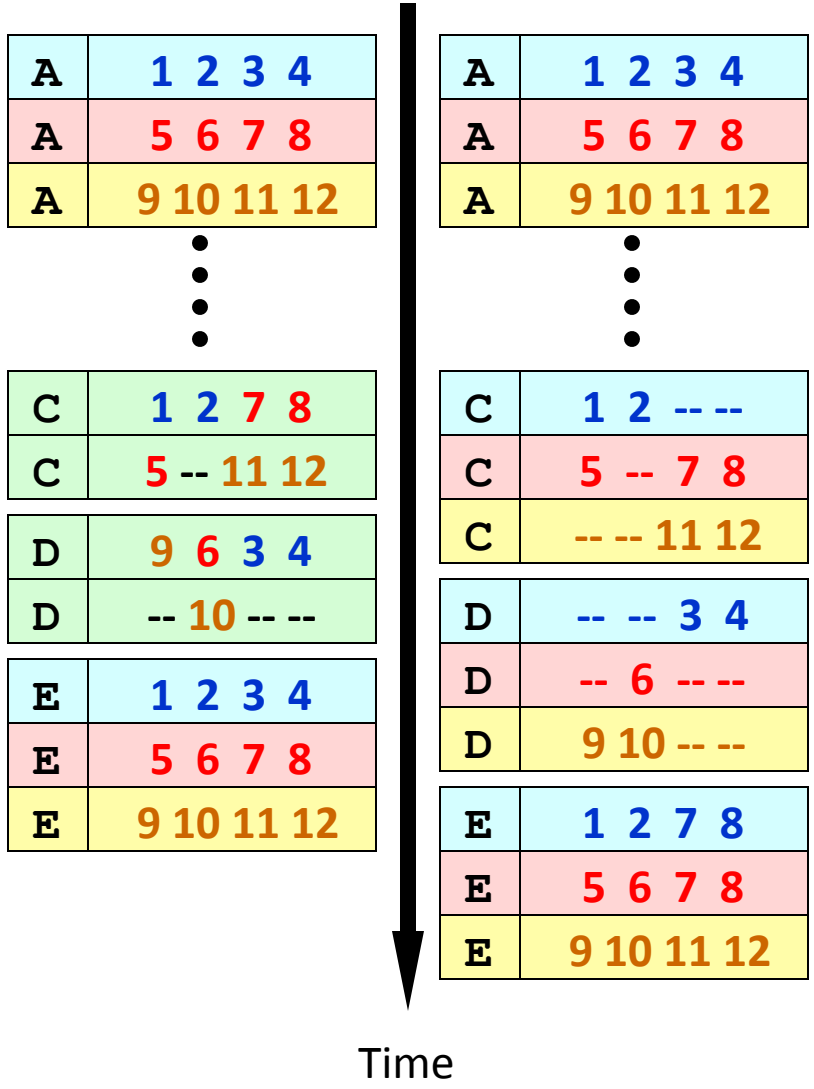
~~**Starvation**~~

~~**Extra Uncoalesced Memory Access**~~

Thread Block Compaction

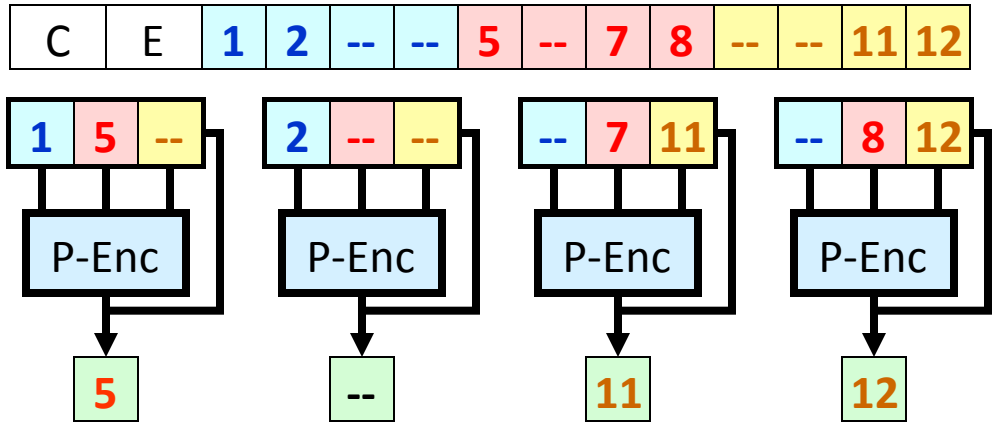
PC	RPC	Active Threads											
		1	2	3	4	5	6	7	8	9	10	11	12
E	-	1	2	3	4	5	6	7	8	9	10	11	12
D	E	--	--	--	--	--	--	--	--	--	--	--	--
C	E	--	--	--	--	--	--	--	--	--	--	--	--

```
A: K = A[tid.x];
B: if (K > 10)
C:     K = 10;
   else
D:     K = 0;
E: B = C[tid.x] + K;
```



Thread Compactor

- Convert *activemask* from block-wide stack to *thread IDs* in warp buffer
- Array of Priority-Encoder

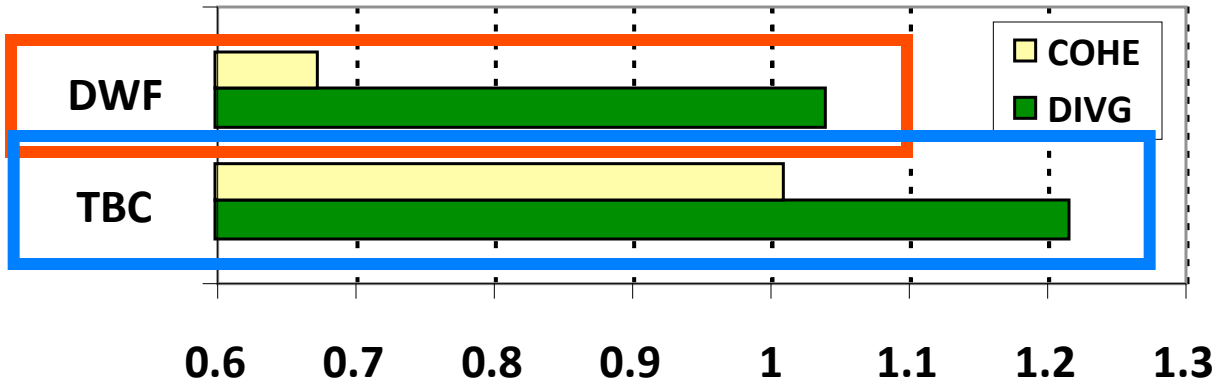


Warp Buffer

C	1 2 7 8
C	5 -- 11 12

Experimental Results

- 2 Benchmark Groups:
  - COHE = Non-Divergent CUDA applications
  - DIVG = Divergent CUDA applications



Serious Slowdown from pathologies

No Penalty for COHE

22% Speedup on DIVG

Per-Warp Stack

Recent work on warp divergence

- Intel [MICRO 2011]: Thread Frontiers – early reconvergence for unstructured control flow.
- UT-Austin/NVIDIA [MICRO 2011]: Large Warps – similar to TBC except decouple size of thread stack from thread block size.
- NVIDIA [ISCA 2012]: Simultaneous branch and warp interweaving. Enable SIMD to execute two paths at once.
- Intel [ISCA 2013]: Intra-warp compaction – extends Xeon Phi uarch to enable compaction.
- NVIDIA: Temporal SIMT [described briefly in IEEE Micro article and in more detail in CGO 2013 paper]
- NVIDIA [ISCA 2015]: Variable Warp-Size Architecture – merge small warps (4 threads) into "gangs".

Thread Frontiers [Diamos et al., MICRO 2011]

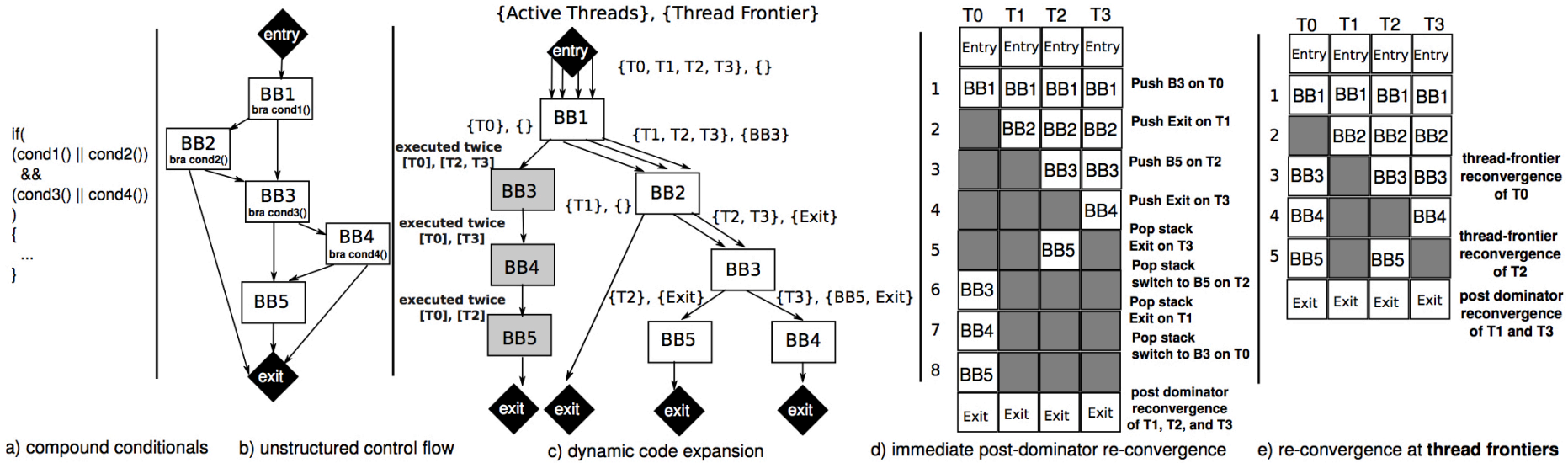
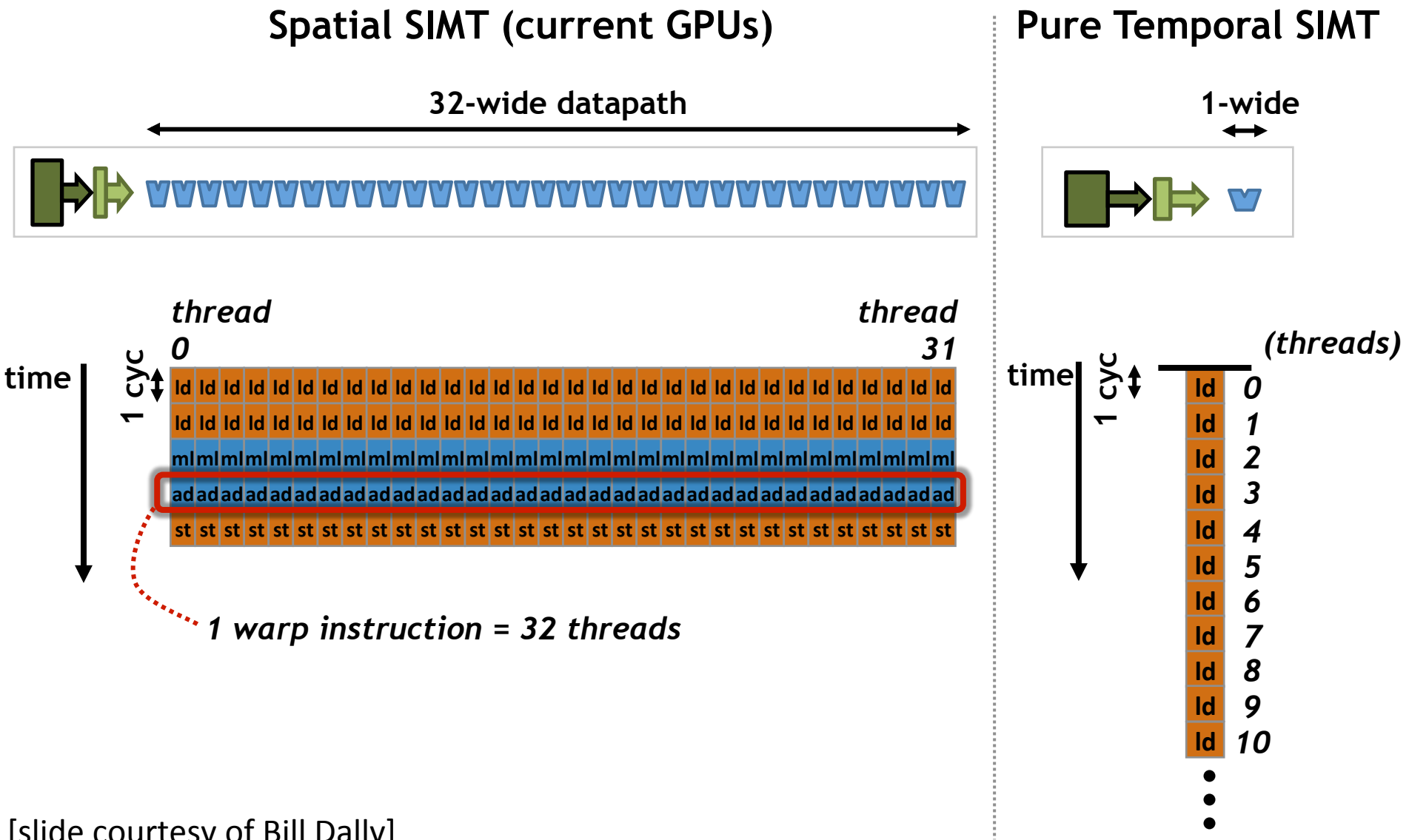


Figure 1: An example of an application with unstructured control flow leading to dynamic code expansion.

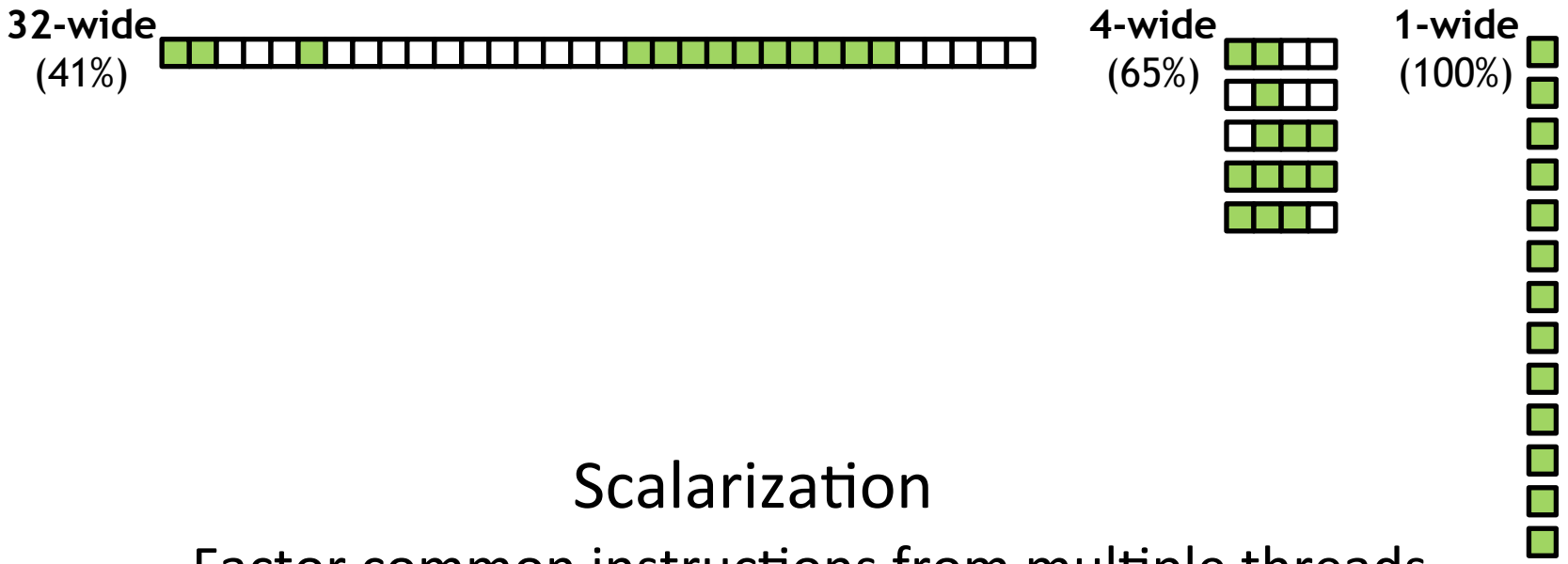
Temporal SIMT



[slide courtesy of Bill Dally]

Temporal SIMT Optimizations

Control divergence — hybrid MIMD/SIMT



Scalarization

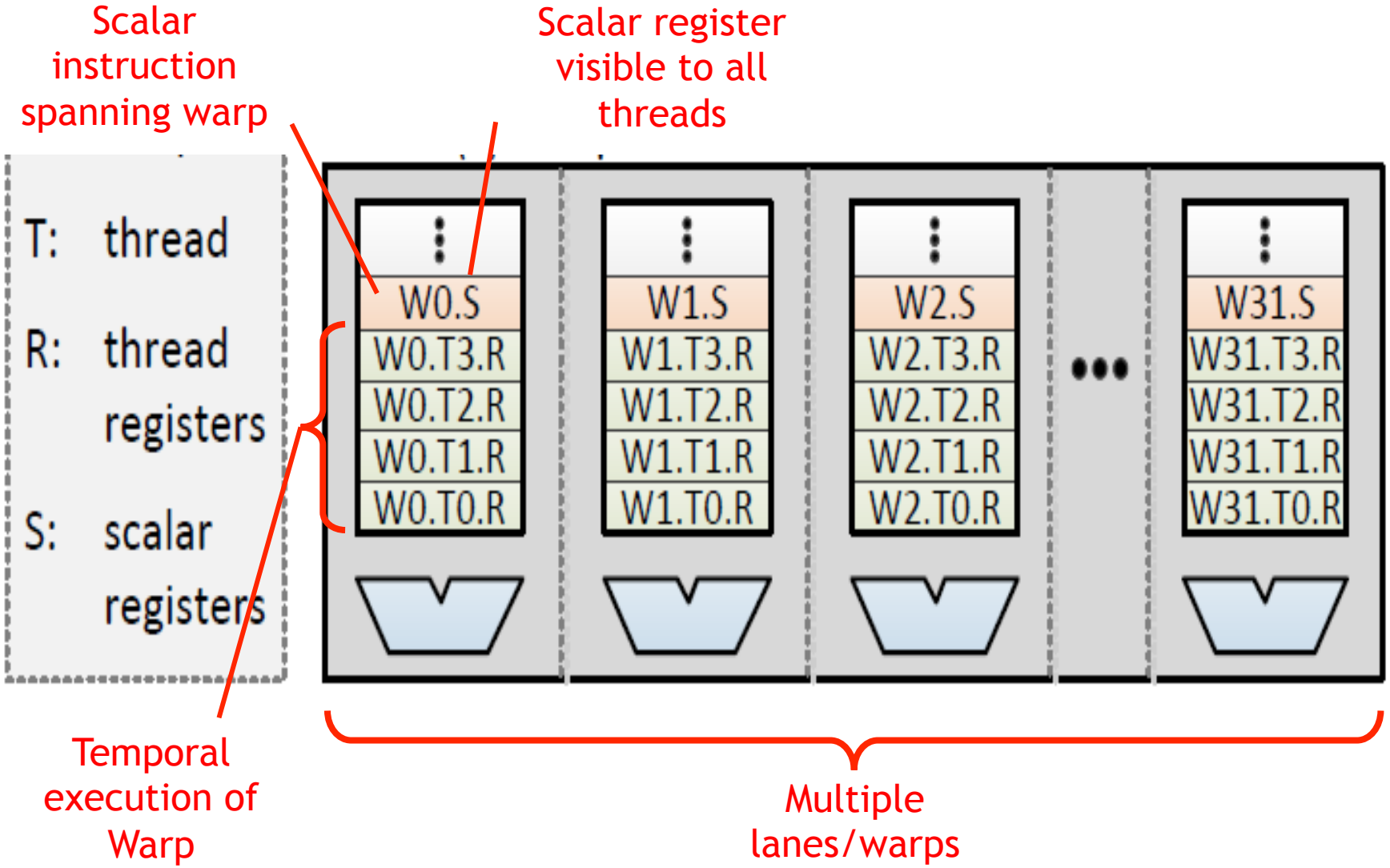
Factor common instructions from multiple threads

Execute once – place results in common registers

[See: SIMT Affine Value Structure (ISCA 2013)]

[slide courtesy of Bill Dally]

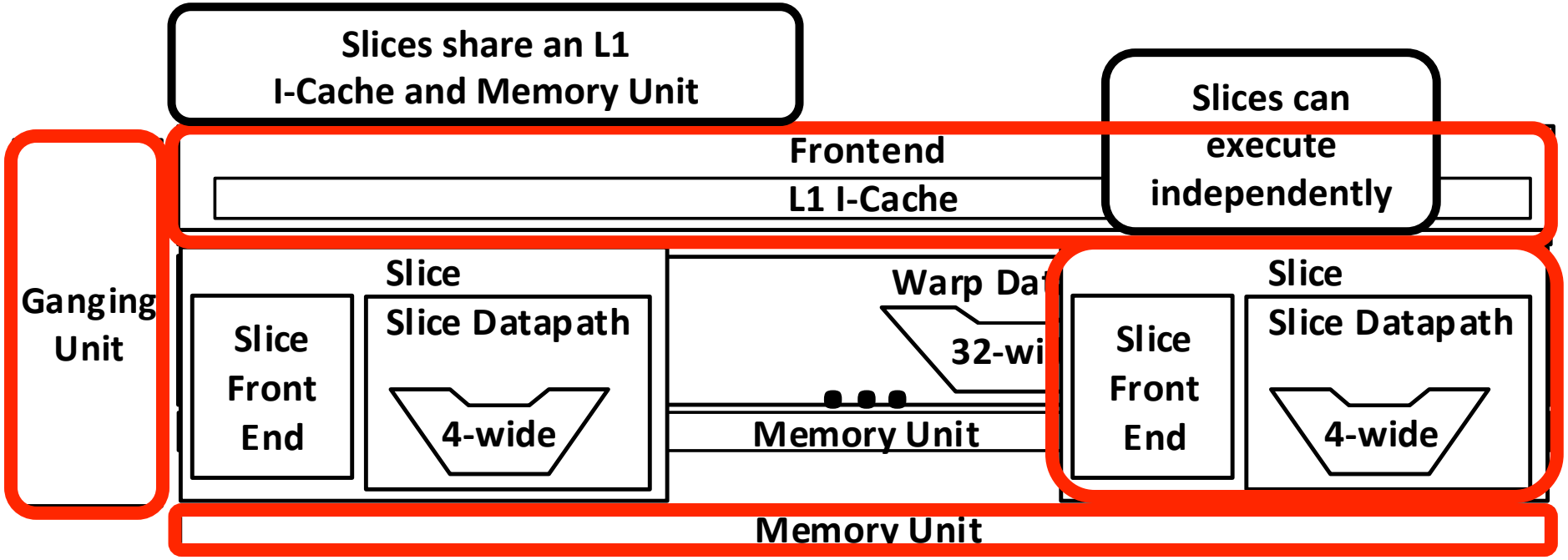
Scalar Instructions in SIMT Lanes



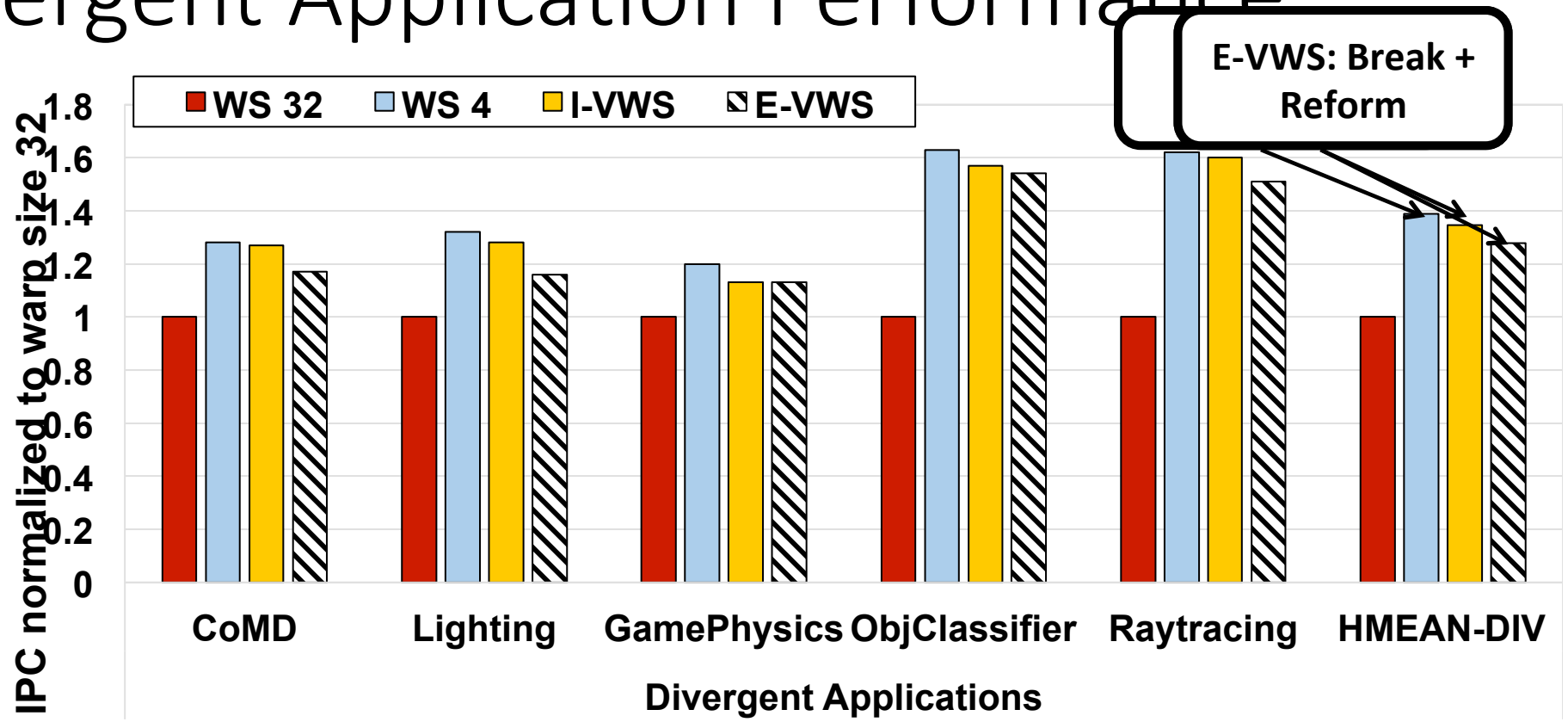
[slide courtesy of Bill Dally]

Variable Warp-Size Architecture

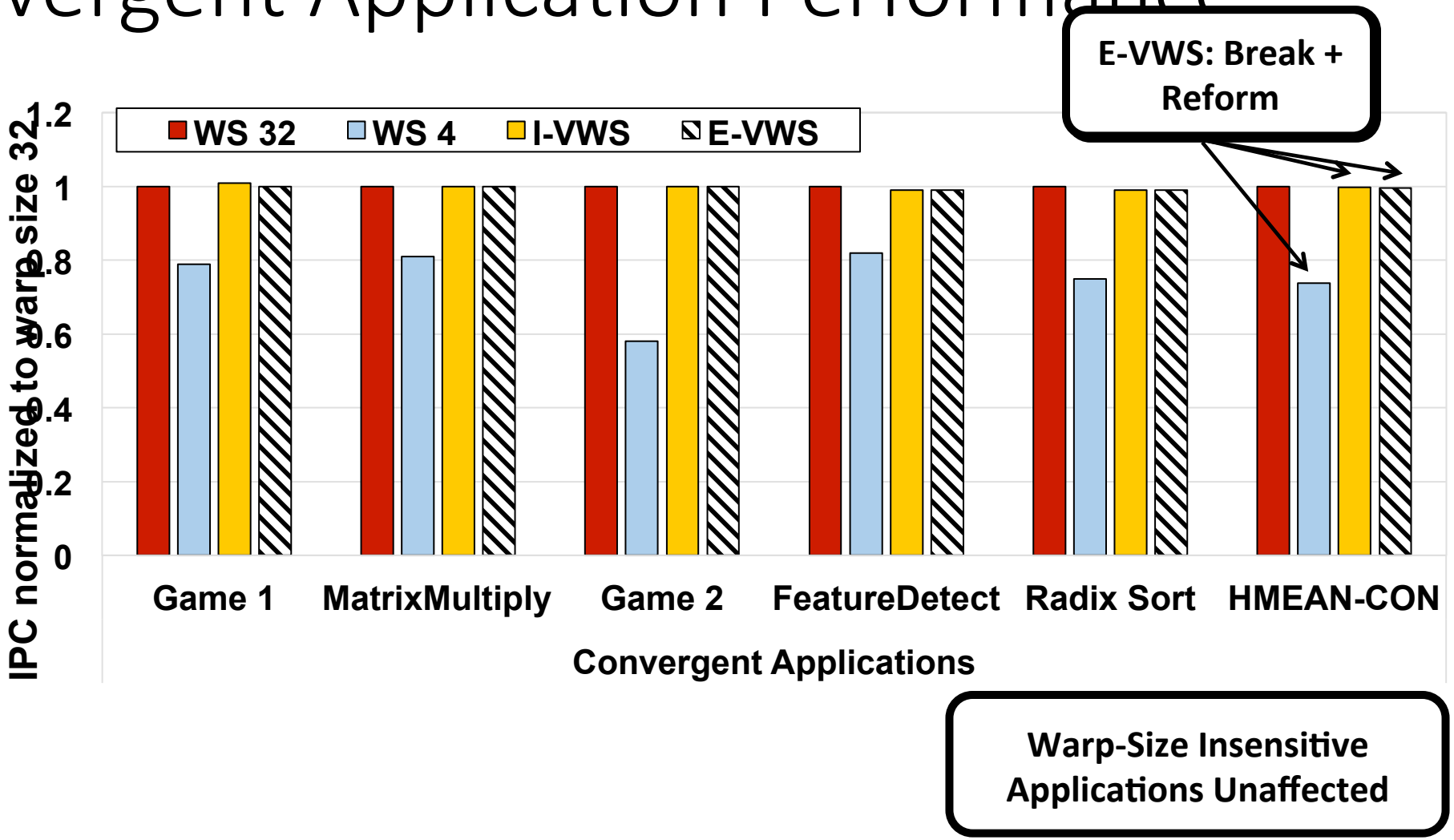
- Most recent work by NVIDIA [ISCA 2015]
- Split the SM datapath into narrow **slices**.
  - Extensively studied 4-thread slices
- Gang slice execution to gain efficiencies of wider warp.



Divergent Application Performance



Convergent Application Performance



Research Direction 2: Mitigating High GPGPU Memory Bandwidth Demands

Reducing Off-Chip Access / Divergence

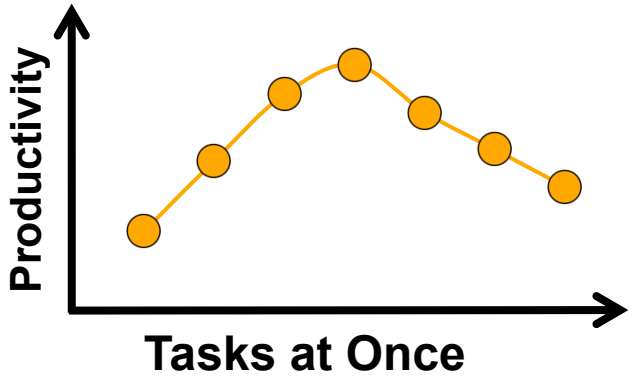
- Re-writing software to use “shared memory” and avoid uncoalesced global accesses is bane of GPU programmer existence.
- Recent GPUs introduce caches, but large number of warps/ wavefronts lead to thrashing.

- NVIDIA: Register file cache (ISCA 2011, MICRO)
  - Register file burns significant energy
  - Many values read once soon after written
  - Small register file cache captures locality and saves energy but does not help performance
  - Recent follow on work from academia
- Prefetching (Kim, MICRO 2010)
- Interconnect (Bakhoda, MICRO 2010)
- Lee & Kim (HPCA 2012) CPU/GPU cache sharing

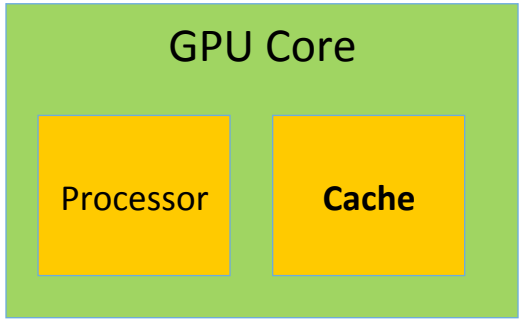
Thread Scheduling Analogy

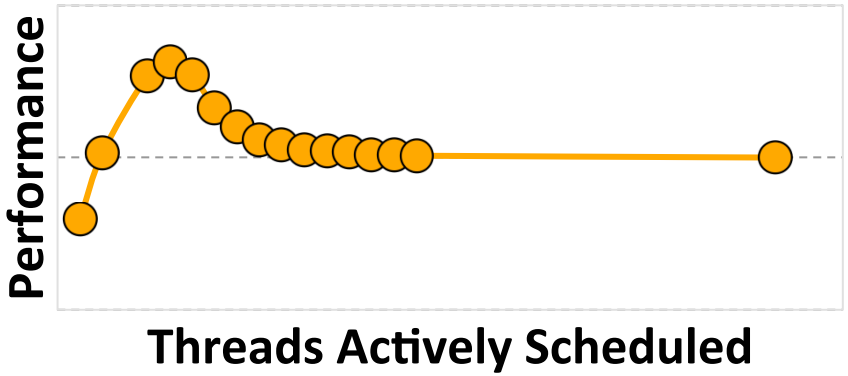
[MICRO 2012]

- Human Multitasking
  - Humans have limited **attention capacity**



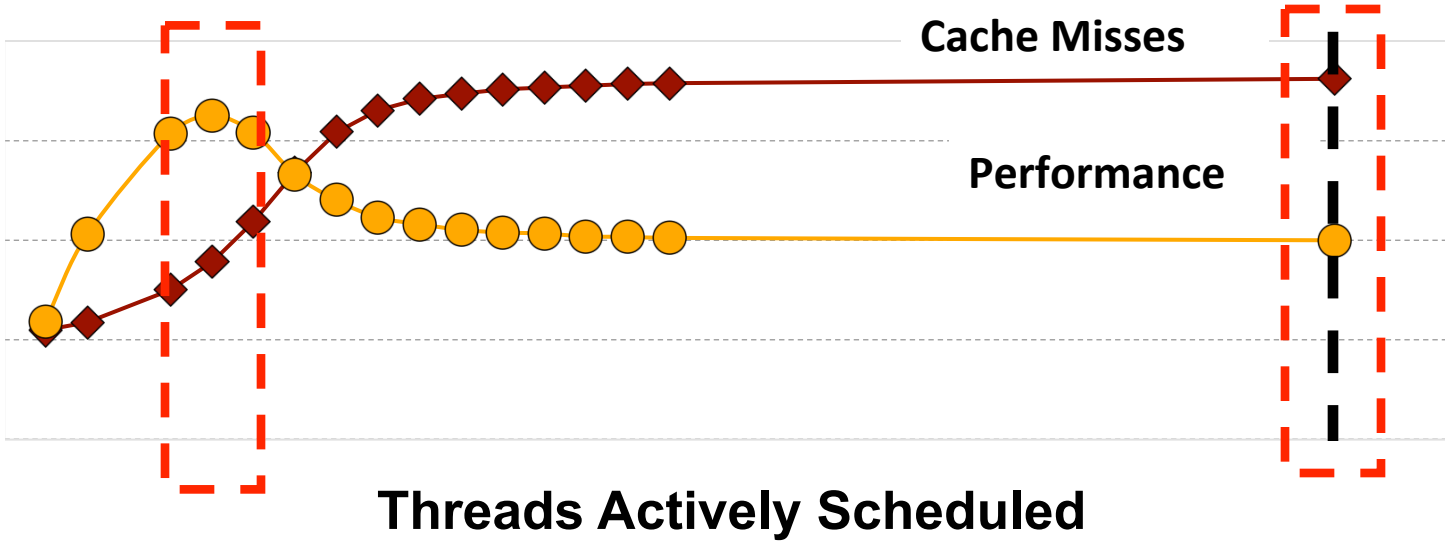
  - GPUs have limited **cache capacity**

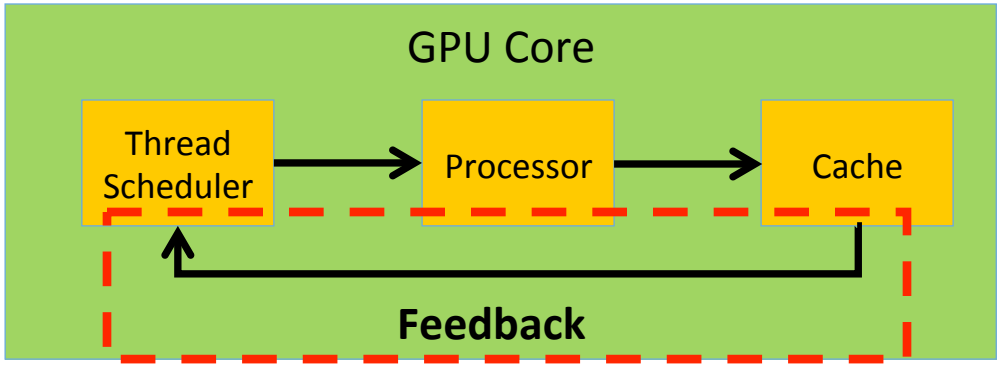




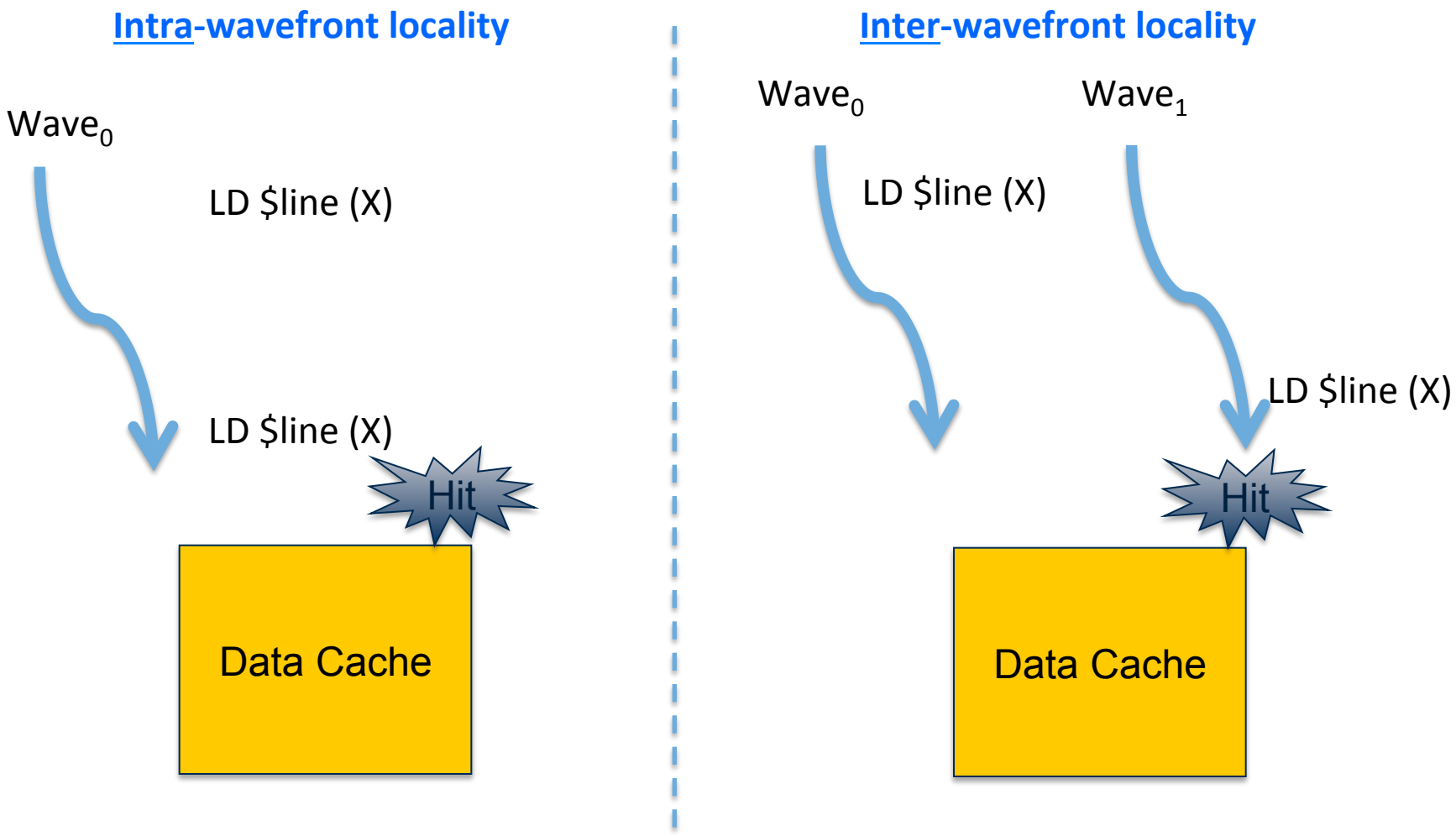
Use Memory System Feedback

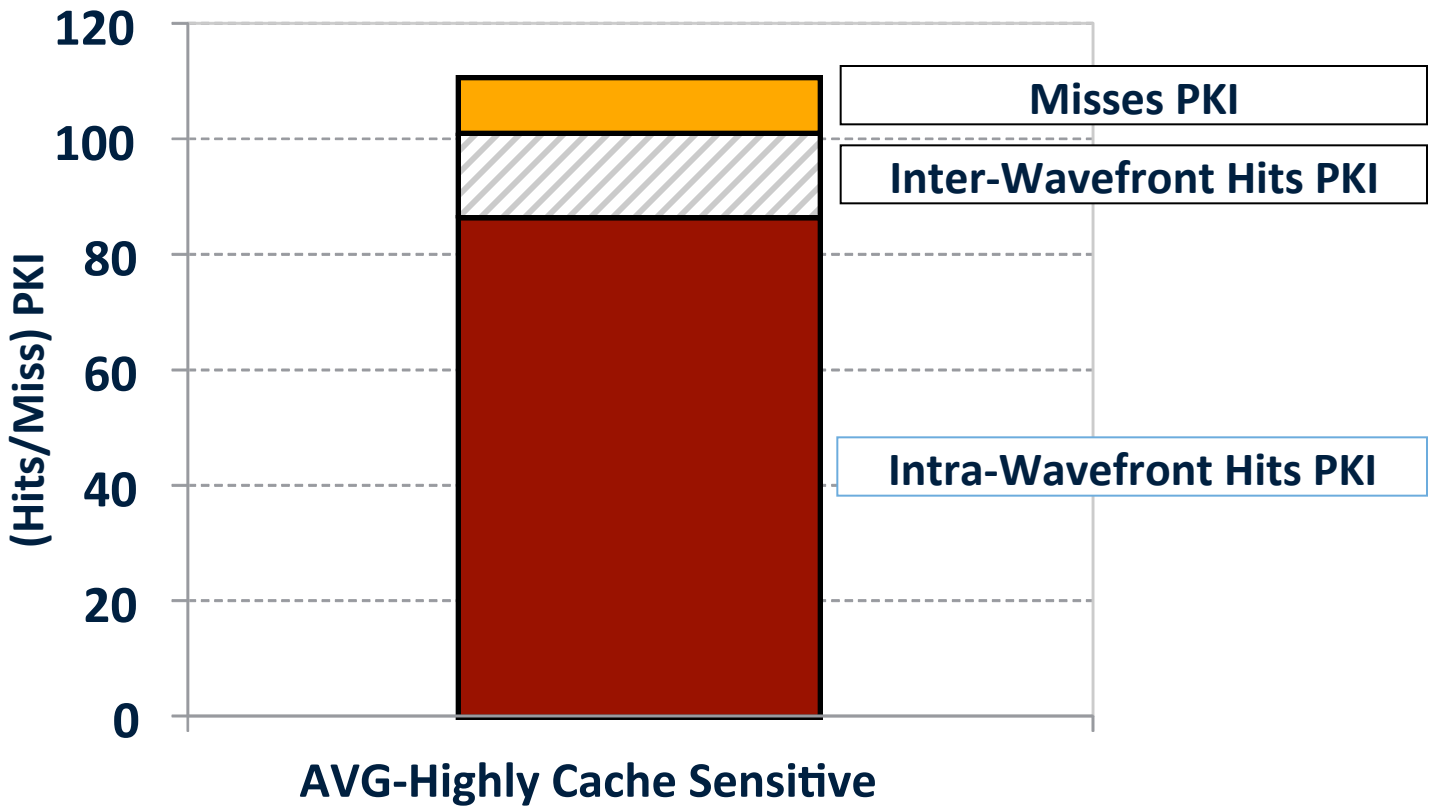
[MICRO 2012]



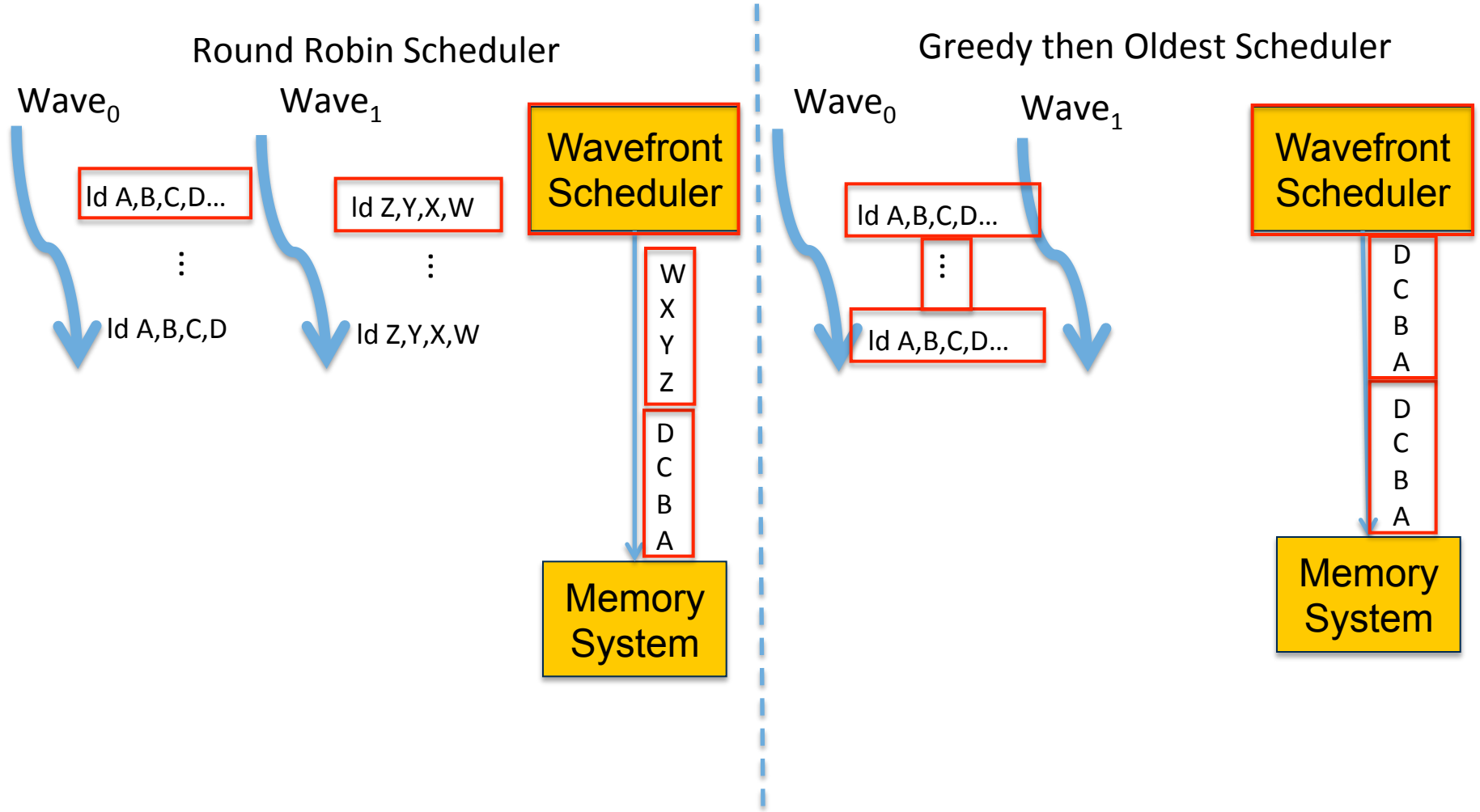


Sources of Locality

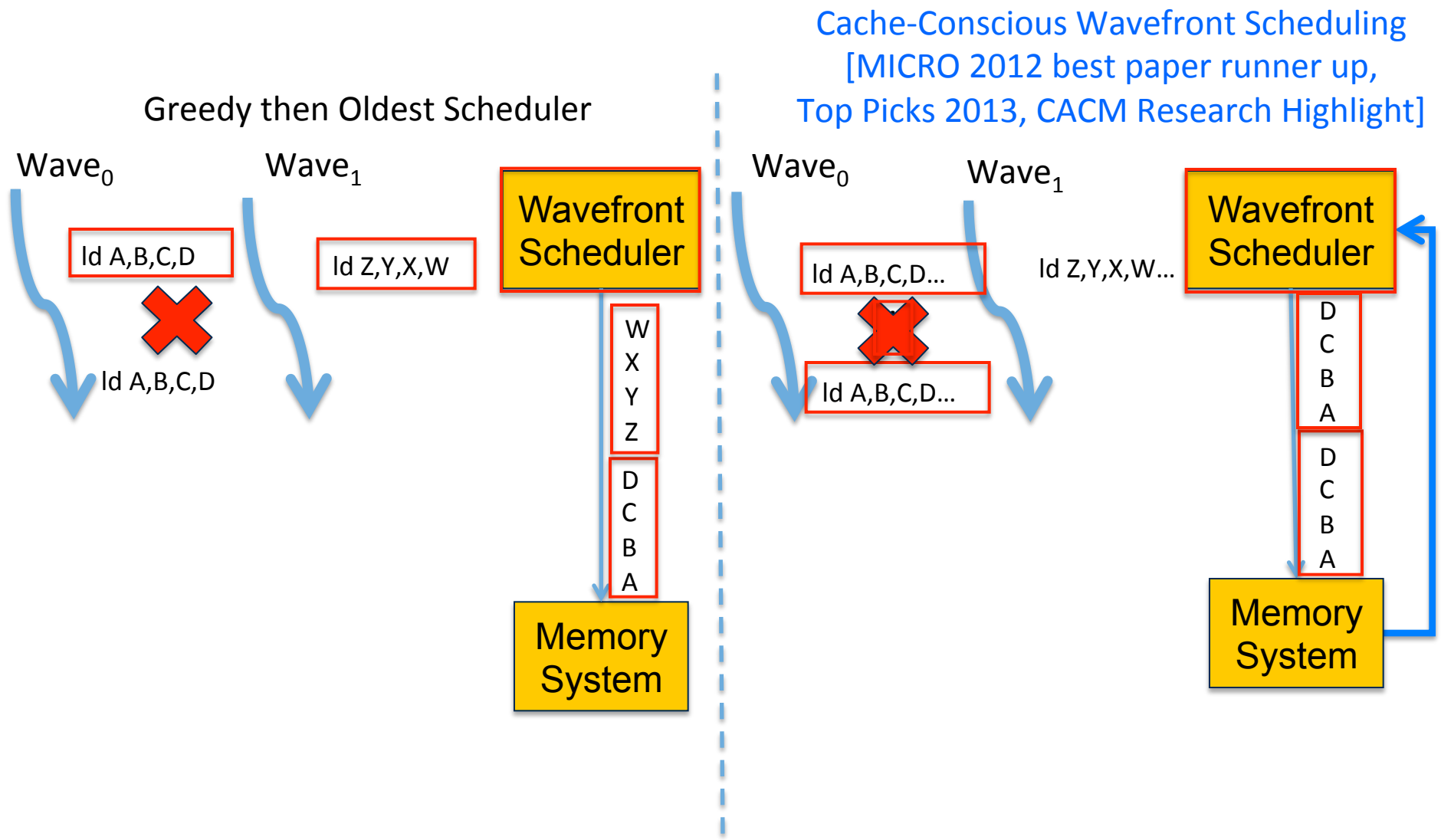


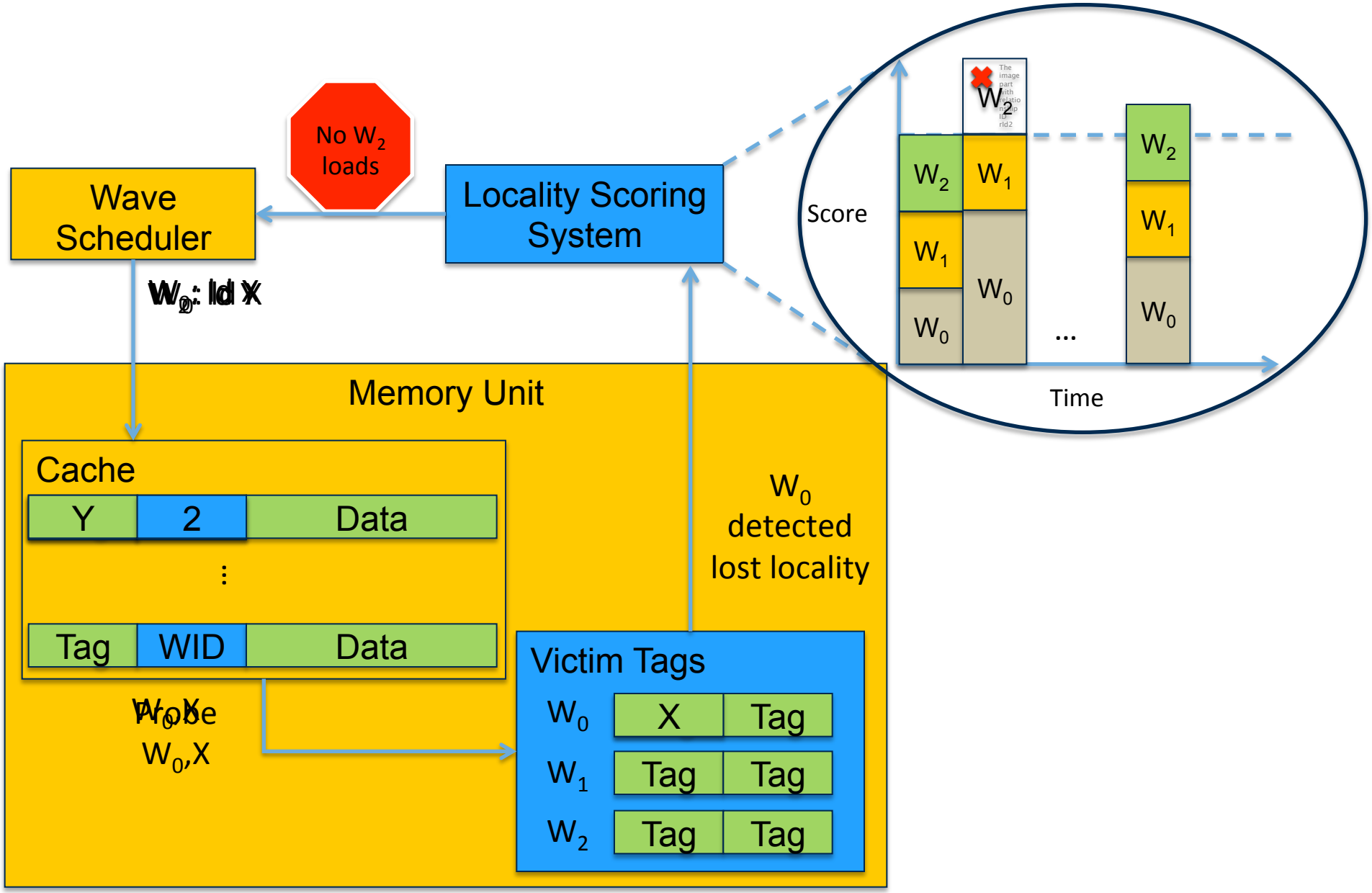

120
100
80
60
40
20
0
(Hits/Miss) PKI
Misses PKI
Inter-Wavefront Hits PKI
Intra-Wavefront Hits PKI
AVG-Highly Cache Sensitive

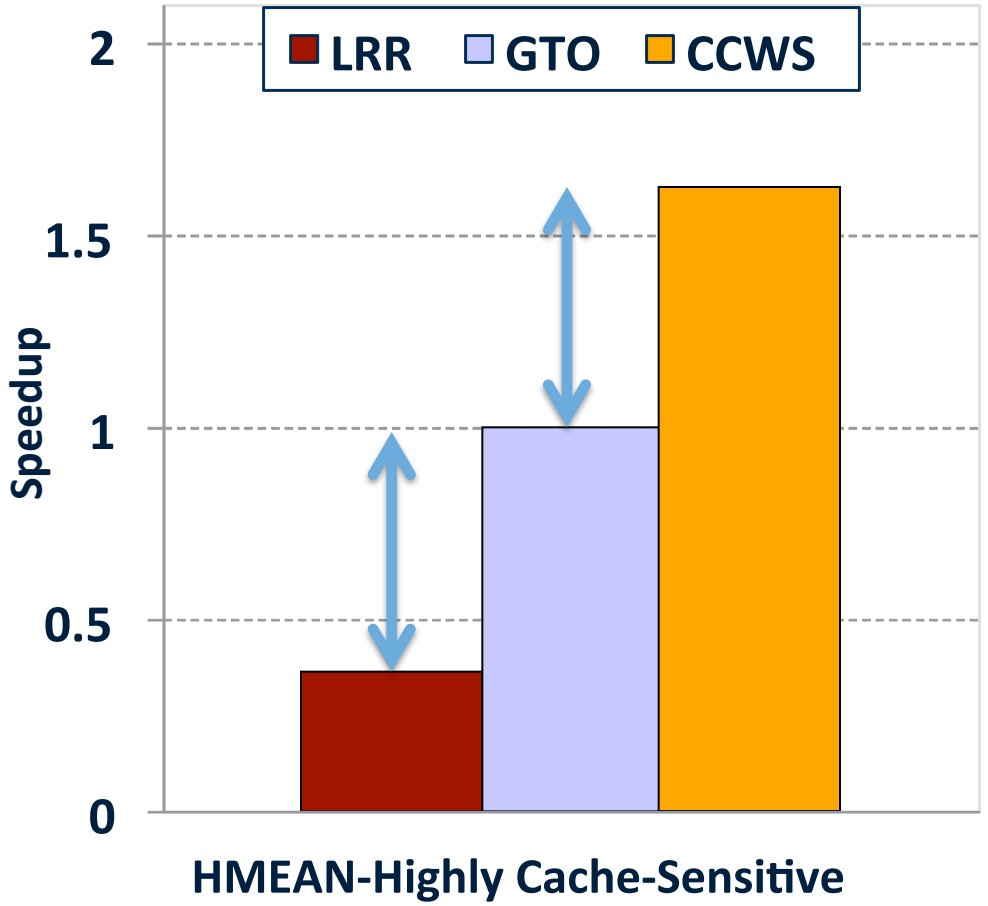
Scheduler affects access pattern



Use scheduler to *shape* access pattern




No W_2 loads
Wave Scheduler
Locality Scoring System
Score
Time
W_2
W_1
W_0
...
Memory Unit
Cache
Y
2
Data
Tag
WID
Data
W_0 detected lost locality
Victim Tags
W_0
X
Tag
W_1
Tag
Tag
W_2
Tag
Tag


LRR
GTO
CCWS
2
1.5
1
0.5
0
Speedup
HMEAN-Highly Cache-Sensitive

Static Wavefront Limiting
[Rogers et al., MICRO 2012]

- Profiling an application we can find an optimal number of wavefronts to execute
- Does a little better than CCWS.
- Limitations: Requires profiling, input dependent, does not exploit phase behavior.

Improve upon CCWS?

- CCWS detects bad scheduling decisions and avoids them in future.

- Would be better if we could “think ahead” / “be proactive” instead of “being reactive”

Programmability case study [MICRO 2013]

Sparse Vector-Matrix Multiply

GPU-Optimized Version
SHOC Benchmark Suite
(Oakridge National Labs)

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;
    int myRow = (blockIdx.x * warpsPerBlock)

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

                                miters[myRow];
                                ters[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)
                                tialSums[t+ 2];
                                tialSums[t+ 1];


                                Sums[t];
```

Explicit Scratchpad Use

Dependent on Warp Size

Added Complication

Parallel Reduction

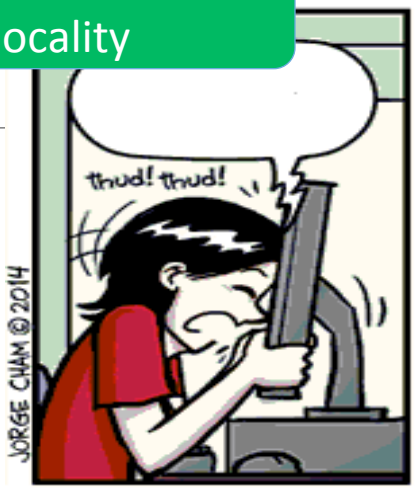
Simple Version

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 * bl
        + 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] * vecTe

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

Divergence

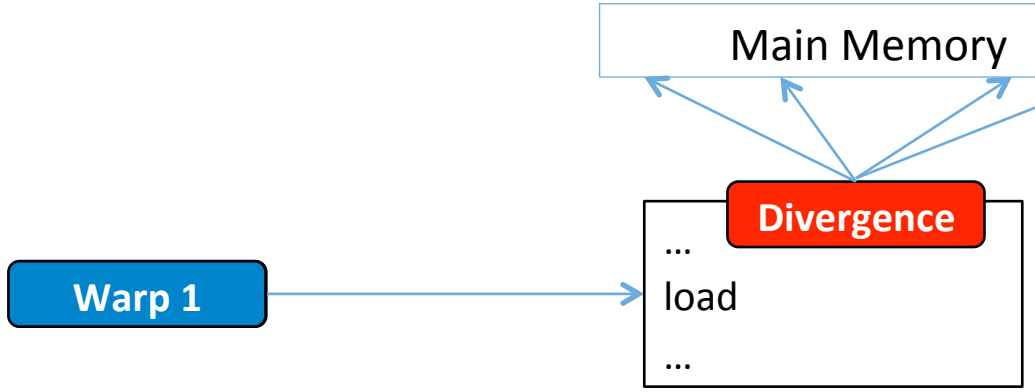
Each thread has locality



Using DAWS scheduling within 4% of optimized with no programmer input

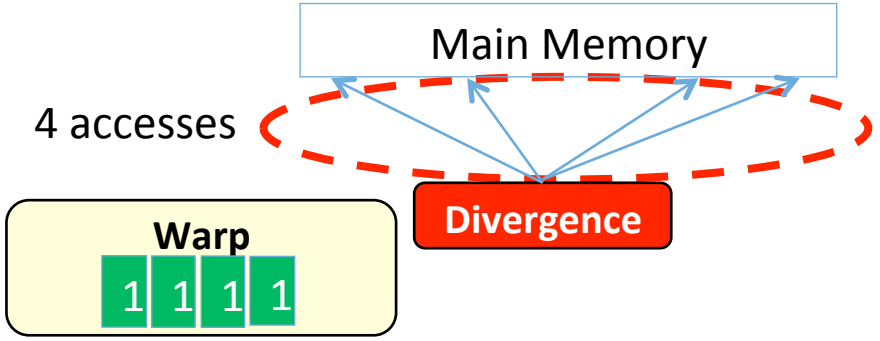
Observations
[Rogers et al., MICRO 2013]

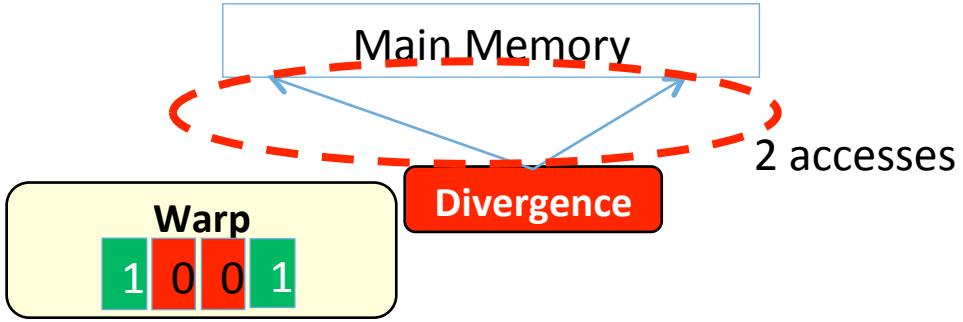
- Memory divergence in static instructions is predictable



- Data touched by divergent loads dependent on active mask

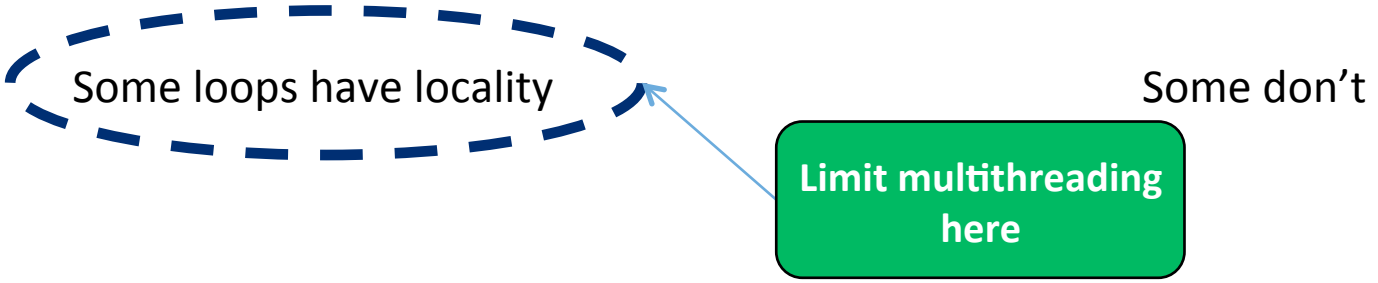
Both Used To Create Cache Footprint Prediction



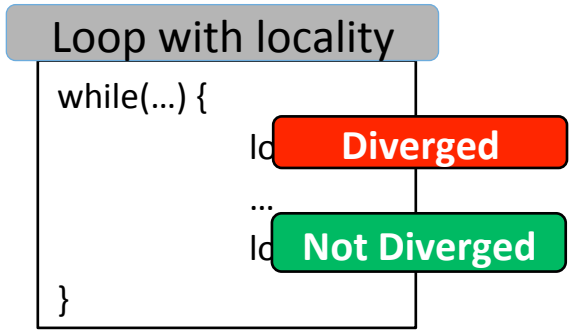


Footprint Prediction

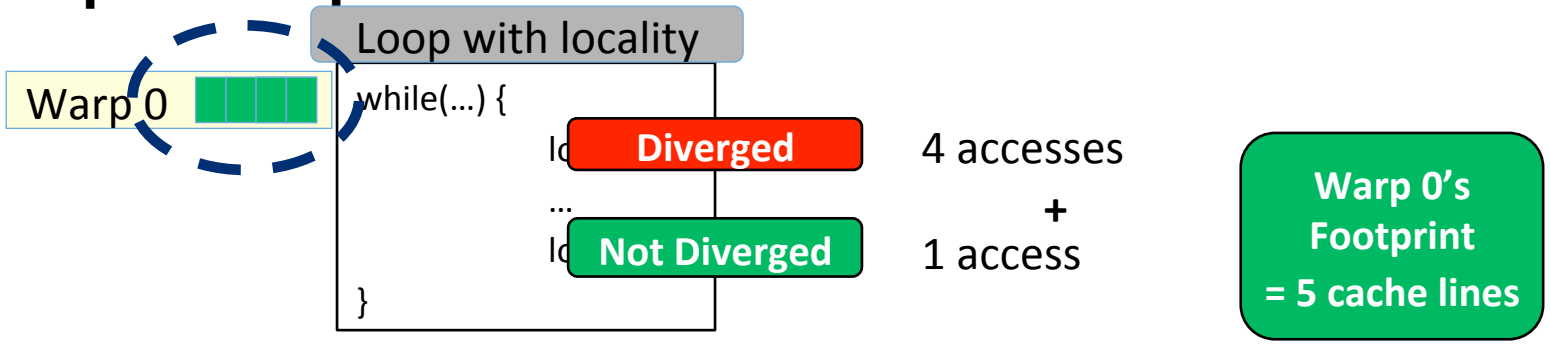
1. Detect loops with locality



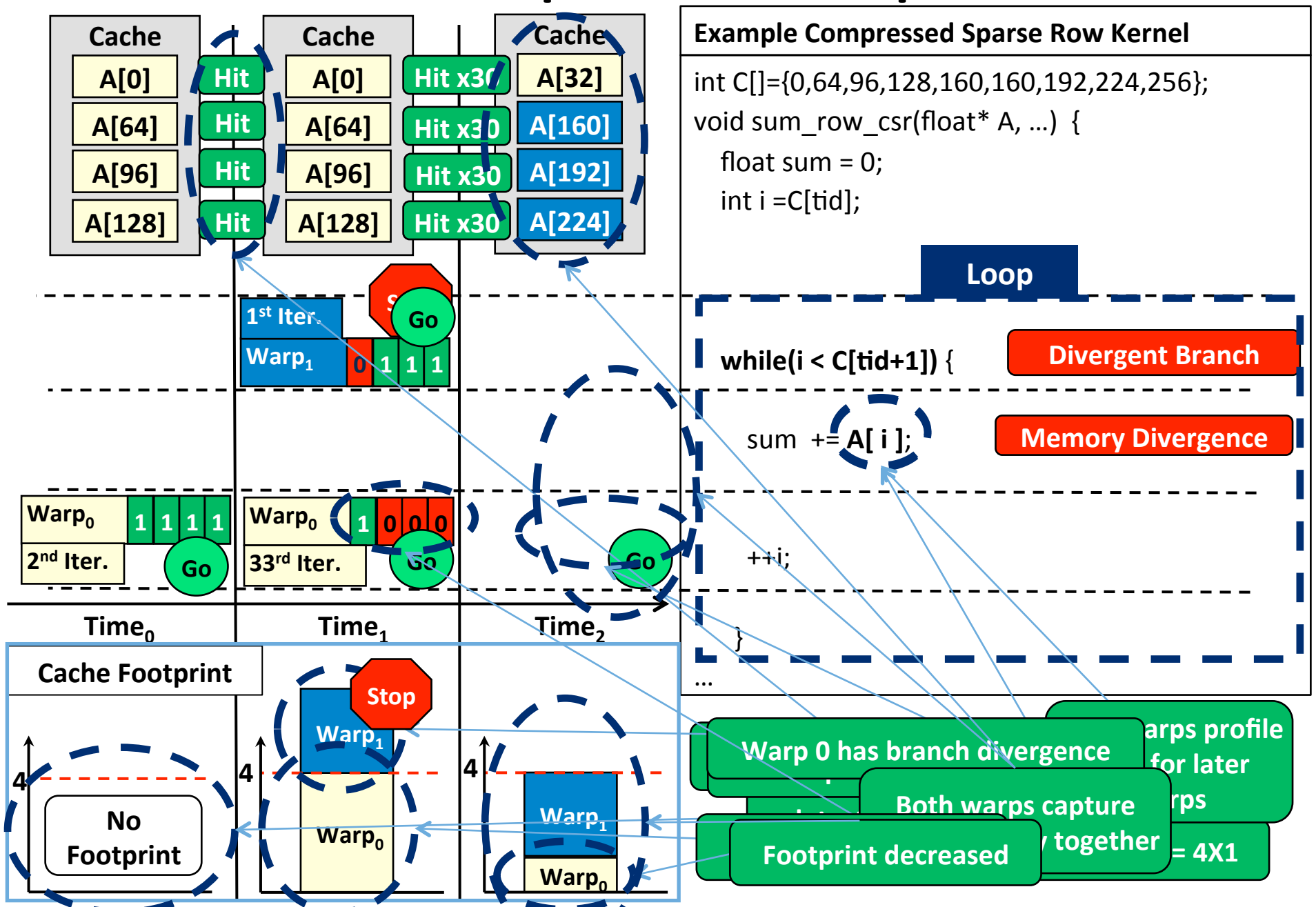
2. Classify loads in the loop



3. Compute footprint from active mask

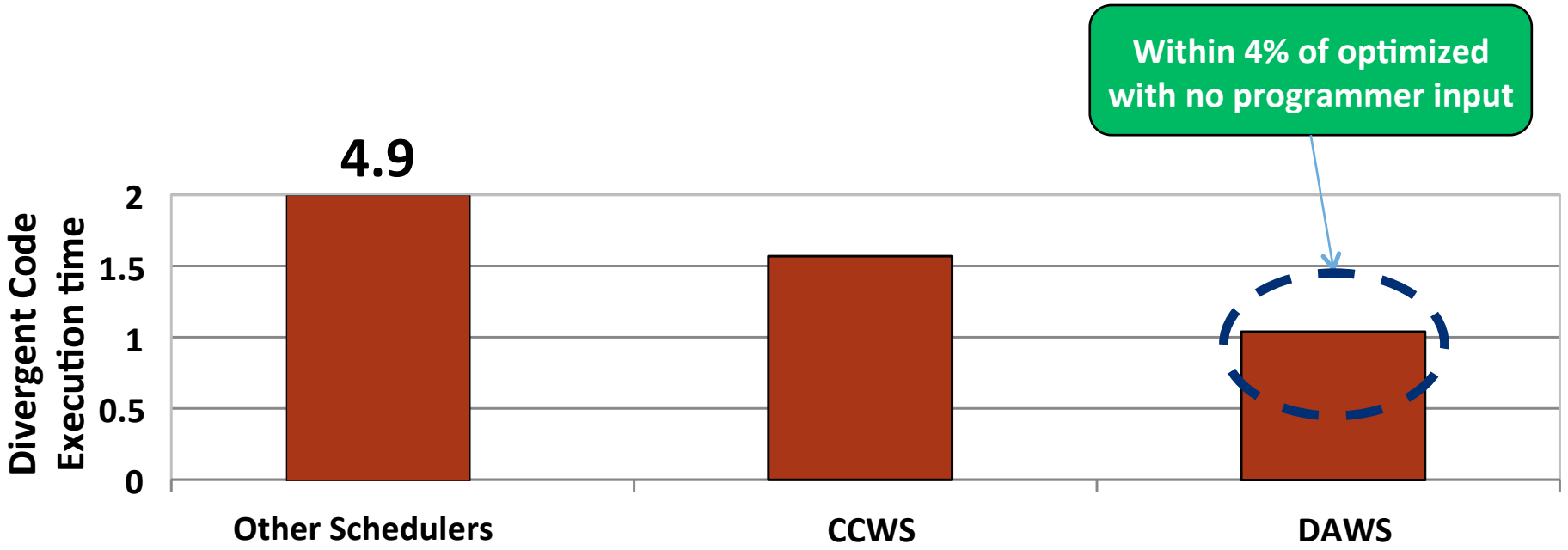


DAWS Operation Example

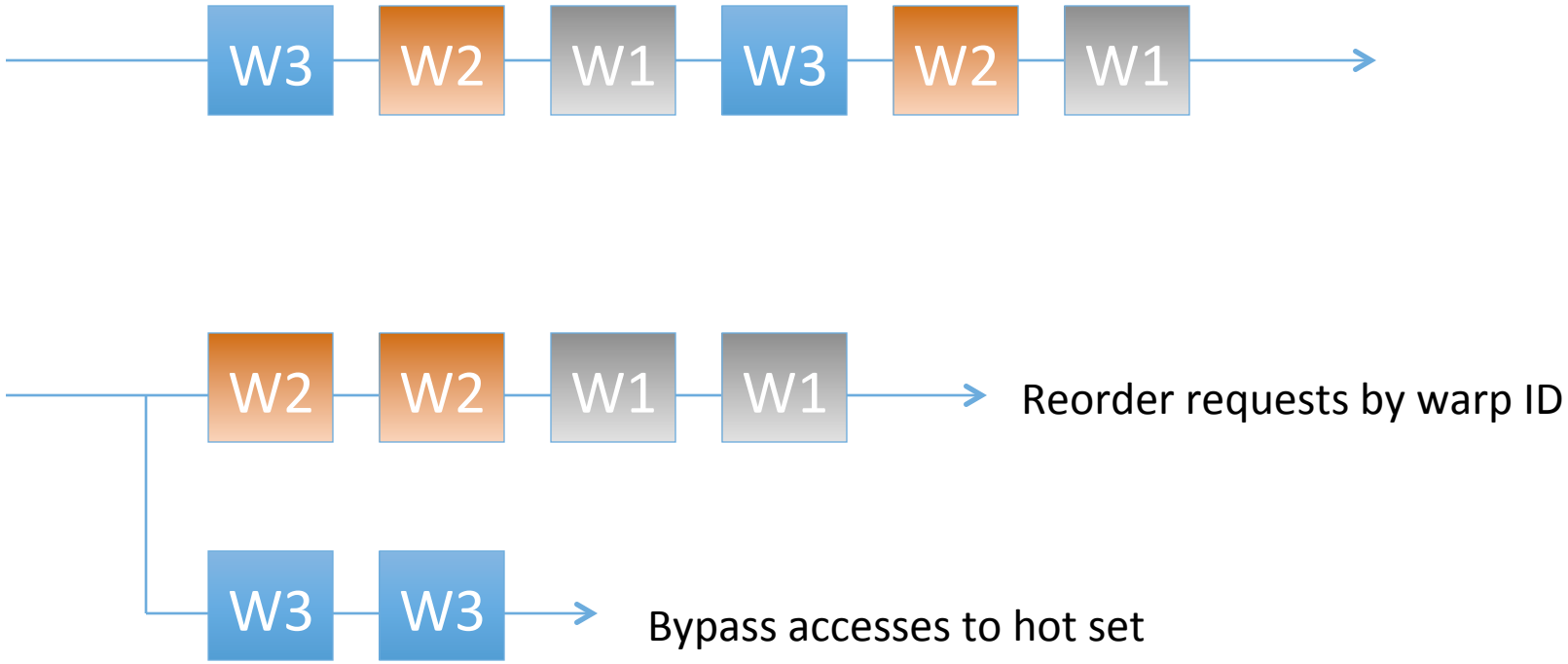


Sparse MM Case Study Results

- Performance (normalized to optimized version)



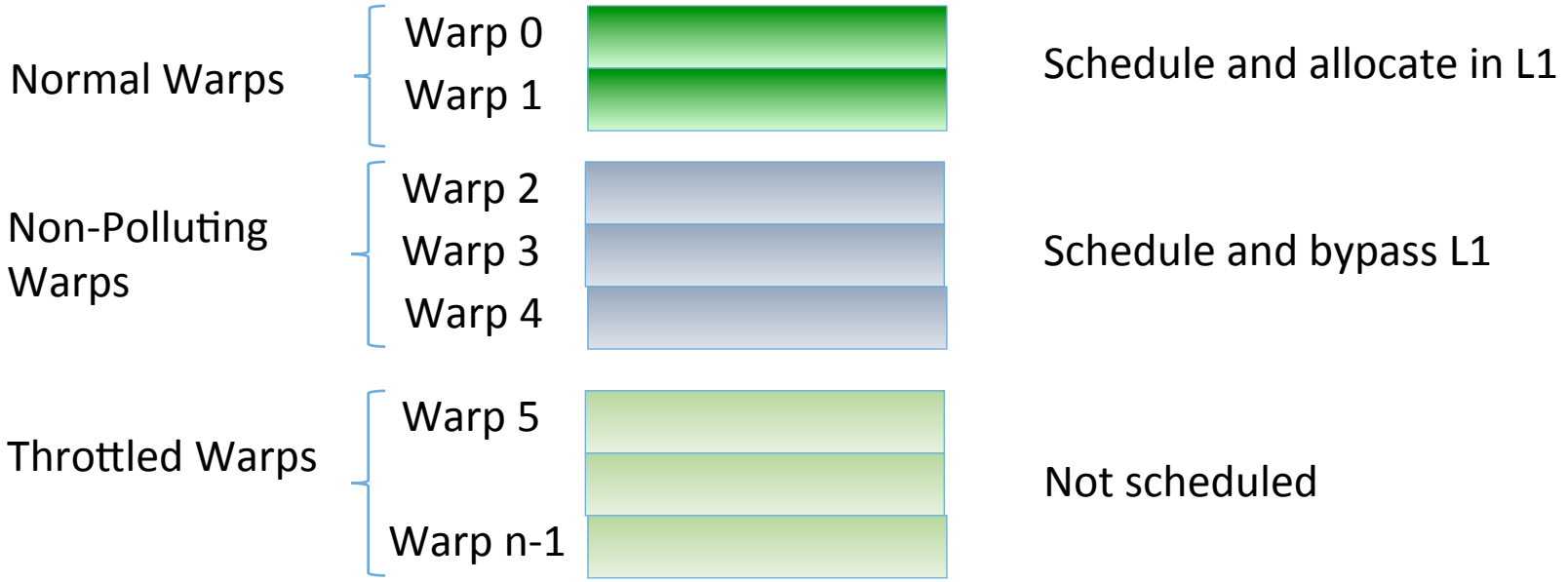
Memory Request Prioritization Buffer [Jia et al., HPCA 2014]



- Reorder requests by sorting by Warp ID.
- Bypass when too many accesses to same cache set.

Priority-Based Cache Allocation in Throughput Processors [Li et al., HPCA 2015]

- CCWS leaves L2 and DRAM underutilized.
- Allow some additional warps to execute but do not allow them to allocate space in cache:



Coordinated criticality-Aware Warp Acceleration (CAWA) [Lee et al., ISCA 2015]

- Some warps execute longer than others due to lack of uniformity in underlying workload.
- Give these warps more space in cache and more scheduling slots.
- Estimate critical path by observing amount of branch divergence and memory stalls.
- Also, predict if line inserted in line will be used by a warp that is critical using modified version of SHiP cache replacement algorithm.

Other Memory System Performance Considerations

- TLB Design for GPUs.
  - Current GPUs have translation look aside buffers (makes managing multiple graphics application surfaces easier; does not support paging)
  - How does large number of threads impact TLB design?
  - E.g., Power et al., *Supporting x86-64 Address Translation for 100s of GPU Lanes*, HPCA 2014. Importance of multithreaded page table walker + page walk cache.

Research Direction 3:
Coherent Memory for Accelerators

Why GPU Coding Difficult?

- Manual data movement CPU ⇔ GPU
- Lack of generic I/O , system support on GPU
- Need for performance tuning to reduce
  - off-chip accesses
  - memory divergence
  - control divergence
- For complex algorithms, synchronization
- Non-deterministic behavior for buggy code
- Lack of good performance analysis tools

Manual CPU ⇔ GPU Data Movement

- **Problem #1:** Programmer needs to identify data needed in a kernel and insert calls to move it to GPU
- **Problem #2:** Pointer on CPU does not work on GPU since different address spaces
- **Problem #3:** Bandwidth connecting CPU and GPU is order of magnitude smaller than GPU off-chip
- **Problem #4:** Latency to transfer data from CPU to GPU is order of magnitude higher than GPU off-chip
- **Problem #5:** Size of GPU DRAM memory much smaller than size of CPU main memory

Identifying data to move CPU ⇔ GPU

- CUDA/OpenCL: Job of programmer ☹
- C++AMP passes job to compiler.
- OpenACC uses pragmas to indicate loops that should be offloaded to GPU.

Memory Model

Rapid change (making programming easier)

- Late 1990's: fixed function graphics only
- 2003: programmable graphics shaders
- 2006: + global/local/shared (GeForce 8)
- 2009: + caching of global/local
- 2011: + unified virtual addressing
- 2014: + unified memory / coherence

Caching

- Scratchpad uses explicit data movement. Extra work. Beneficial when reuse pattern statically predictable.

- NVIDIA Fermi / AMD Southern Island add caches for accesses to global memory space.

CPU memory vs. GPU global memory

- Prior to CUDA: input data is texture map.
- CUDA 1.0 introduces cudaMemcpy
  - Allows copy of data between CPU memory space to global memory on GPU
- Still has problems:
  - #1: Programmer still has to think about it!
  - #2: Communicate only at kernel grid boundaries
  - #3: Different virtual address space
    - pointer on CPU not a pointer on GPU => cannot easily share complex data structures between CPU and GPU

Fusion / Integrated GPUs

- Why integrate?
  - One chip versus two (cf. Moore's Law, VLSI)
  - Latency and bandwidth of communication: shared physical address space, even if off-chip, eliminates copy: AMD Fusion. 1st iteration 2011. Same DRAM
  - Shared virtual address space? (AMD Kavari 2014)
  - Reduce latency to spawn kernel means kernel needs to do less to justify cost of launching

CPU Pointer not a GPU Pointer

- NVIDIA Unified Virtual Memory partially solves the problem but in a bad way:
  - GPU kernel reads from CPU memory space
- NVIDIA Uniform Memory (CUDA 6) improves by enabling automatic migration of data
- Limited academic work. Gelado et al. ASPLOS 2010.

CPU ⇔ GPU Bandwidth

- Shared DRAM as found in AMD Fusion (recent Core i7) enables the elimination of copies from CPU to GPU. Painful coding as of 2013.

- One question how much benefit versus good coding. Our limit study (WDDD 2008) found only ~50% gain. Lustig & Martonosi HPCA 2013.

- Algorithm design—MummerGPU++

CPU ⇔ GPU Latency

- NVIDIA's solution: CUDA Streams. Overlap GPU kernel computation with memory transfer. Stream = ordered sequence of data movement commands and kernels. Streams scheduled independently. Very painful programming.
- Academic work: Limit Study (WDDD 2008), Lustig & Martonosi HPCA 2013, Compiler data movement (August, PLDI 2011).

GPU Memory Size

- CUDA Streams

- Academic work: Treat GPU memory as cache on CPU memory (Kim et al., ScaleGPU, IEEE CAL early access).

Solution to all these sub-issues?

- Heterogeneous System Architecture: Integrated CPU and GPU with coherence memory address space.

- Need to figure out how to provide coherence between CPU and GPU.
- Really two problems: Coherence within GPU and then between CPU and GPU.

Research Direction 4: Easier Programming with Synchronization

Synchronization

- Locks are not encouraged in current GPGPU programming manuals.
- Interaction with SIMT stack can easily cause deadlocks:

```
  while( atomicCAS(&lock[a[tid]],0,1) != 0 )
    ;  // deadlock here if a[i] = a[j] for any i,j = tid in
warp

  // critical section goes here

  atomicExch (&lock[a[tid]], 0) ;
```

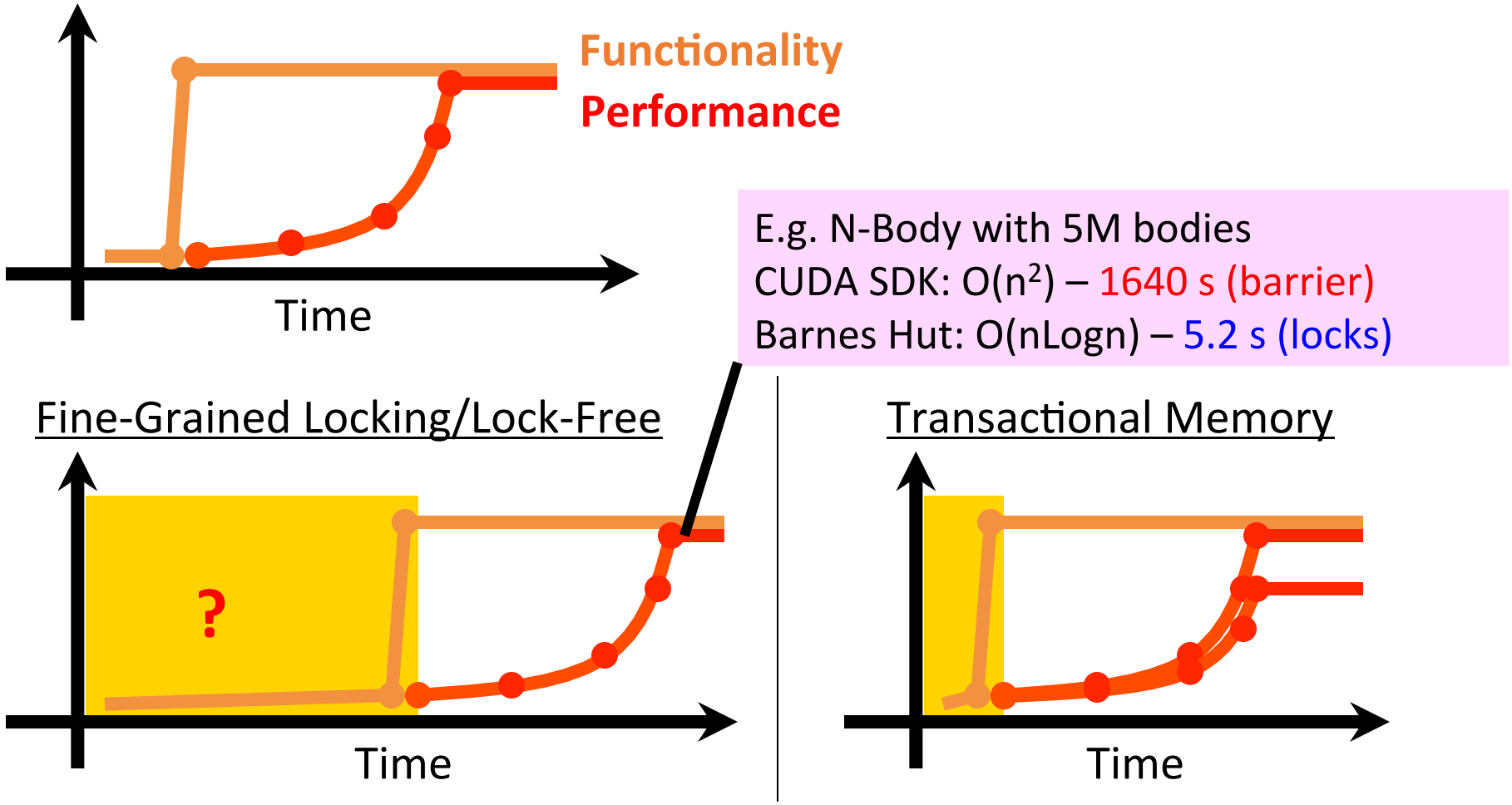
Correct way to write critical section for GPGPU:

```
done = false;
while( !done ) {
  if( atomicCAS (&lock[a[tid]], 0 , 1 )==0 ) {

    // critical section goes here

    atomicExch(&lock[a[tid]], 0 ) ;
  }
}
```

Most current GPGPU programs use barriers within thread blocks and/or lock-free data structures.

This leads to the following picture…

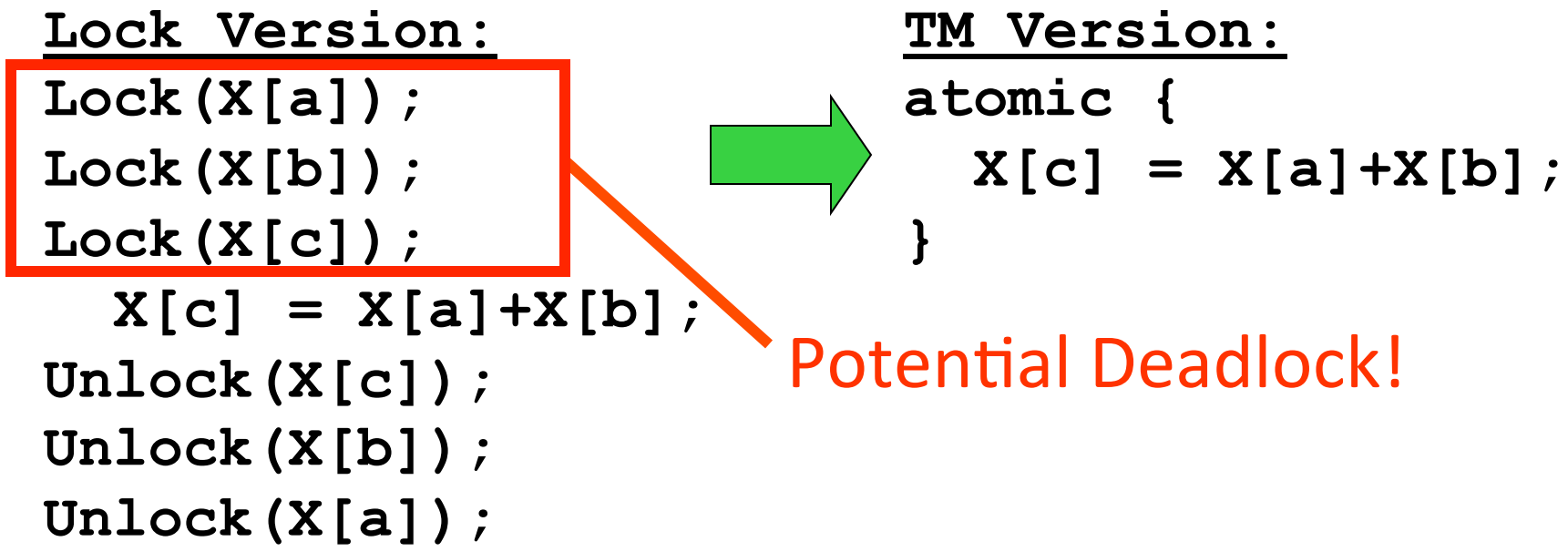
- Lifetime of GPU Application Development



Wilson Fung, Inderpeet Singh, Andrew Brownsword, Tor Aamodt

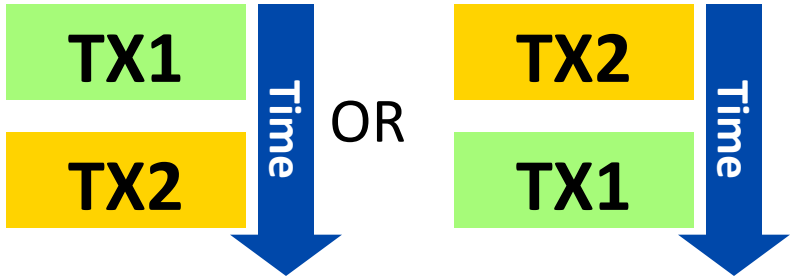
Transactional Memory

- Programmer specifies atomic code blocks called transactions [Herlihy'93]

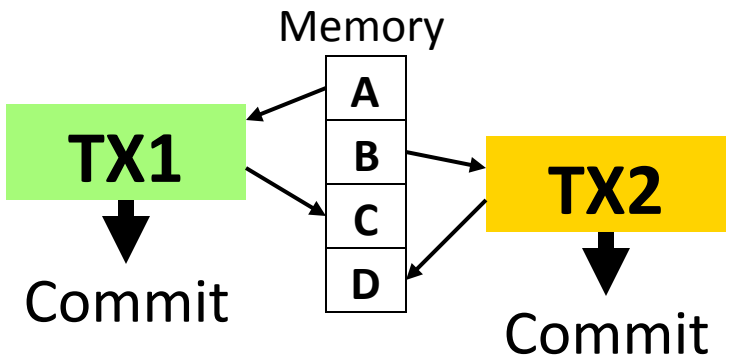


Transactional Memory

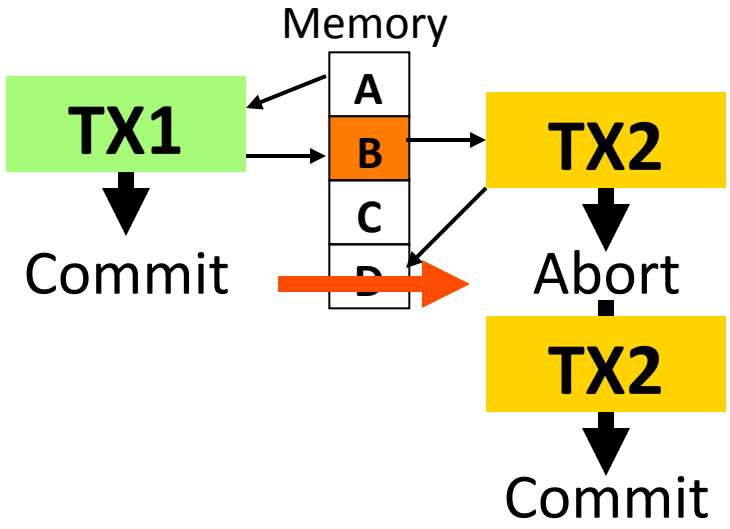
Programmers' View:



Non-conflicting transactions may run in parallel



Conflicting transactions automatically serialized



Are TM and GPUs Incompatible?

GPU uarch very different from multicore CPU…

KILO TM [MICRO'11, IEEE Micro Top Picks]

- Hardware TM for GPUs
- Half performance of fine grained locking
- Chip area overhead of 0.5%

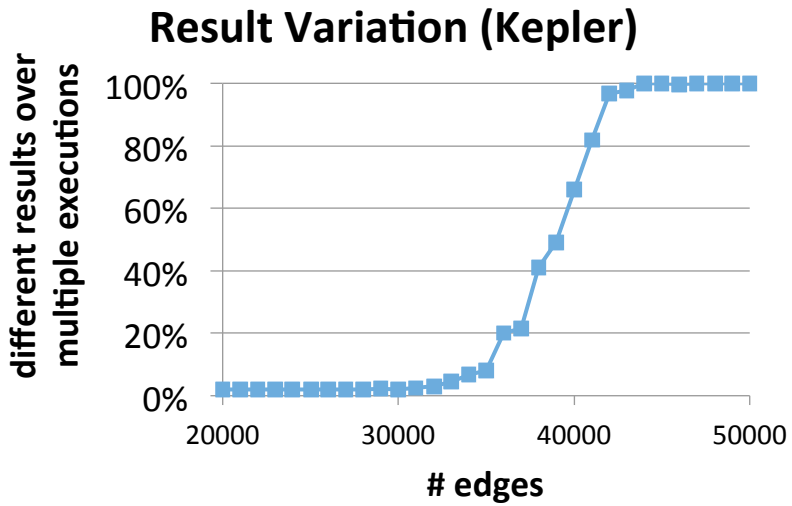
Research Direction 5:
GPU Power Efficiency

GPU power

- More efficient than CPU but
  - Consumes a lot of power
  - Much less efficient than ASIC or FPGAs
  - What can be done to reduce power consumption?
- Look at the most power hungry components
  - What can be duty cycled/power gated?
  - GPUWattch to evaluate ideas

Other Research Directions….

- Non-deterministic behavior for buggy code
  - GPUDet ASPLOS 2013



- Lack of good performance analysis tools
  - NVIDIA Profiler/Parallel NSight
  - AerialVision [ISPASS 2010]
  - GPU analytical perf/power models (Hyesoon Kim)

Lack of I/O and System Support…

- Support for printf, malloc from kernel in CUDA
- File system I/O?
- GPUfs (ASPLOS 2013):
  - POSIX-like file system API
  - One file per warp to avoid control divergence
  - Weak file system consistency model (close->open)
  - Performance API: O_GWRONCE, O_GWRONCE
  - Eliminate seek pointer
- GPUnet (OSDI 2014): Posix like API for sockets programming on GPGPU.

Conclusions

- GPU Computing is growing in importance due to energy efficiency concerns
- GPU architecture has evolved quickly and likely to continue to do so
- We discussed some of the important microarchitecture bottlenecks and recent research.
- Also discussed some directions for improving programming model