ACACES 2018 Summer School

GPU Architectures: Basic to Advanced Concepts

Adwait Jog, Assistant Professor

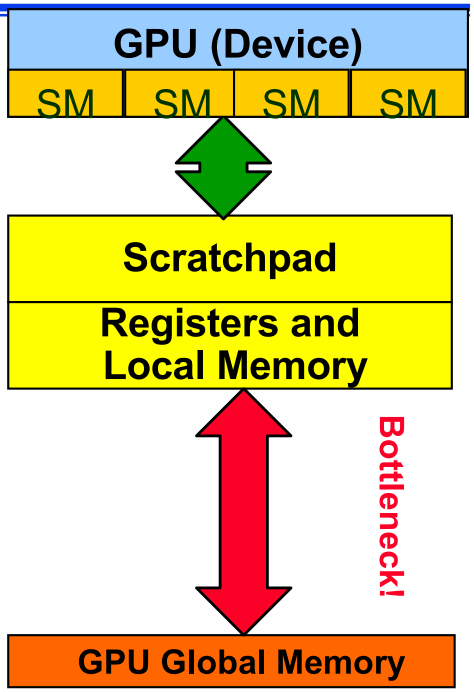
College of William & Mary
(http://adwaitjog.github.io/)

Course Outline

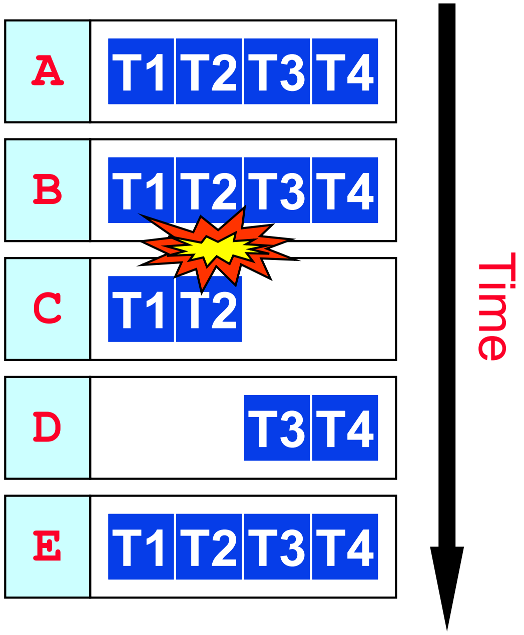
- Lectures 1 and 2: Basics Concepts
  - Basics of GPU Programming
  - Basics of GPU Architecture
- Lecture 3: GPU Performance Bottlenecks
  - Memory Bottlenecks
  - Compute Bottlenecks
  - Possible Software and Hardware Solutions
- Lecture 4: GPU Security Concerns
  - Timing channels
  - Possible Software and Hardware Solutions

Key GPU Performance Concerns

Memory Concerns: Data transfers between SMs and global memory are costly.



Compute Concerns: Threads that do not take the same *control* path lead to serialization in the GPU compute pipeline.



Reducing Off-Chip Access

- ❑ Re-writing software to use “shared memory” and avoid un-coalesced global accesses is difficult for the GPU programmer.

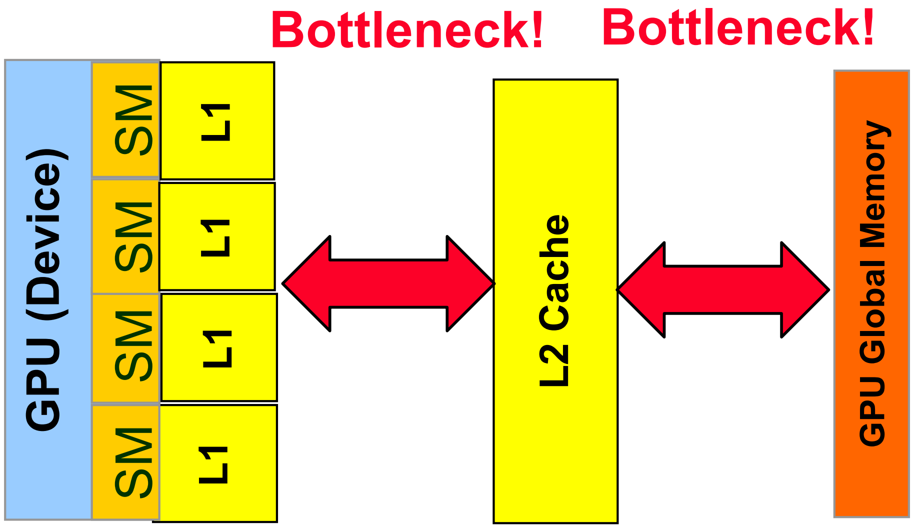
- ❑ Recent GPUs introduce hardware-managed caches (L1/L2), but large number of threads lead to thrashing.

- ❑ General purpose code, now being ported to GPUs, has branches and irregular accesses. Not always possible to fix them in the code.

We need intelligent hardware solutions!

I) Alleviating the Memory Bottlenecks

– Memory concerns: Thousands of threads running on SMs need data from DRAM, however, DRAM bandwidth is limited. Increasing it is very costly

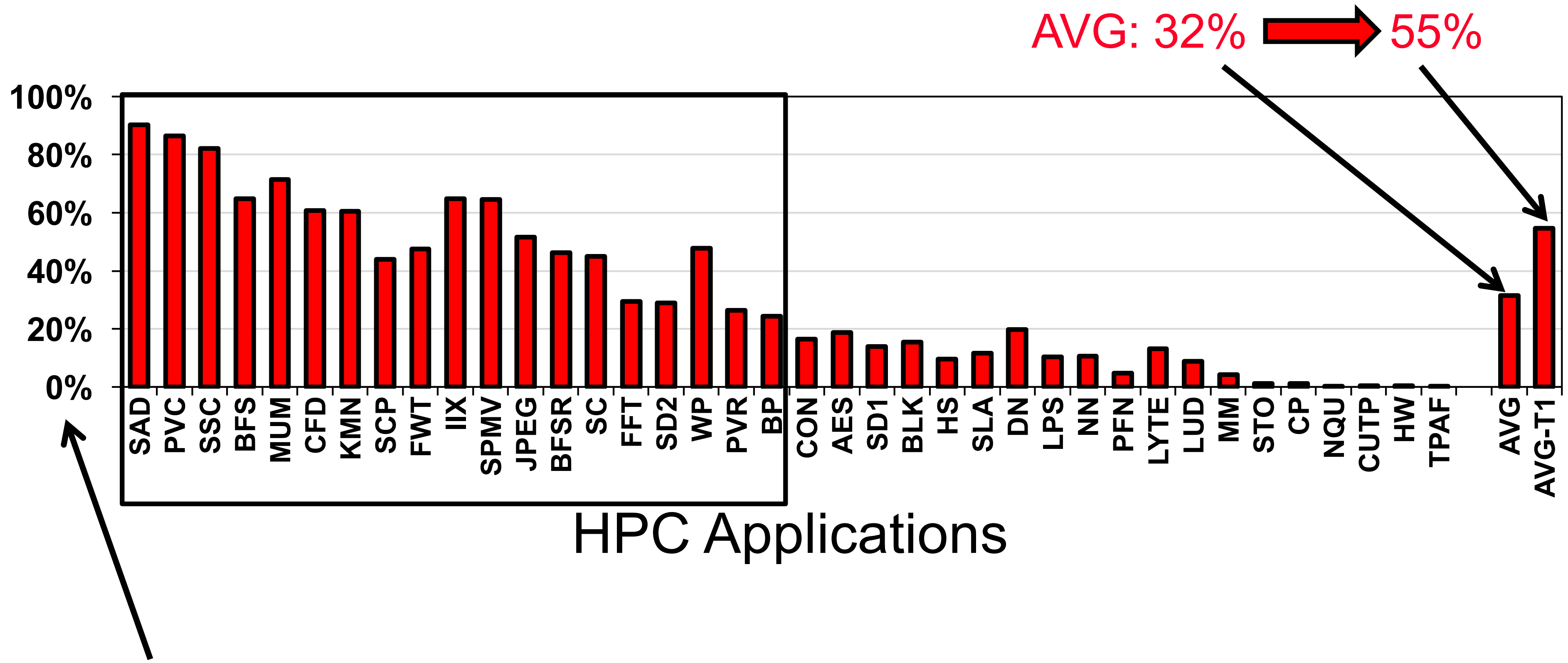


– Q1. How can we use caches effectively to reduce the bandwidth demand?

– Q2. Can we effectively data compression and reduce the data consumption?

– Q3. How can we effectively/fairly allocate memory bandwidth across concurrent streams/apps?

Quantifying Memory Bottlenecks



AVG: 32% → 55%

Percentage of total execution cycles wasted waiting for the data to come back from memory.

[Jog et al., ASPLOS 2013]

Strategies

- Cache-Aware Warp Scheduling Techniques
  - Effective caching → Less Pressure on Memory

- Employing Assist Warps for Helping Data Compression
  - Bandwidth Preserved

- Bandwidth Allocation Strategies for Multi-Application execution on GPUs
  - Better System Throughput and Fairness

Application-Architecture Co-Design

- Architecture: GPUs typically employ smaller caches compared to CPUs
- Scheduler: Many warps concurrently access the small caches in a *round-robin* manner leading to thrashing.

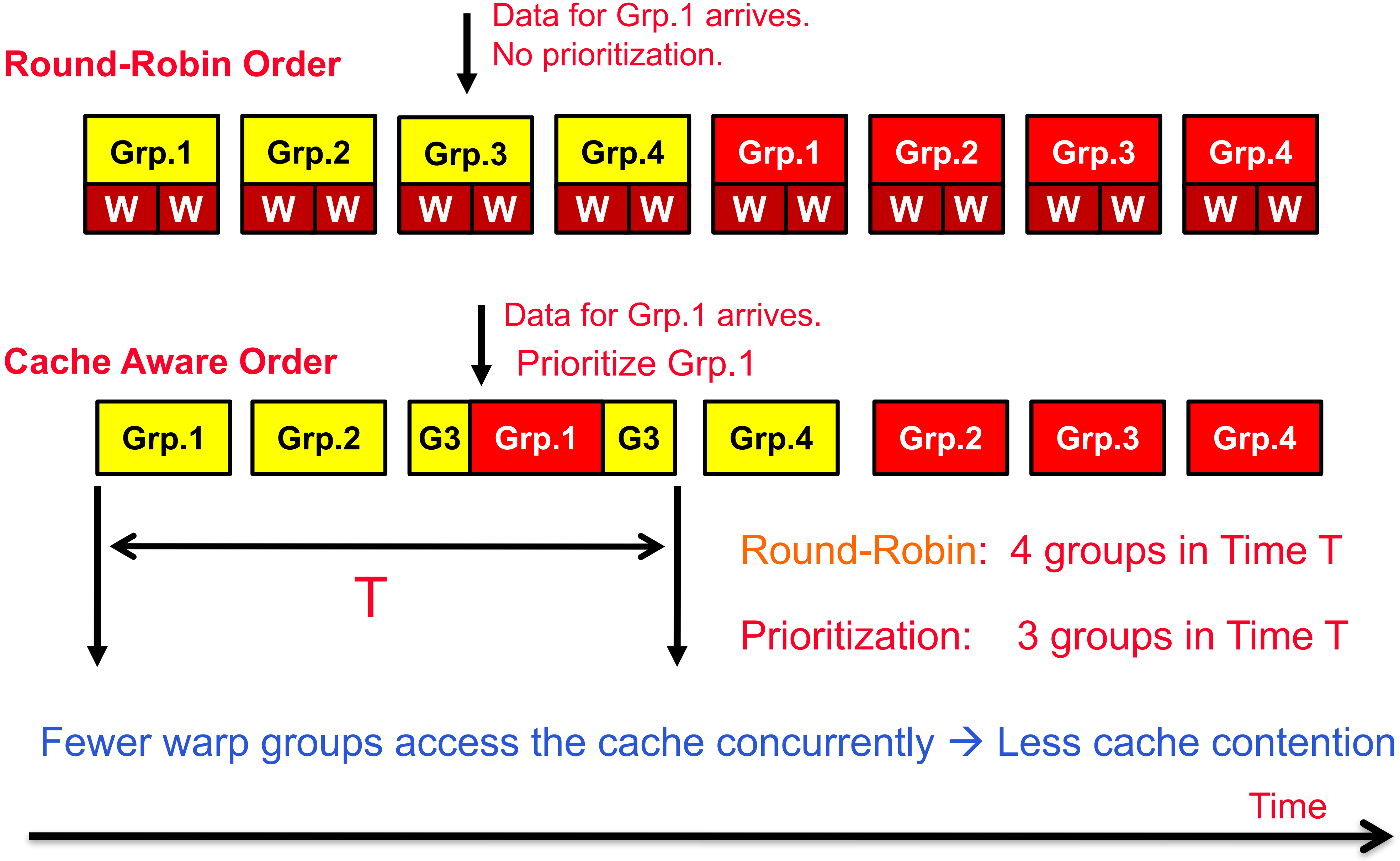
Cache Aware Scheduling

- Philosophy: "One work at a time"

- Working
  - Select a "group" (*work*) of warps
  - *Always* prioritizes it over other groups
  - Group switch is not round-robin

- Benefits:
  - Preserve locality
  - Fewer Cache Misses

Improve L1 Cache Hit Rates



Round-Robin Order

Data for Grp.1 arrives.
No prioritization.

Grp.1 W W | Grp.2 W W | Grp.3 W W | Grp.4 W W | Grp.1 W W | Grp.2 W W | Grp.3 W W | Grp.4 W W

Cache Aware Order

Data for Grp.1 arrives.
Prioritize Grp.1

Grp.1 | Grp.2 | G3 | Grp.1 | G3 | Grp.4 | Grp.2 | Grp.3 | Grp.4

T

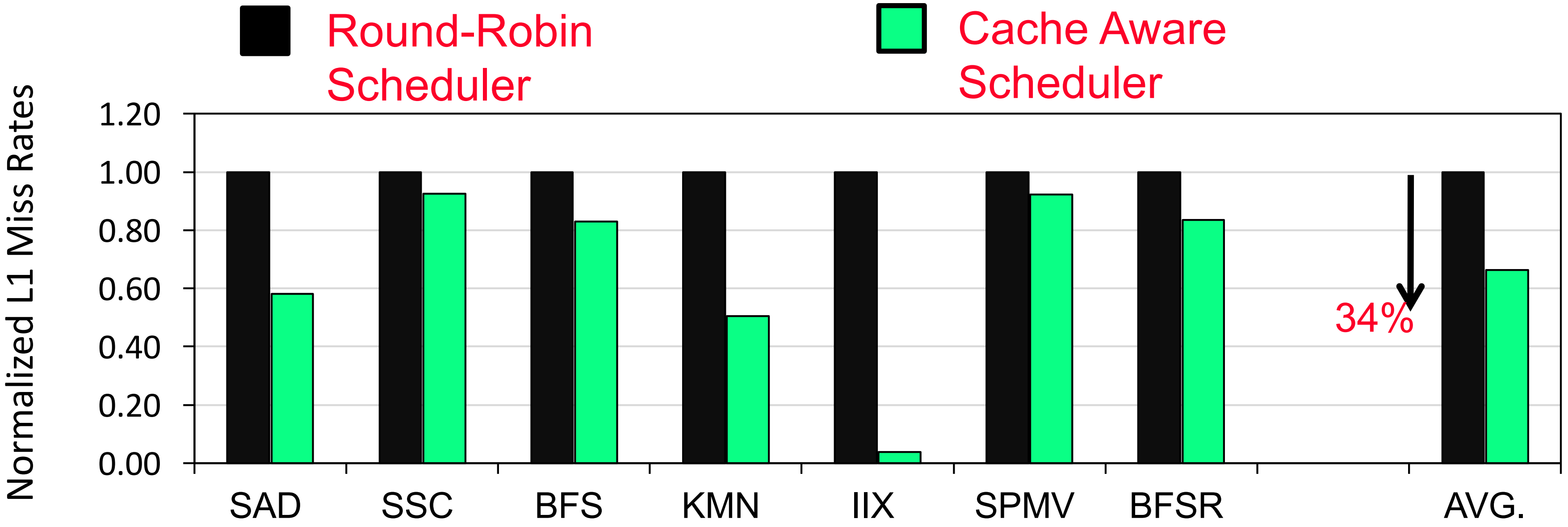
Round-Robin: 4 groups in Time T

Prioritization: 3 groups in Time T

Fewer warp groups access the cache concurrently → Less cache contention

Time

Reduction in L1 Miss Rates



- 25% improvement in IPC across 19 applications
- Limited benefits for cache insensitive applications
- Software Support (e.g., specify data-structures that should be "*uncacheable*")

[Jog et al., ASPLOS 2013]

Other Sophisticated Mechanisms

- Rogers et al., Cache Conscious Wavefront Scheduling, MICRO’12
- Kayiran et al., Neither more Nor Less: Optimizing Thread-level Parallelism for GPGPUs, PACT’13
- Chen et al., Adaptive cache management for energy-efficient GPU computing, MICRO’14
- Lee et al., CAWS: criticality-aware warp scheduling for GPGPU workloads

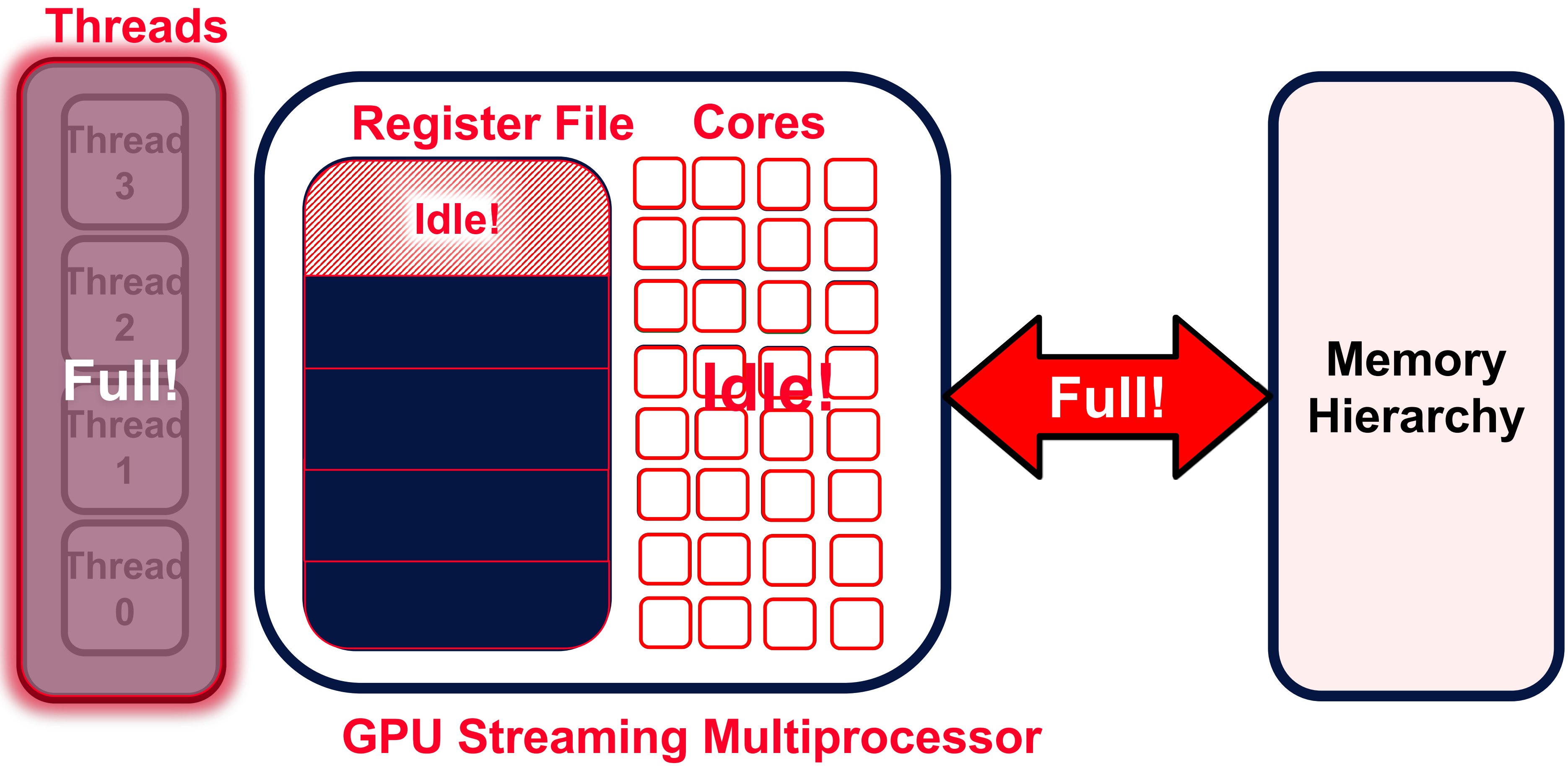
Strategies

- Cache-Aware Warp Scheduling Techniques
  - Effective caching → Less Pressure on Memory

- Employing Assist Warps for Helping Data Compression
  - Bandwidth Preserved

- Bandwidth Allocation Strategies for Multi-Application execution on GPUs
  - Better System Throughput and Fairness

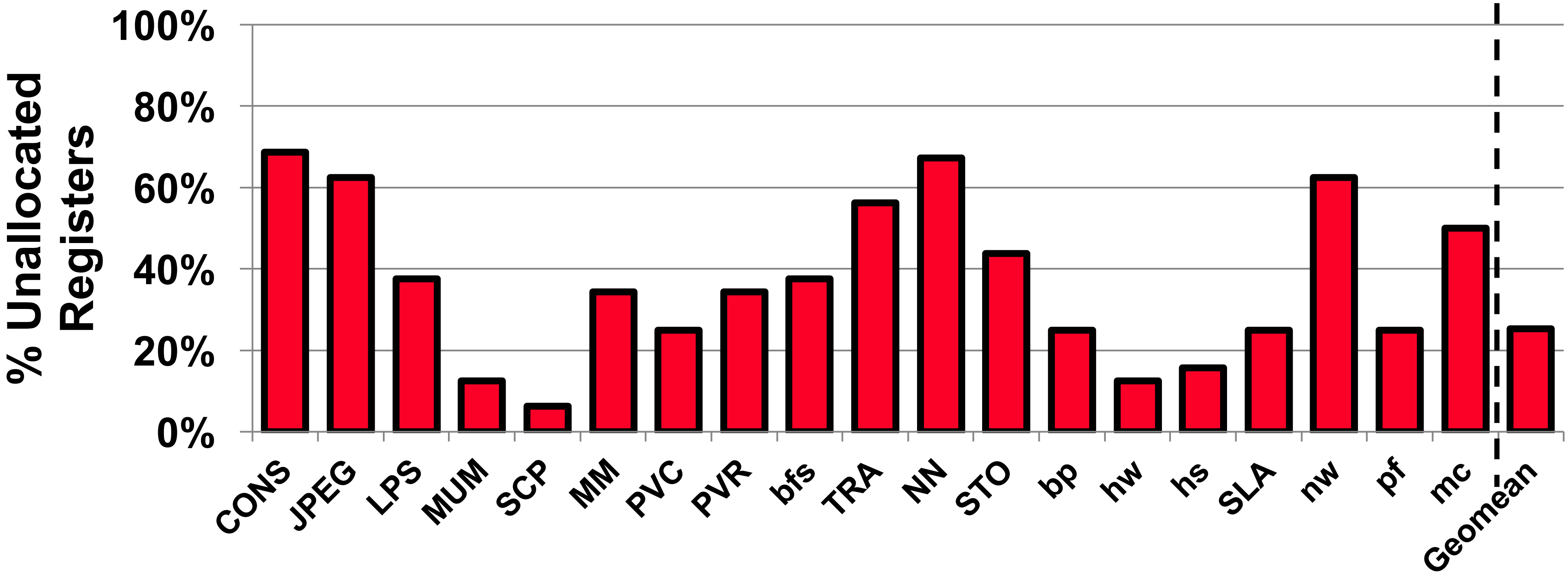
Challenges in GPU Efficiency



The memory bandwidth bottleneck leads to idle cores

Thread limits lead to an underutilized register file

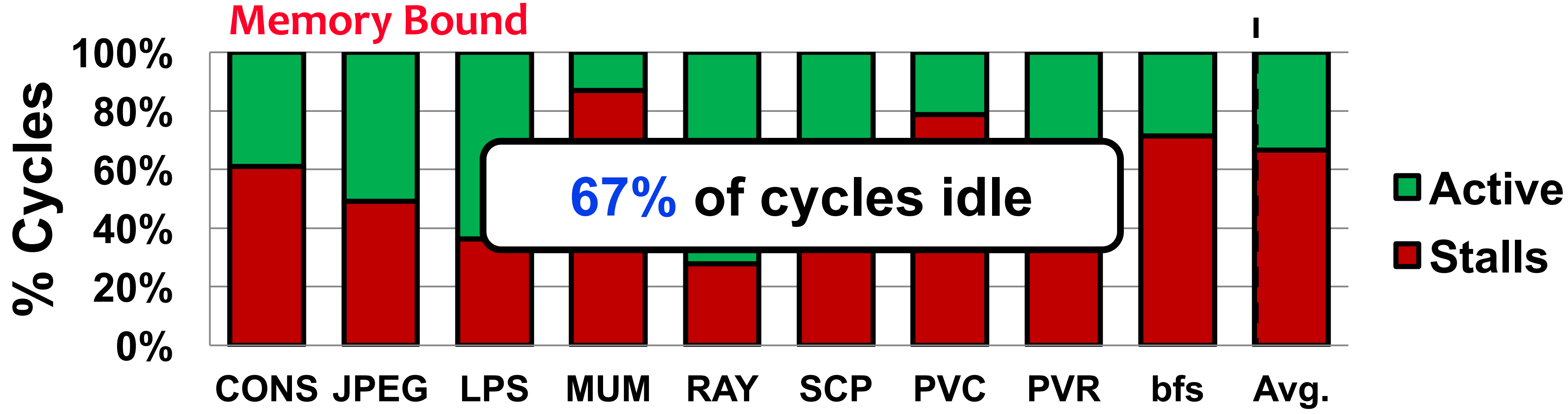
Motivation: Unutilized On-chip Memory



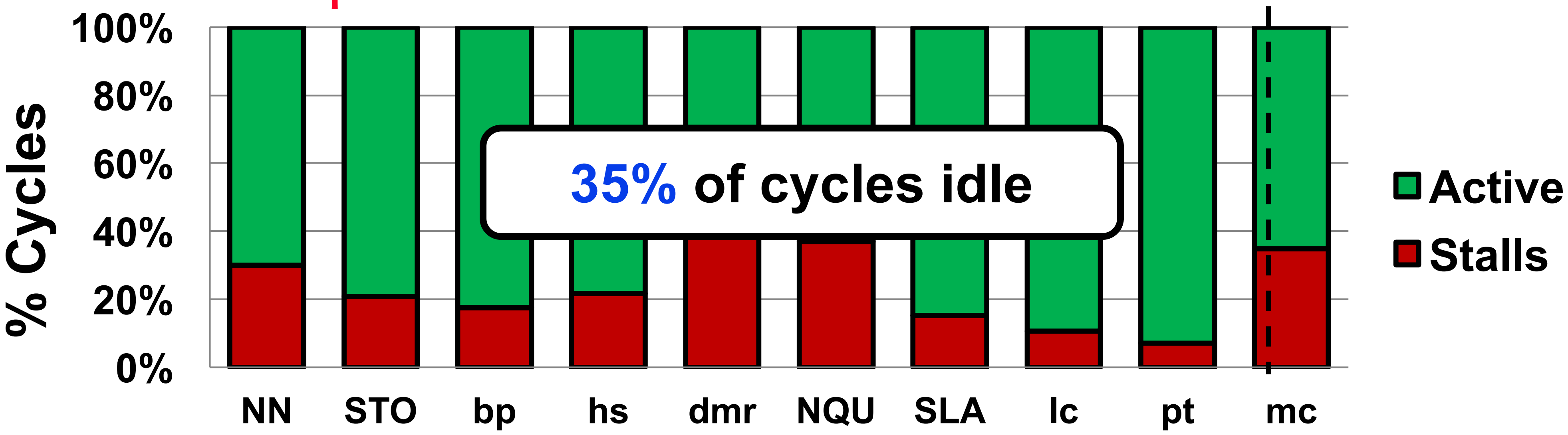
- 24% of the register file is unallocated on average
- Similar trends for on-chip scratchpad memory

Motivation: Idle Pipelines

Memory Bound


67% of cycles idle

Compute Bound


35% of cycles idle

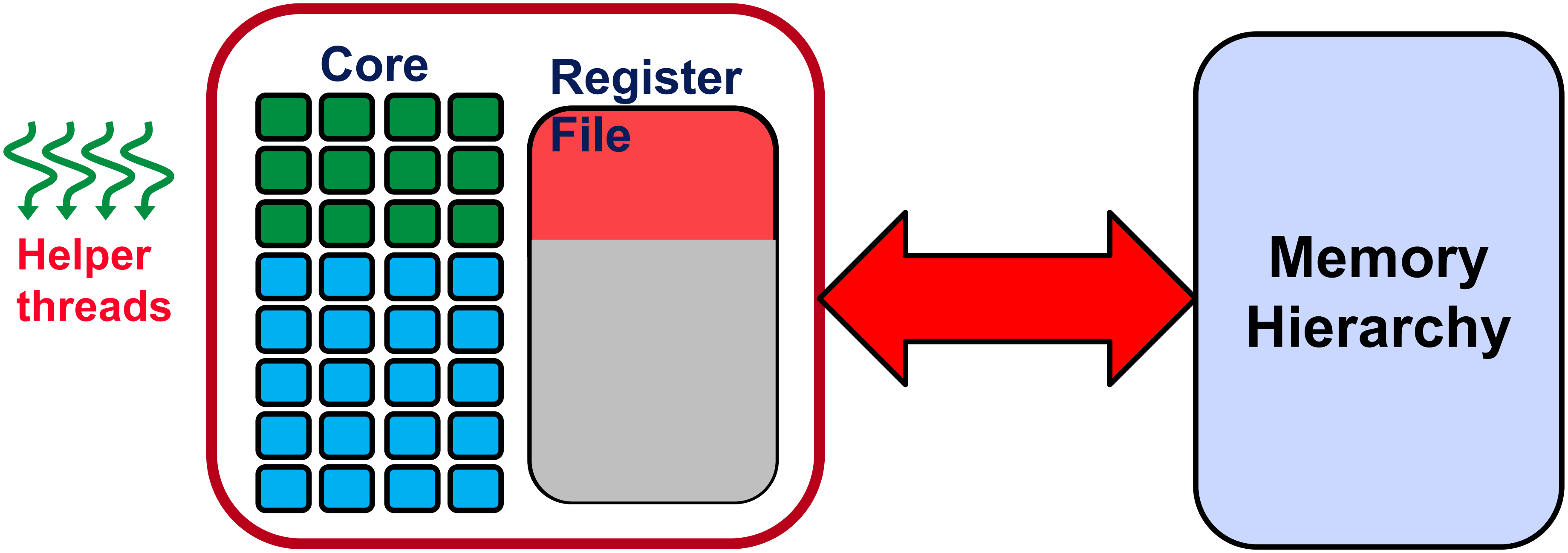
Motivation: Summary

Heterogeneous application requirements lead to:

- **Bottlenecks** in execution
- **Idle** resources

Our Goal

- Use idle resources to do something useful: **accelerate bottlenecks using helper threads**



- A flexible framework to enable helper threading in GPUs: **C**ore-**A**ssisted **B**ottleneck **A**cceleration (**CABA**)

Helper threads in GPUs

- Large body of work in CPUs …
  - [Chappell+ ISCA ’99, MICRO ’02], [Yang+ USC TR ’98], [Dubois+ CF ’04], [Zilles+ ISCA ’01], [Collins+ ISCA ’01, MICRO ’01], [Aamodt+ HPCA ’04], [Lu+ MICRO ’05], [Luk+ ISCA ’01], [Moshovos+ ICS ’01], [Kamruzzaman+ ASPLOS ’11], etc.
- **However, there are new challenges with GPUs…**

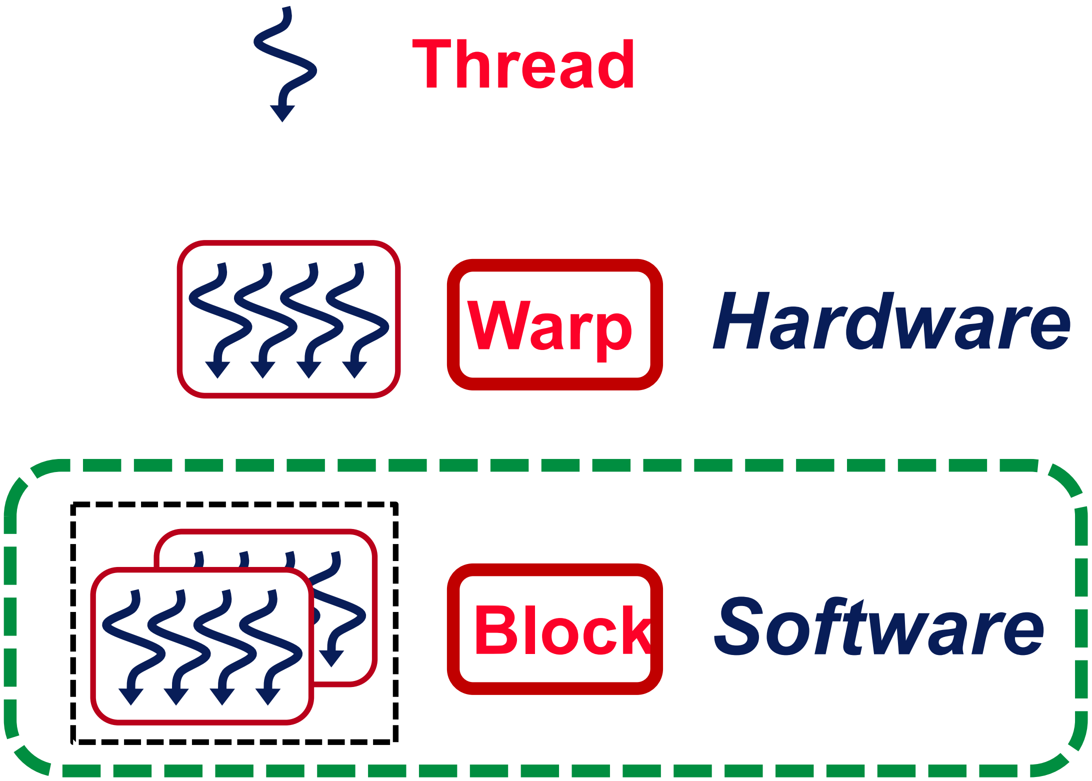
Challenge

How do you efficiently

manage and use helper threads

in a throughput-oriented architecture?

Managing Helper Threads in GPUs



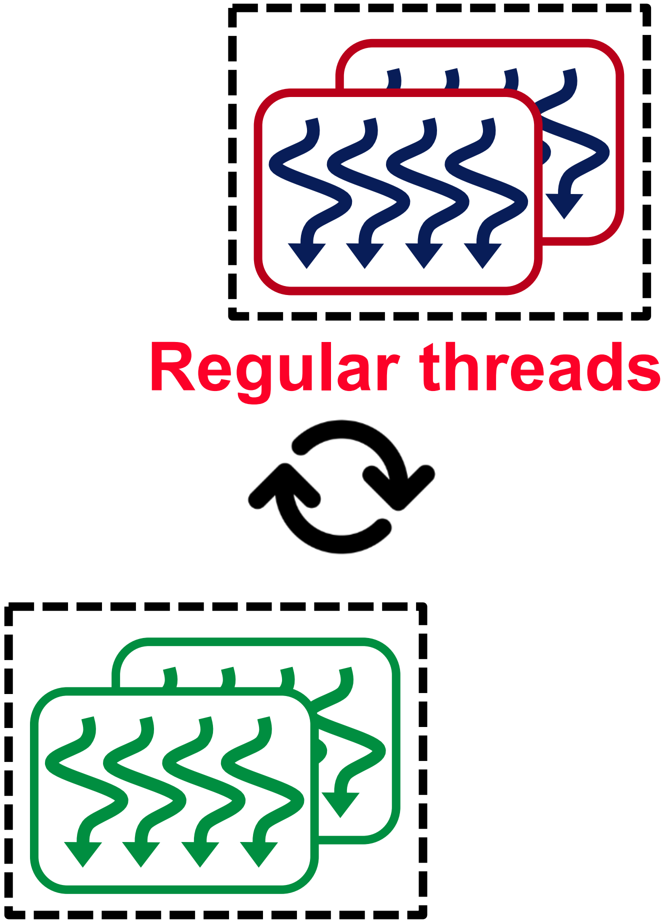
Thread

Warp *Hardware*

Block *Software*

Where do we add helper threads?

Approach #1: Software-only

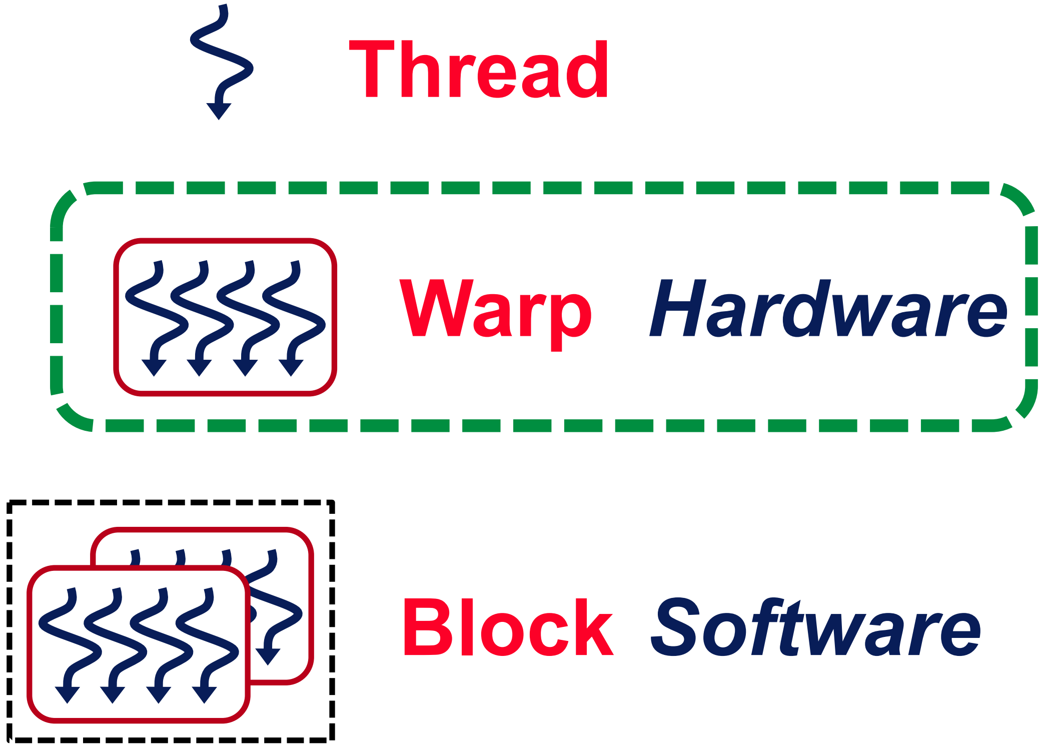


Regular threads

Helper threads

- ✓ No hardware changes
- × Coarse grained
- × Synchronization is difficult
- × Not aware of runtime program behavior

Where Do We Add Helper Threads?



Thread

Warp *Hardware*

Block *Software*

Other functionality

In the paper:

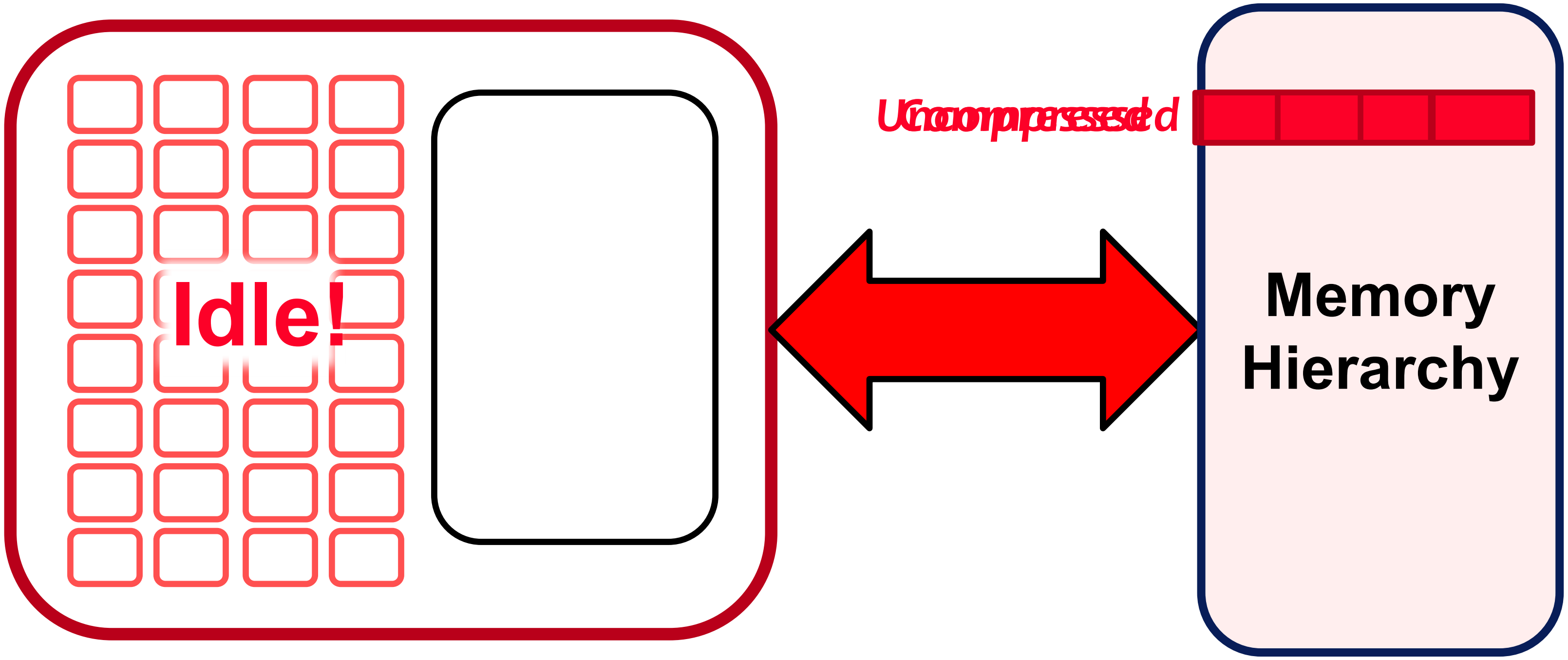
- More details on the hardware structures
- Data communication and synchronization
- Enforcing priorities

CABA: Applications

- Data compression
- Memoization
- Prefetching
- Encyrption …

A Case for CABA: Data Compression

- **Data compression** can help alleviate the **memory bandwidth bottleneck** - transmits data in a more condensed form

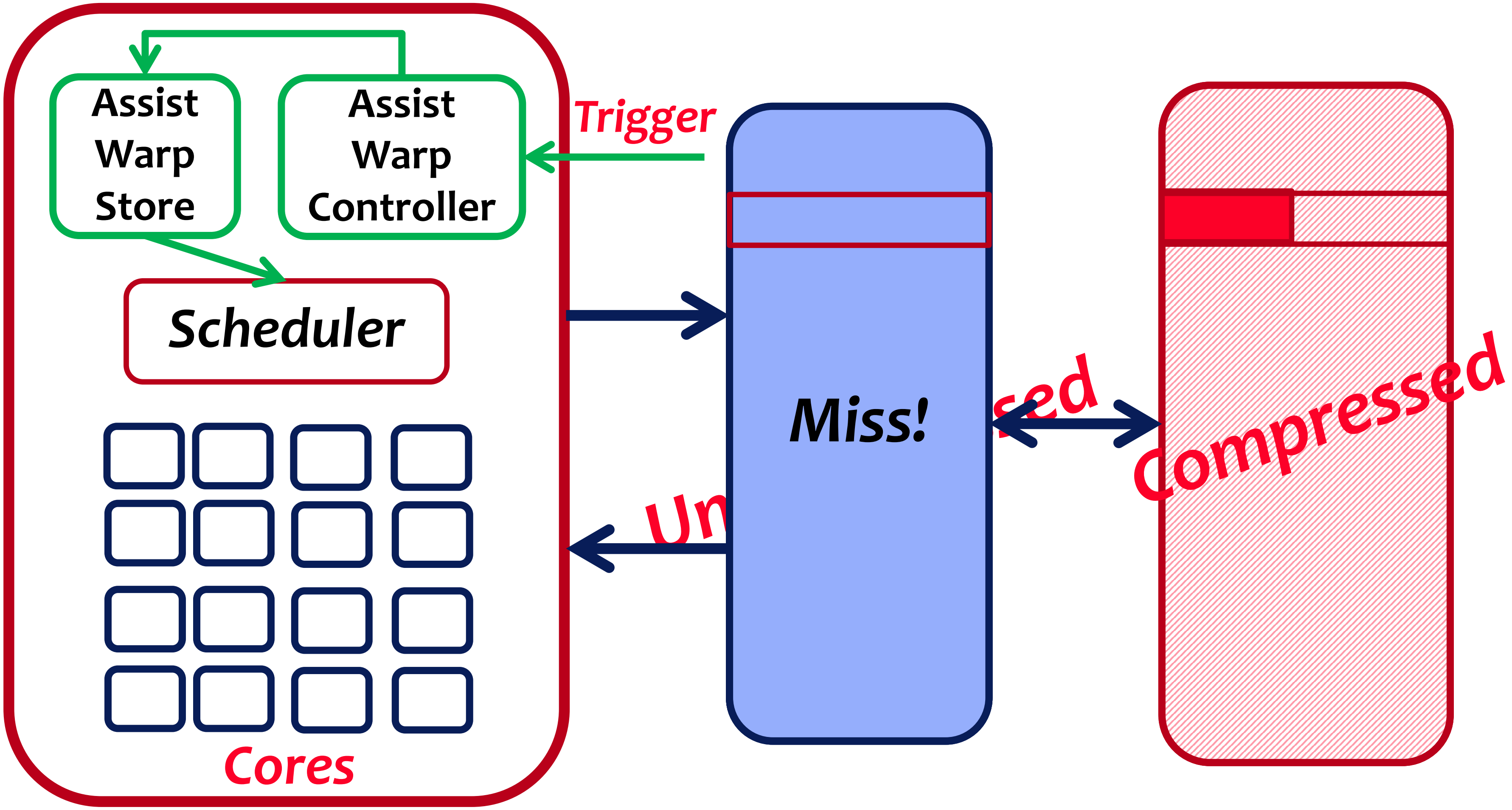


- CABA employs idle compute pipelines to perform compression

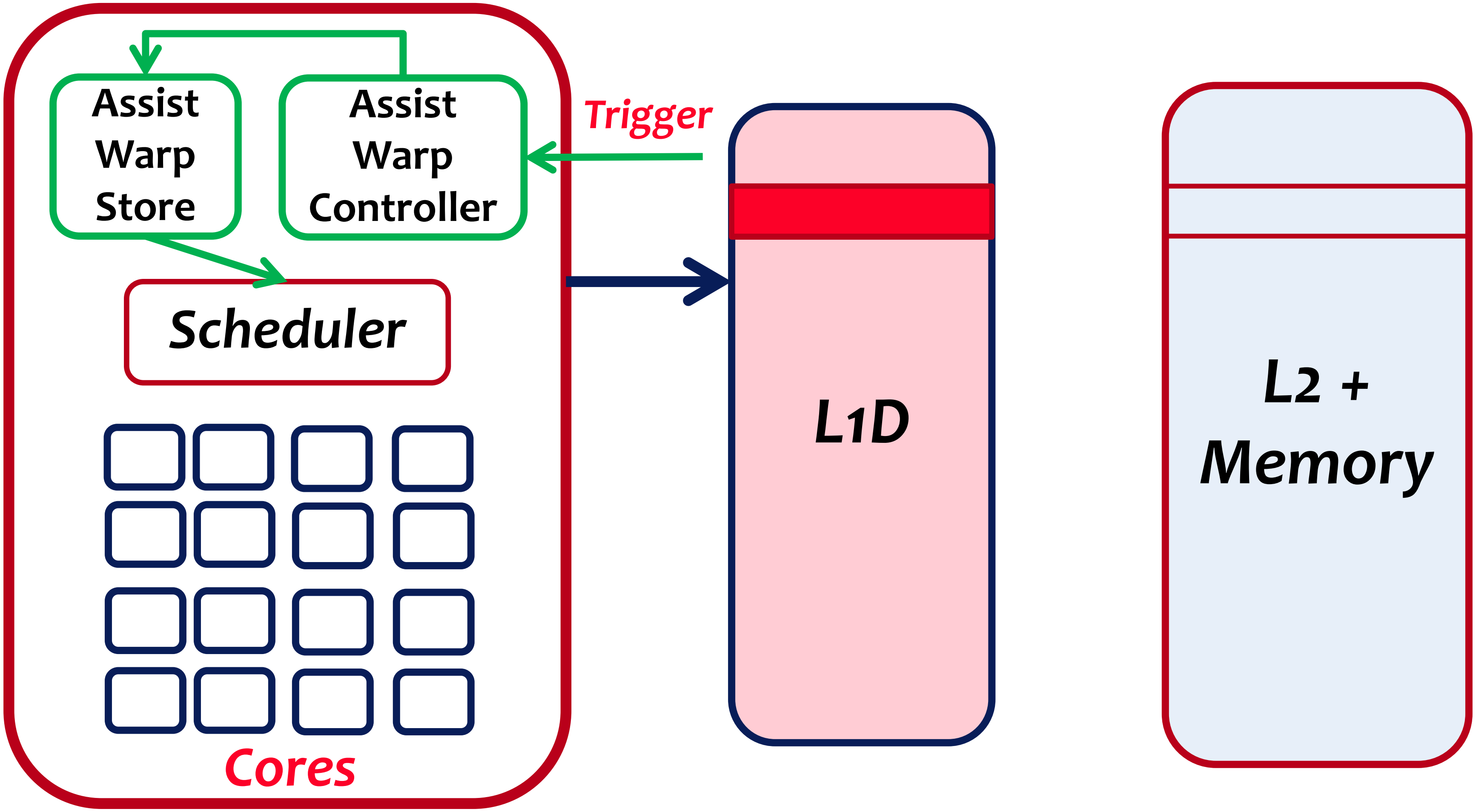
Data Compression with CABA

- Use assist warps to:
  - Compress cache blocks before writing to memory
  - Decompress cache blocks before placing into the cache
- CABA flexibly enables various compression algorithms
- Example: **BDI Compression** **[Pekhimenko+ PACT '12]**
  - Parallelizable across SIMT width
  - Low latency
- Others: **FPC** **[Alameldeen+ TR '04], C-Pack [Chen+ VLSI '10]**

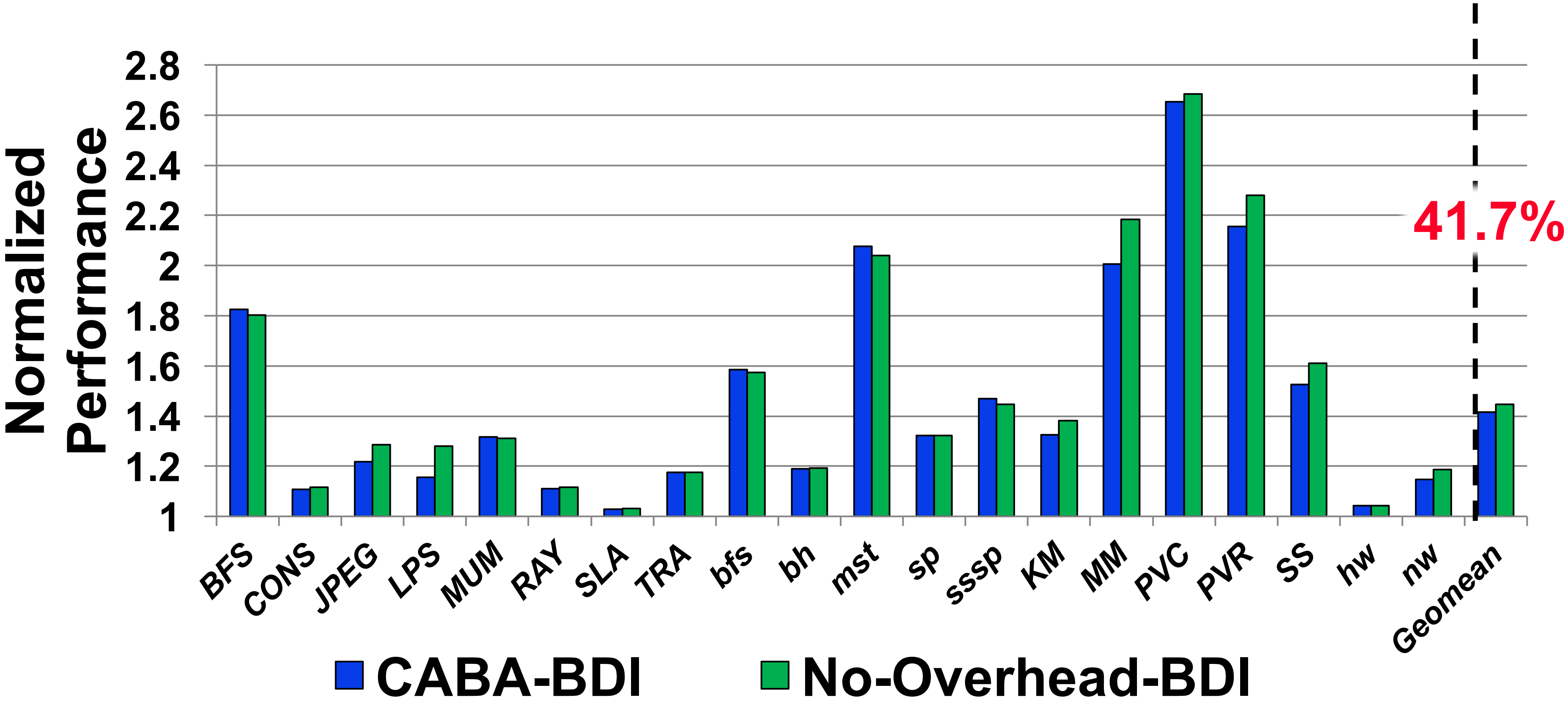
Walkthrough of Decompression



Walkthrough of Compression

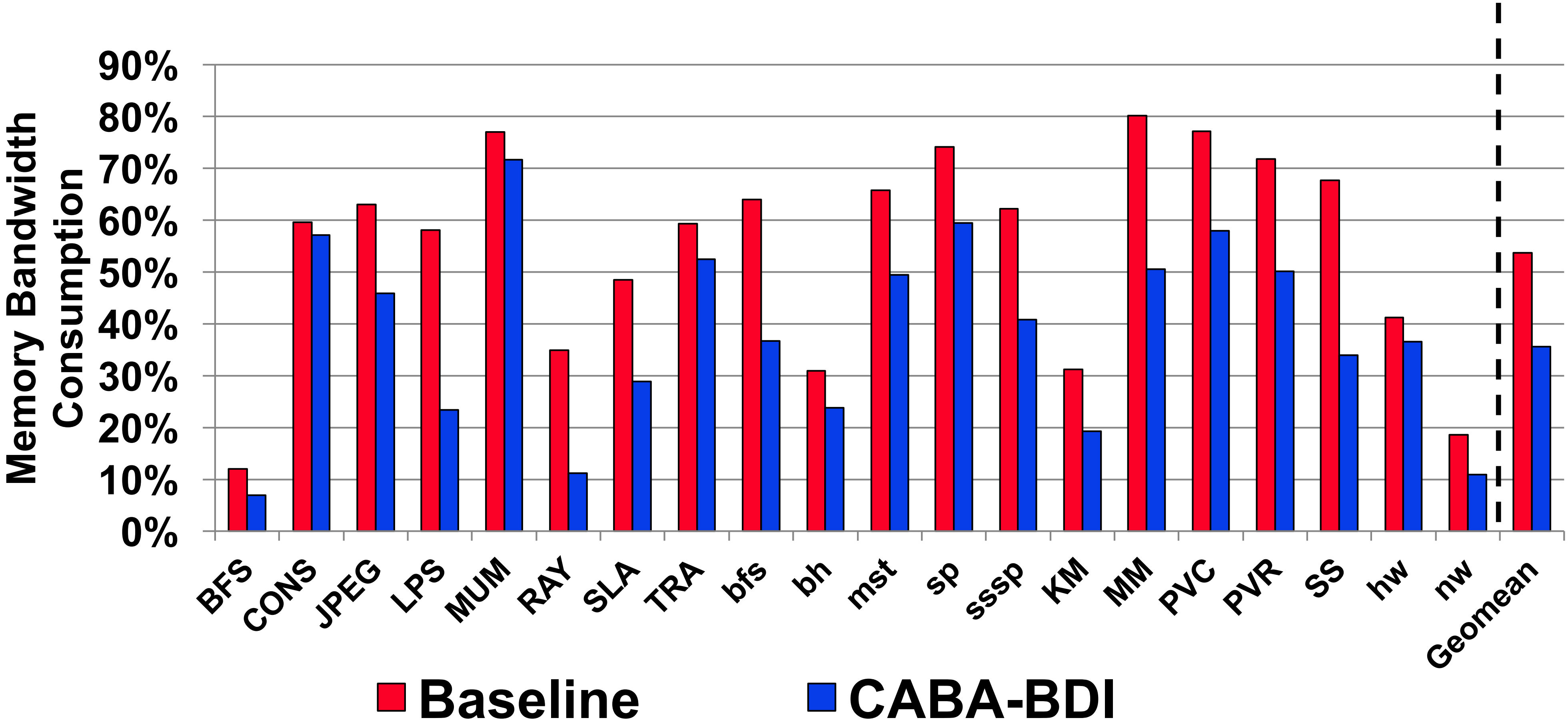


Effect on Performance



- CABA provides a 41.7% performance improvement
- CABA achieves performance close to that of designs with no overhead for compression

Effect on Bandwidth Consumption



Data compression with CABA alleviates the memory bandwidth bottleneck

Conclusion

- **Observation:** Imbalances in execution leave GPU resources underutilized

- **Goal:** Employ underutilized GPU resources to do something useful – **accelerate bottlenecks using helper threads**

- **Challenge:** How do you efficiently **manage and use** helper threads in a **throughput-oriented** architecture?

- **Solution: CABA (Core-Assisted Bottleneck Acceleration, ISCA'15)**
  - A new framework to enable helper threading in GPUs
  - Enables flexible data compression to alleviate the memory bandwidth bottleneck
  - A wide set of use cases (e.g., prefetching, memoization)

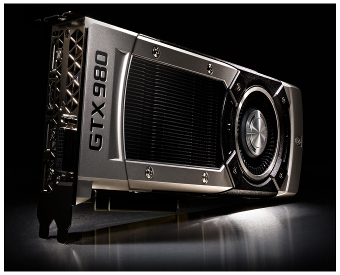
Strategies

- Cache-Aware Warp Scheduling Techniques
  - Effective caching → Less Pressure on Memory

- Employing Assist Warps for Helping Data Compression
  - Bandwidth Preserved

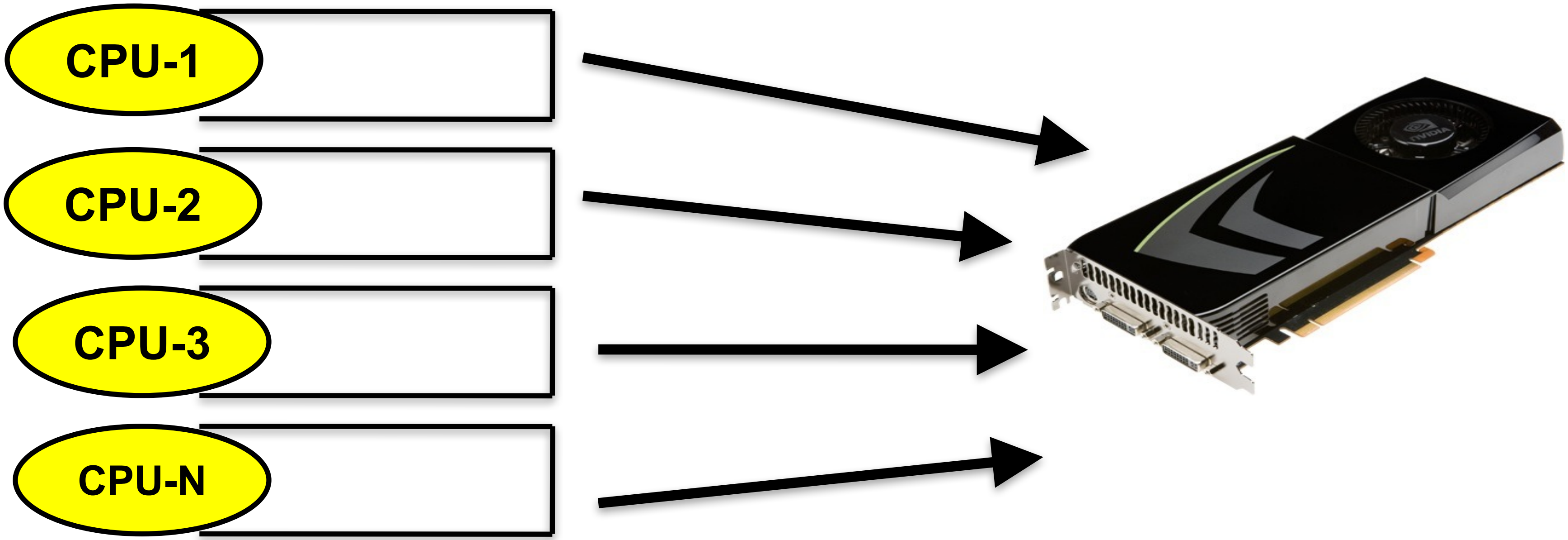
- Bandwidth Allocation Strategies for Multi-Application execution on GPUs
  - Better System Throughput and Fairness

Discrete GPU Cards --- Scaling Trends

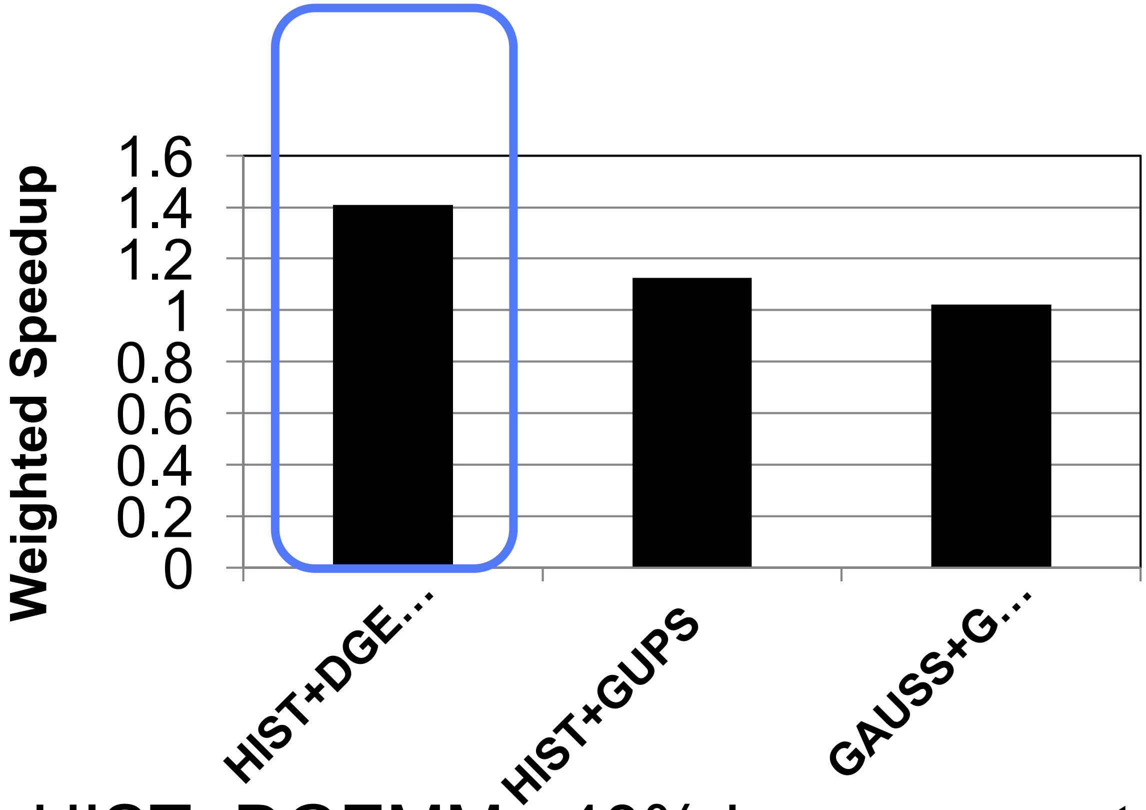
2008	2010	2012	2014	2016	2018
					
GTX 275	GTX 480	GTX 680	GTX 980	GP 100	GV 100
(Tesla)	*(Fermi)*	*(Kepler)*	*(Maxwell)*	*(Pascal)*	*(Volta)*
240 CUDA Cores	448 CUDA Cores	1536 CUDA Cores	2048 CUDA Cores	3584 CUDA Cores	5120 CUDA Cores
(127 GB/sec)	(139 GB/sec)	(192 GB/sec)	(224 GB/sec)	(720 GB/sec)	(900 GB/sec)

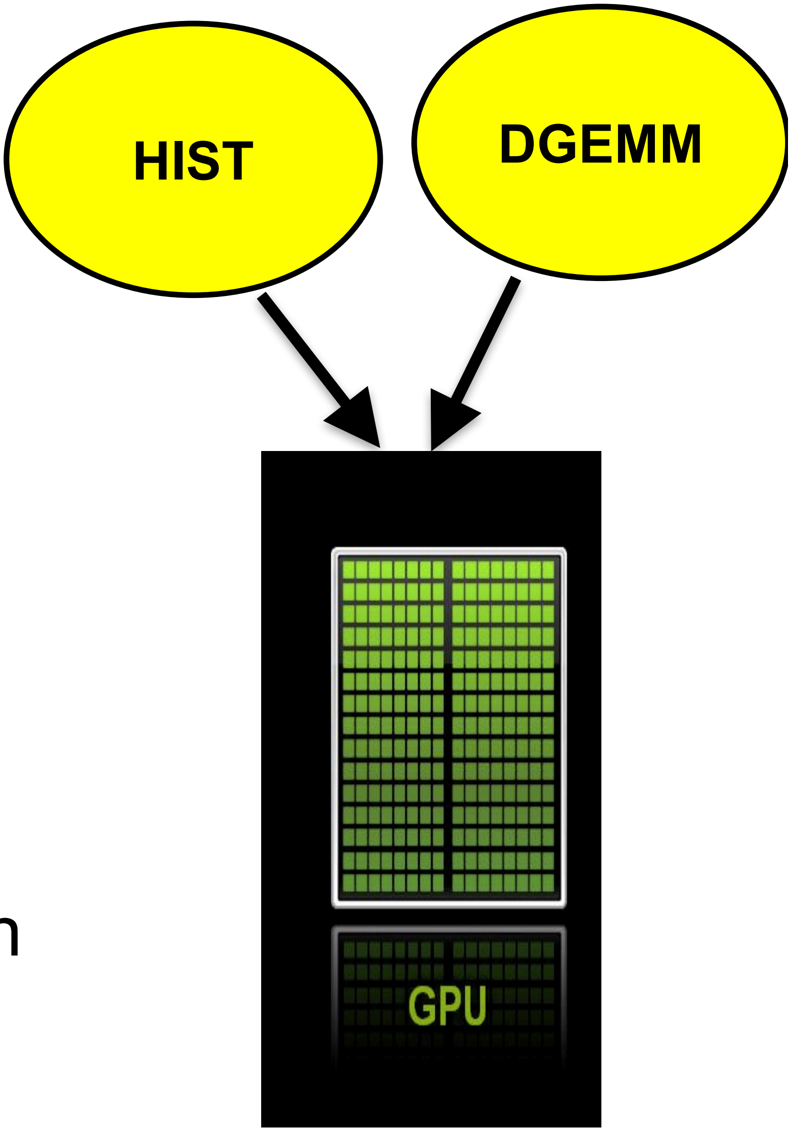
Multi-Application Execution

- ❑ Not all applications have enough parallelism
  - ● GPU resources can be under-utilized
- ❑ Multiple CPUs send requests to GPUs
- ❑ Multiple players concurrently play games on the cloud



System Throughput (Jobs/sec)

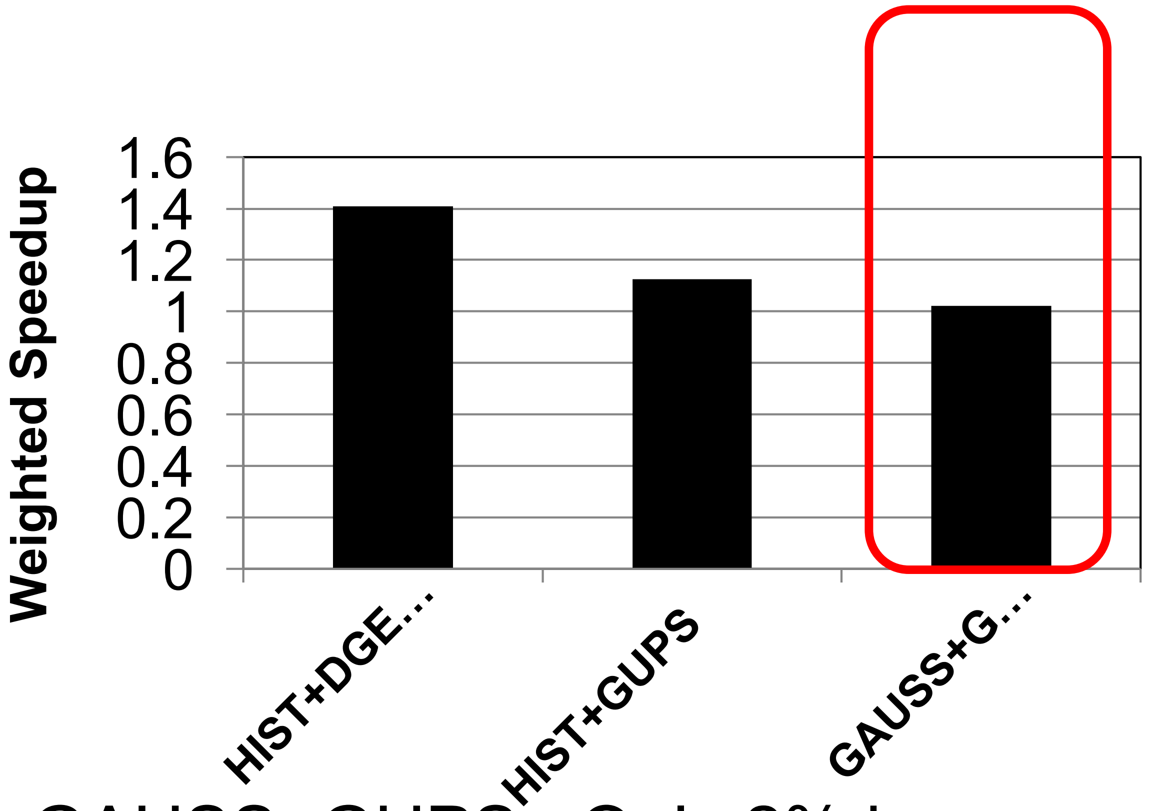


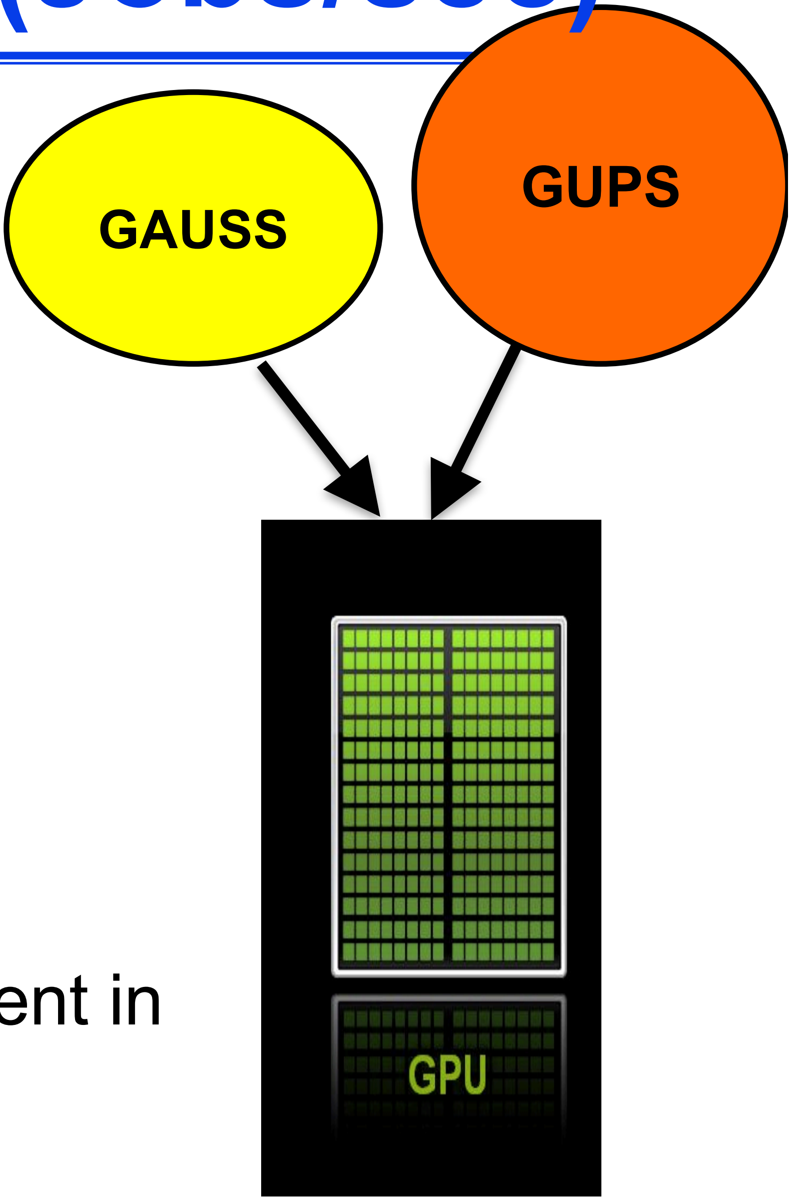


- HIST+DGEMM: 40% improvement in System throughput, over running alone

[Jog et al., GPGPU 2014]

System Throughput (Jobs/sec)



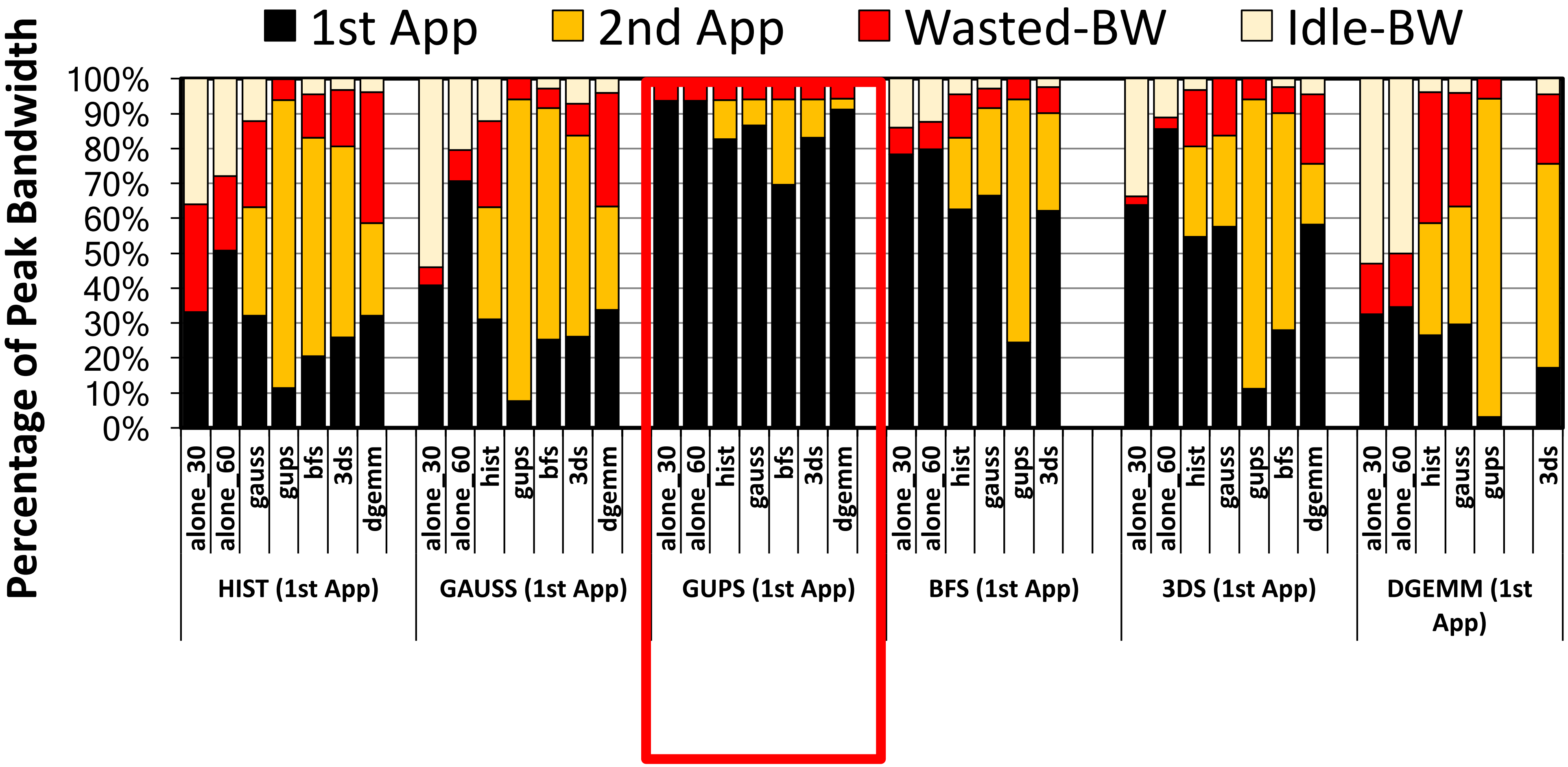


- GAUSS+GUPS: Only 2% improvement in

System throughput, over running alone

[Jog et al., GPGPU 2014]

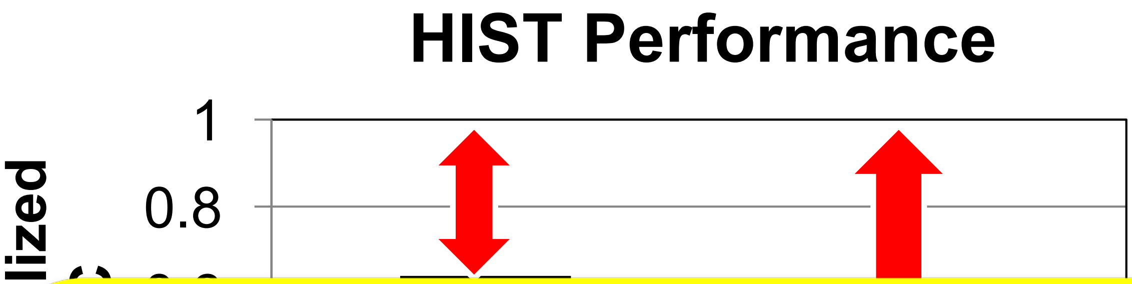
Memory Bandwidth Allocation

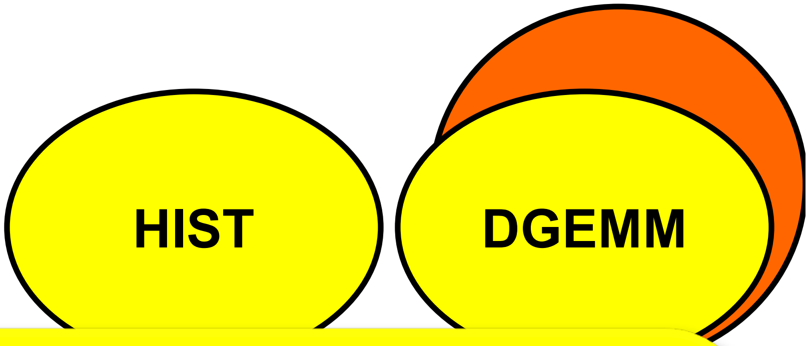


GUPS (Heavy Application) hurts other light applications

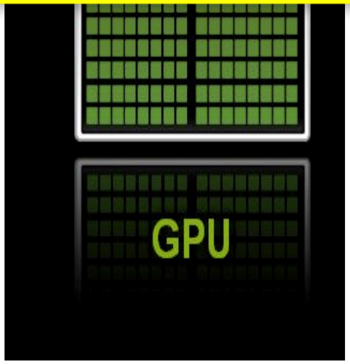
[Jog et al., GPGPU 2014]

Fairness





What is the best way to allocate bandwidth to different applications?



- Fairness problems in the system
  - Unequal performance impact

[Jog et al., GPGPU 2014]

1. Infrastructure Development

- Many existing CUDA applications do not employ "CUDAStreams" to enable multi-programmed execution

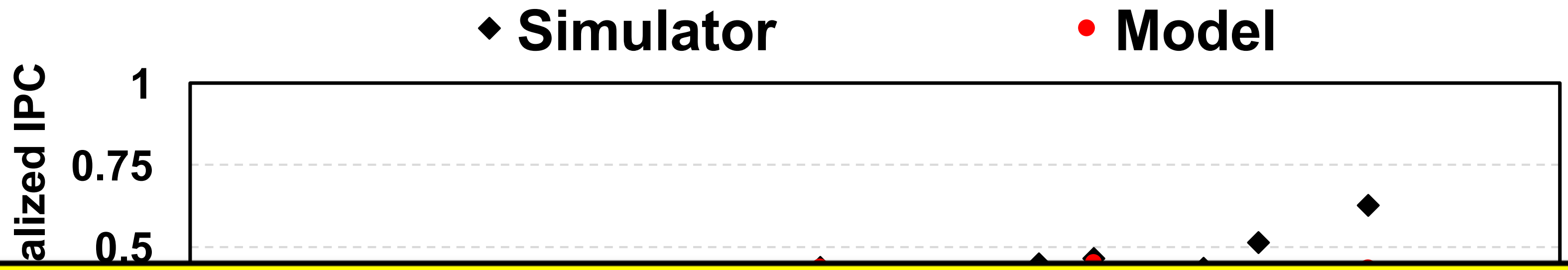
- Developed GPU concurrent application framework to enable multi-programming in GPUs

- Available at *https://github.com/adwaitjog/mafia*

[Jog et al., MEMSYS 2015]

2. Application Performance Modeling

$$\text{Performance} \quad \alpha \quad \frac{\text{Attained Bandwidth (BW)}}{\text{Misses Per Instruction (MPI)}}$$



How can we utilize this model to develop better memory scheduler?

is less than 10% averaged across 22 applications

[Jog et al., MEMSYS 2015]

Bandwidth Sharing Mechanisms

- Metric: Weighted Speedup (WS) = Sum of Slowdowns
- WS = $\frac{P_1}{P_1^{alone}} + \frac{P_2}{P_2^{alone}}$, where P α $\frac{BW}{MPI}$
- In order to have higher WS at t = t_B compared to at t = t_A , where at $t_B > t_A$, we give an additional Ɛ bandwidth to the first application by taking it from the other.

$$\boxed{\frac{P_1^B}{P_1^{alone}} + \frac{P_2^B}{P_2^{alone}} > \frac{P_1^A}{P_1^{alone}} + \frac{P_2^A}{P_2^{alone}}}$$

$$\boxed{\frac{\frac{BW_1 + \varepsilon}{MPI_1}}{\frac{BW_1^{alone}}{MPI_1}} + \frac{\frac{BW_2 - \varepsilon}{MPI_2}}{\frac{BW_2^{alone}}{MPI_2}} > \frac{\frac{BW_1}{MPI_1}}{\frac{BW_1^{alone}}{MPI_1}} + \frac{\frac{BW_2}{MPI_2}}{\frac{BW_2^{alone}}{MPI_2}}}$$

- Prioritize the application with the *least BW (alone)* to optimize for weighted speedup
- In the paper, we show that prioritizing the application with the least attained bandwidth can improve weighted speedup

[Jog et al., MEMSYS 2015]

Results

- ❑ Misses Per Instruction (MPI) Metric is not a good proxy for GPU performance

- ❑ Attained Bandwidth (BW) and Misses Per Instruction (MPI) metrics can drive memory scheduling decisions for better throughput and fairness.

- ❑ 10% improvement in weighted speedup and fairness over 25 representative 2-app workloads

- ❑ More results: Scalability; Application to Core Mapping Mechanisms.

[Jog et al., MEMSYS 2015]

Conclusions

- Data Movement and Bandwidth are Major Bottlenecks.

- Three issues we discussed today:
  - High cache miss-rates → *warp scheduling!*
  - Bandwidth is critical → *data compression!*
  - Sub-optimal memory bandwidth allocation → *memory scheduling!*

- Other avenues and directions?
  - Processing Near/In Memory (PIM)
  - Value Prediction and Approximations

Other Sophisticated Mechanisms

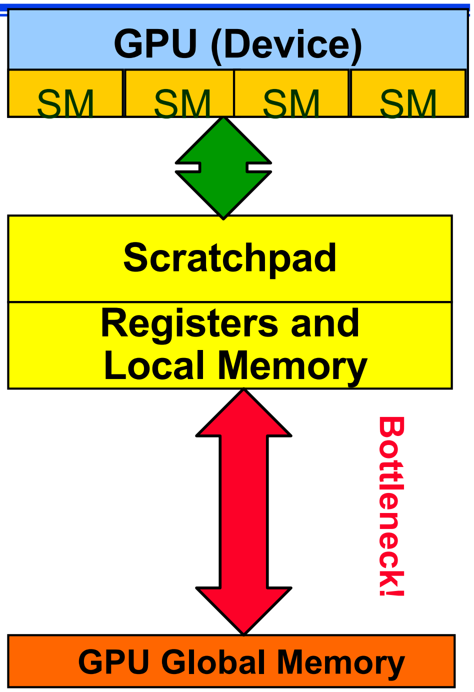
- Wang et al., Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management, HPCA'18

- Park et al., Dynamic Resource Management for Efficient Utilization of Multitasking GPUs, ASPLOS'17

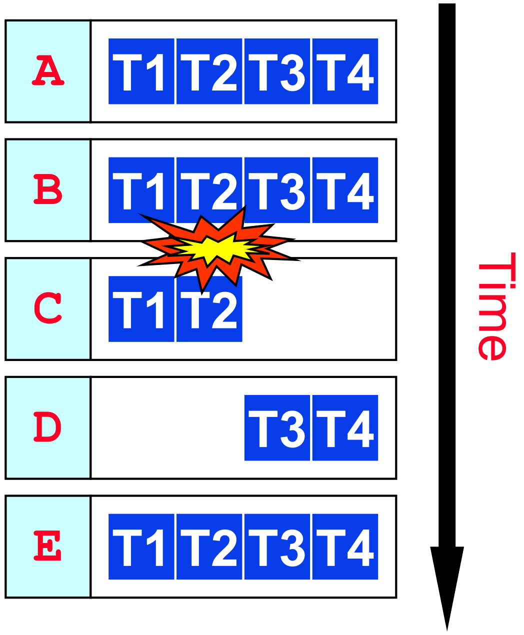
- Xu et al., Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming, ISCA'16

Key GPU Performance Concerns

Memory Concerns: Data transfers between SMs and global memory are costly.



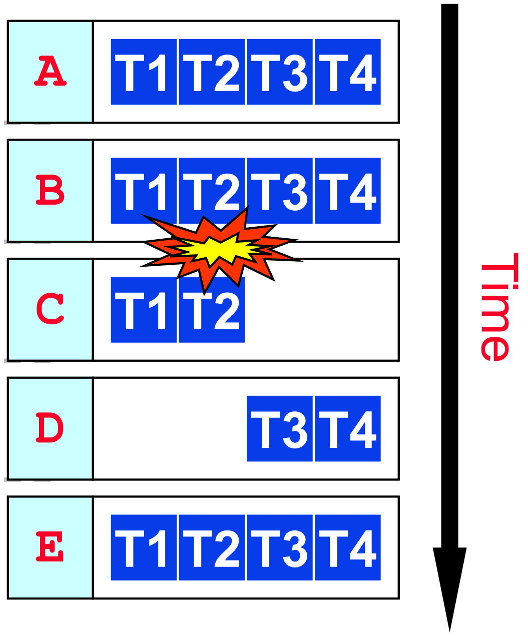
Compute Concerns: Threads that do not take the same *control* path lead to serialization in the GPU compute pipeline.



Compute Concerns

- Challenge: How to handle branch operations when different threads in a warp follow a different path through program?
- Solution: Serialize different paths.

```
foo[] = {4,8,12,16};
 A: v = foo[threadIdx.x];
 B: if (v < 10)
 C:     v = 0;
    else
 D:     v = 10;
 E: w = bar[threadIdx.x]+v;
```



Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others

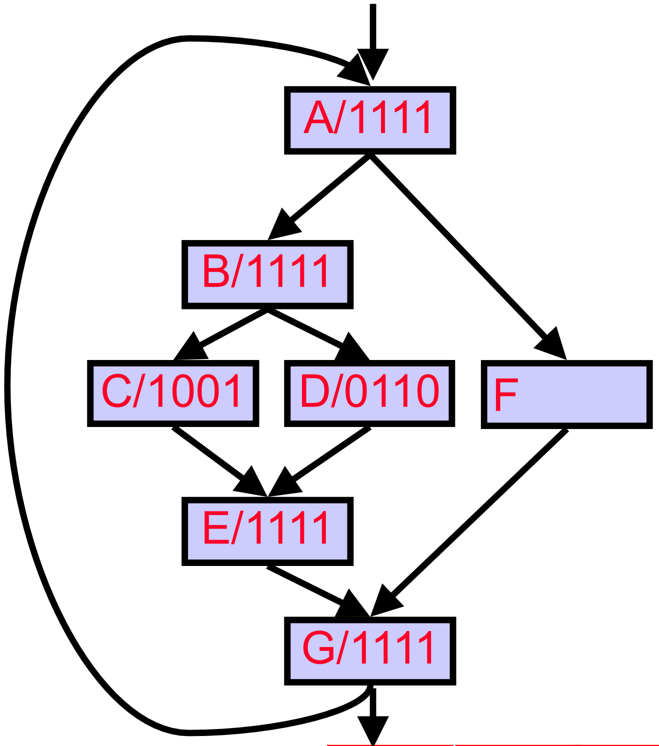
- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices

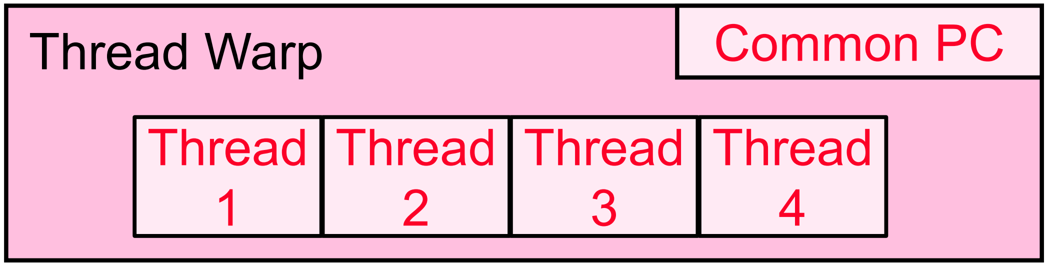
- Example kernel statement with divergence:
  - if (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

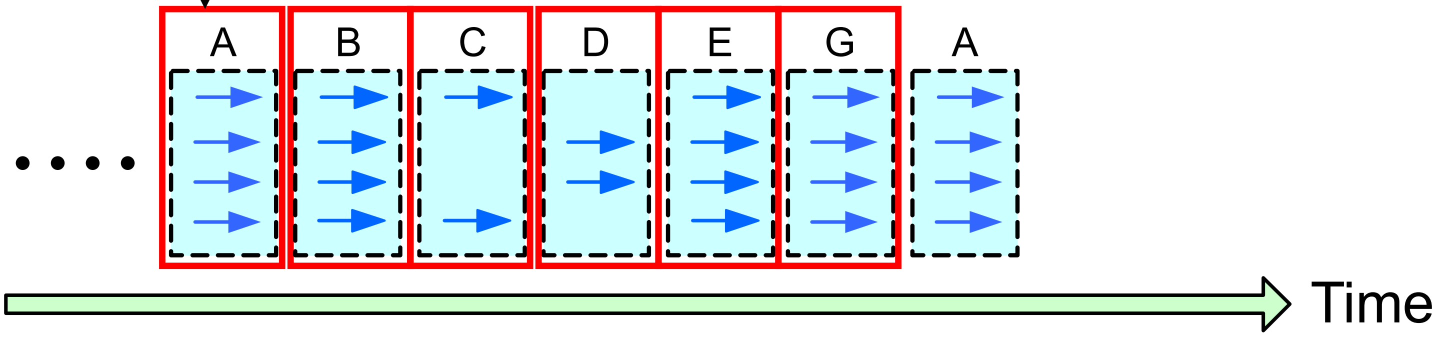
SIMT Hardware Stack



Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001

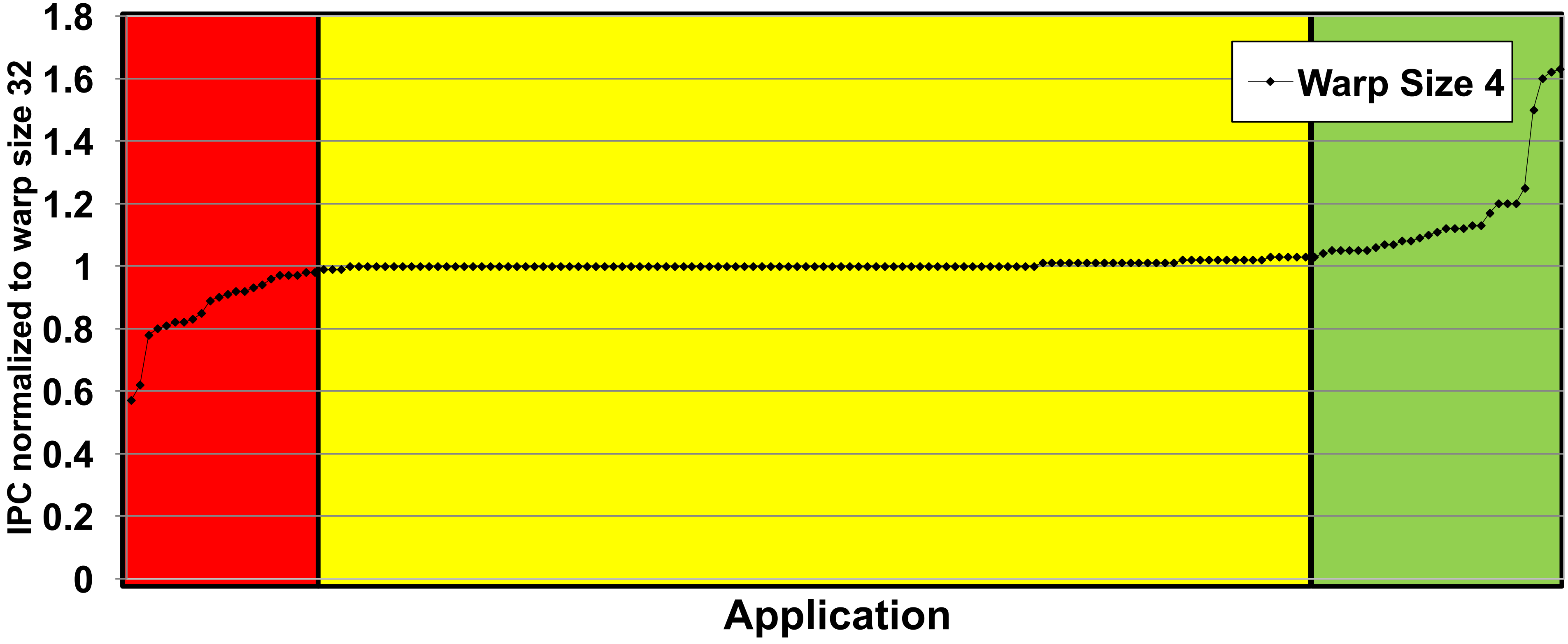




Potential for significant loss of throughput when control flow diverged!

Performance vs. Warp Size

- 165 Applications

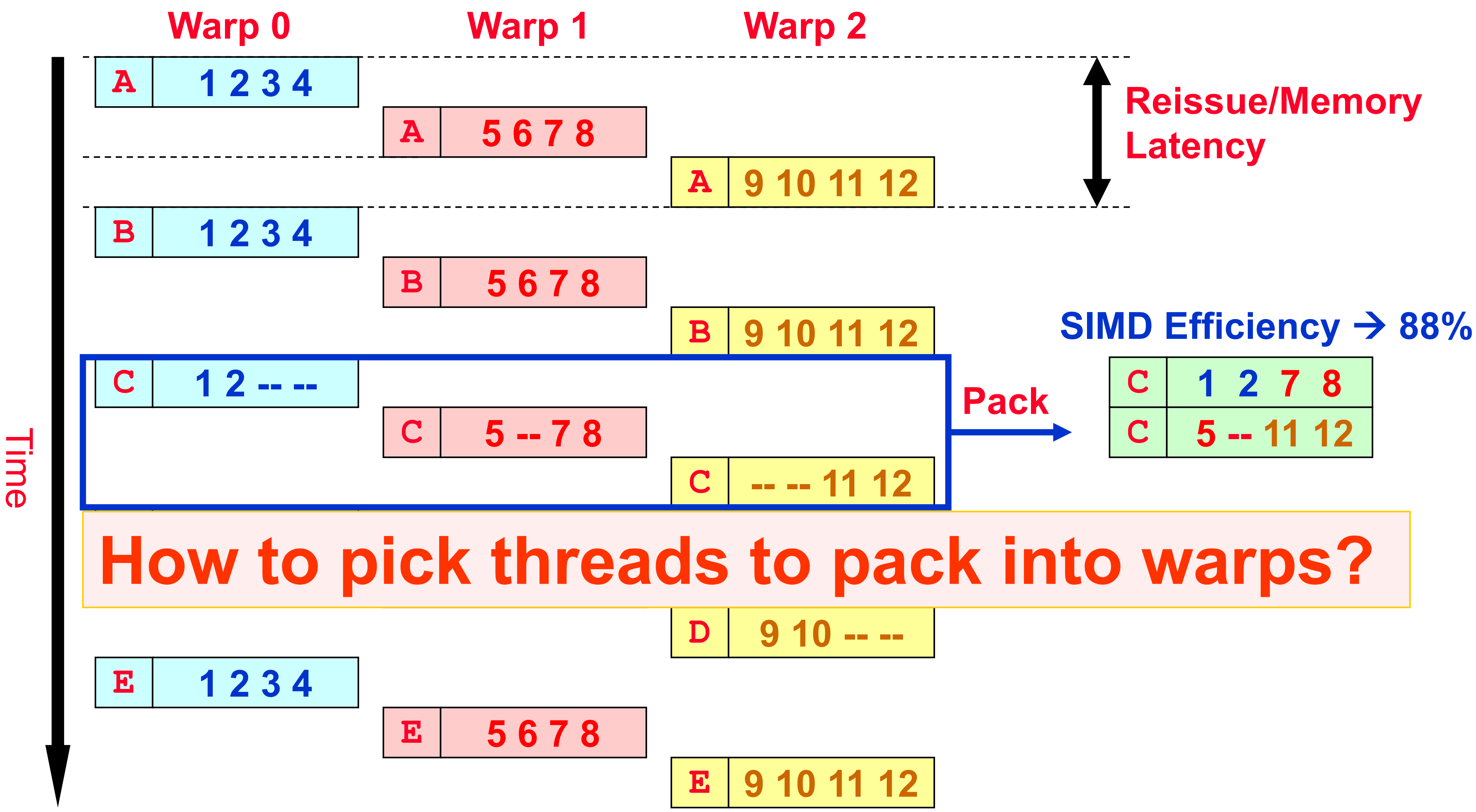


Convergent Applications

Warp-Size Insensitive Applications

Divergent Applications

Dynamic Warp Formation
(Fung MICRO’07)



How to pick threads to pack into warps?

More References

- 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”.