

# Staged Memory Scheduling

**Rachata Ausavarungnirun**, Kevin Chang, Lavanya Subramanian,  
Gabriel H. Loh\*, Onur Mutlu

Carnegie Mellon University, \*AMD Research  
June 12<sup>th</sup> 2012

**SAFARI**

**Carnegie Mellon**

**AMD** 

# Executive Summary

---

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with **large request buffers**
- **Problem:** Existing monolithic application-aware memory scheduler designs are **hard to scale** to large request buffer sizes
- **Solution:** Staged Memory Scheduling (SMS)  
**decomposes the memory controller into three simple stages:**
  - 1) Batch formation: maintains row buffer locality
  - 2) Batch scheduler: reduces interference between applications
  - 3) DRAM command scheduler: issues requests to DRAM
- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

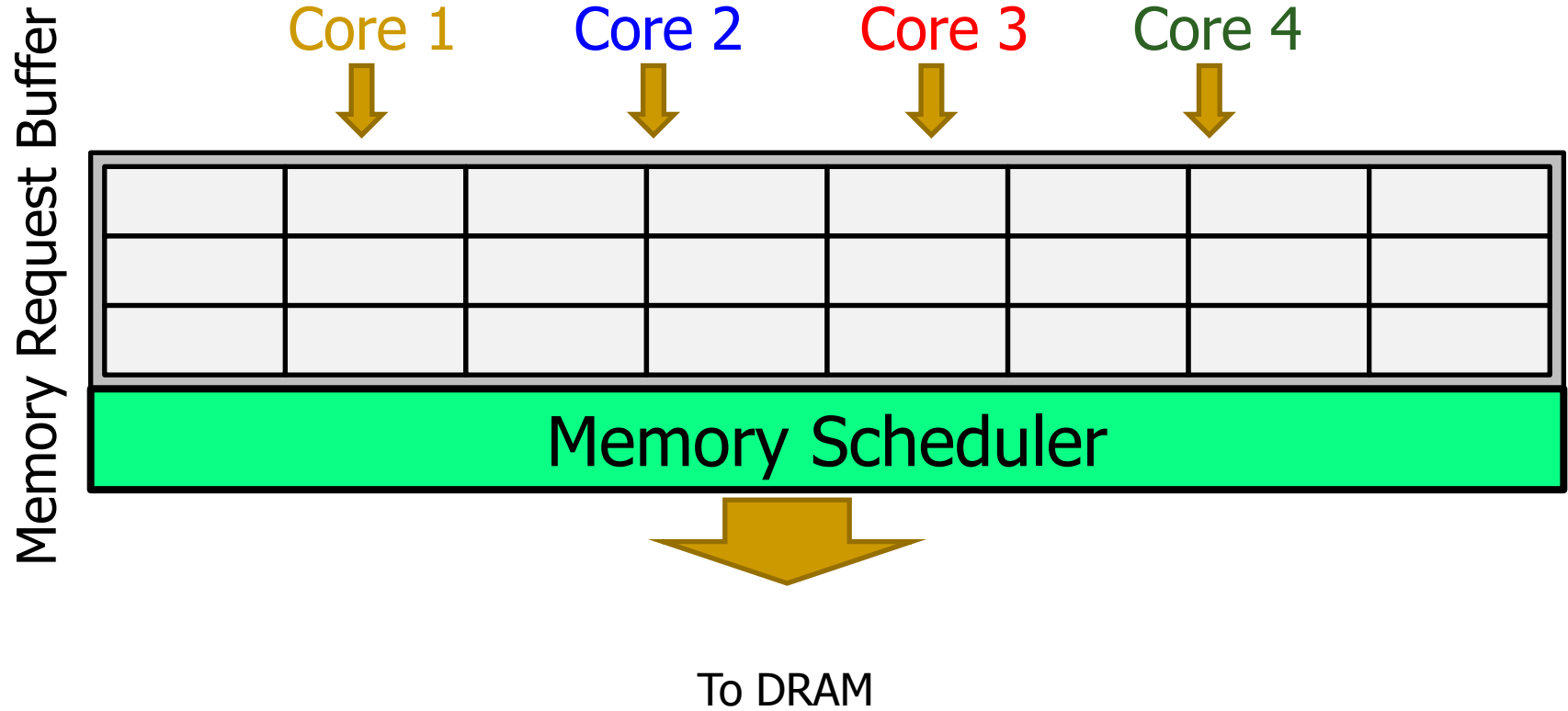
# Outline

---

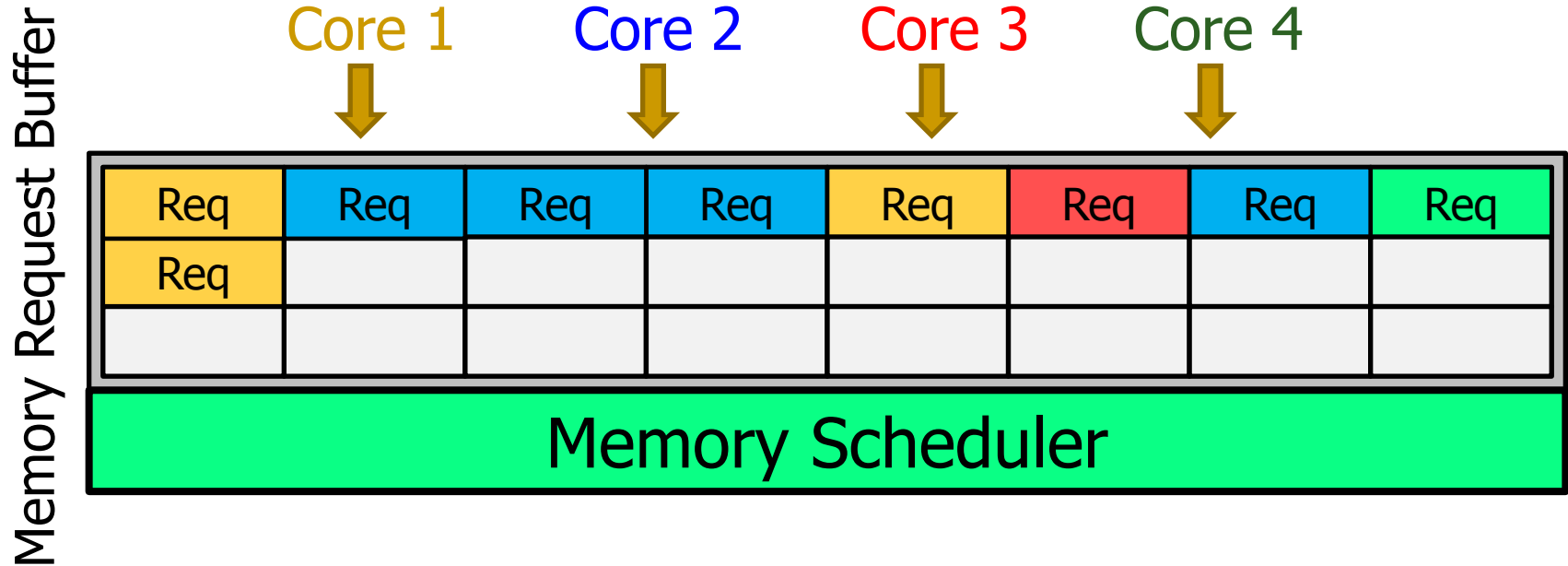
- Background
- Motivation
- Our Goal
- Observations
- Staged Memory Scheduling
  - 1) Batch Formation
  - 2) Batch Scheduler
  - 3) DRAM Command Scheduler
- Results
- Conclusion

# Main Memory is a Bottleneck

---



# Main Memory is a Bottleneck



- All cores contend for limited off-chip bandwidth
  - Inter-application interference **degrades system performance**
  - The memory scheduler can help mitigate the problem
- How does the memory scheduler deliver good performance and fairness?

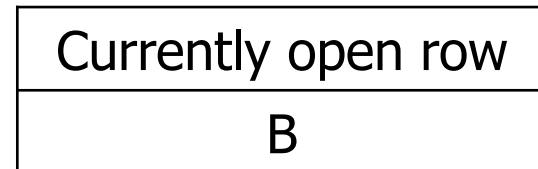
# Three Principles of Memory Scheduling

---

# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth



# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth

Req 1	Row A
Req 2	Row B
Req 3	Row C
Req 4	Row A
Req 5	Row B

Currently open row
B

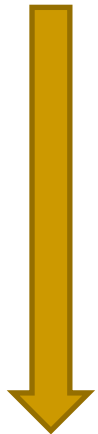


# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth

Older



Newer

Req 1	Row A
Req 2	Row B
Req 3	Row C
Req 4	Row A
Req 5	Row B

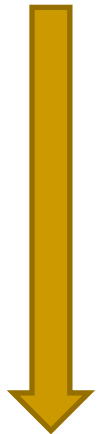
Currently open row
B

# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth

Older



Newer

Req 1	Row A
Req 2	Row B
Req 3	Row C
Req 4	Row A
Req 5	Row B

Currently open row
B

# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth
- Prioritize latency-sensitive applications [Kim+, HPCA'10]
  - To maximize system throughput

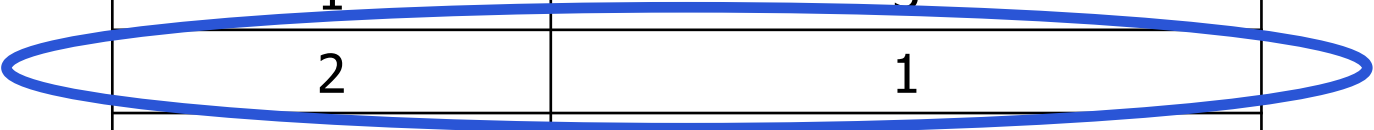
Application	Memory Intensity (MPKI)
1	5
2	1
3	2
4	10

# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth
- Prioritize latency-sensitive applications [Kim+, HPCA'10]
  - To maximize system throughput

Application	Memory Intensity (MPKI)
1	5
2	1
3	2
4	10



# Three Principles of Memory Scheduling

---

- Prioritize row-buffer-hit requests [Rixner+, ISCA'00]
  - To maximize memory bandwidth
- Prioritize latency-sensitive applications [Kim+, HPCA'10]
  - To maximize system throughput
- Ensure that no application is starved [Mutlu and Moscibroda, MICRO'07]
  - To minimize unfairness

# Outline

---

- Background
- **Motivation: CPU-GPU Systems**
- Our Goal
- Observations
- Staged Memory Scheduling
  - 1) Batch Formation
  - 2) Batch Scheduler
  - 3) DRAM Command Scheduler
- Results
- Conclusion

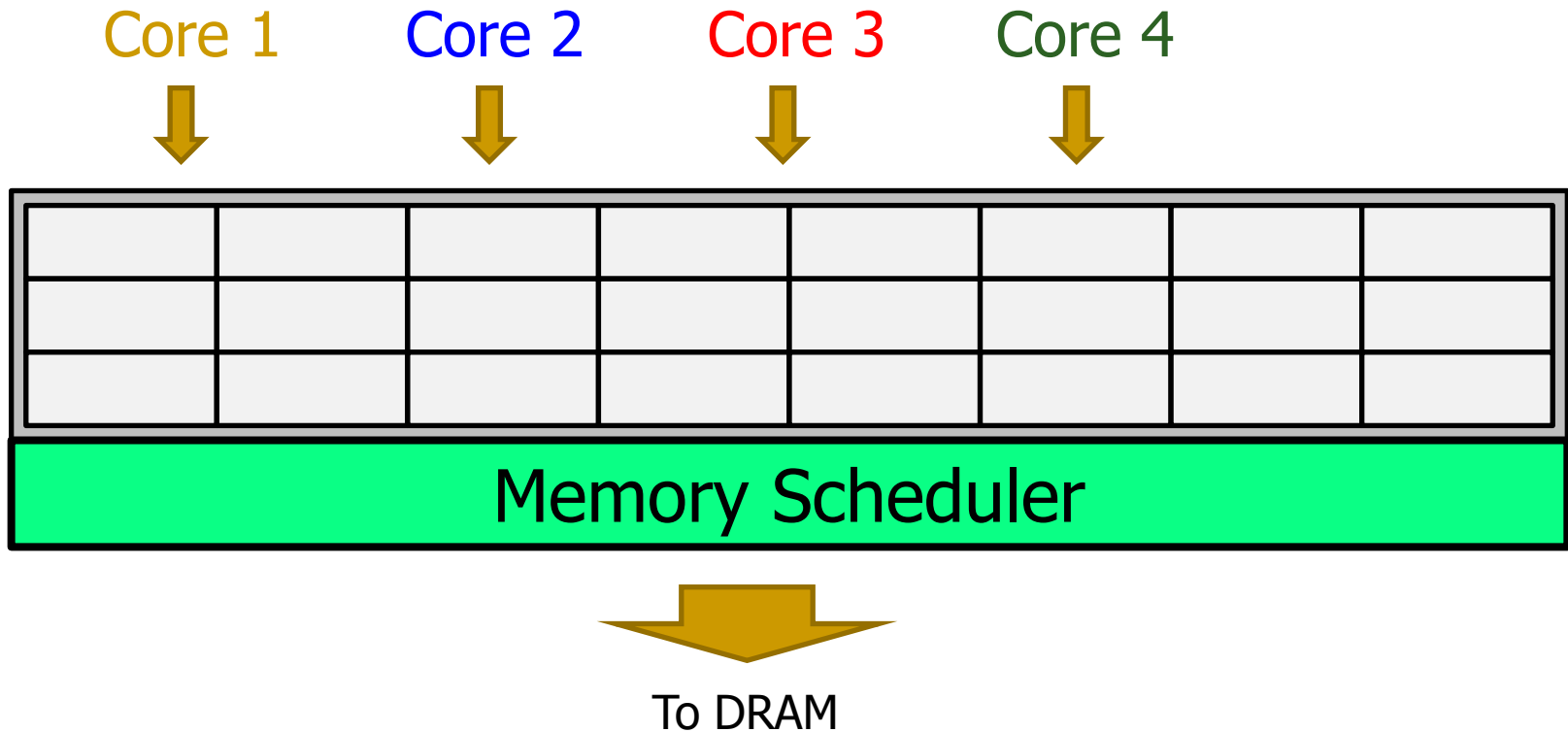
# Memory Scheduling for CPU-GPU Systems

---

- Current and future systems integrate a GPU along with multiple cores
- GPU shares the main memory with the CPU cores
- GPU is **much more (4x-20x) memory-intensive** than CPU
- How should memory scheduling be done when GPU is integrated on-chip?

# Introducing the GPU into the System

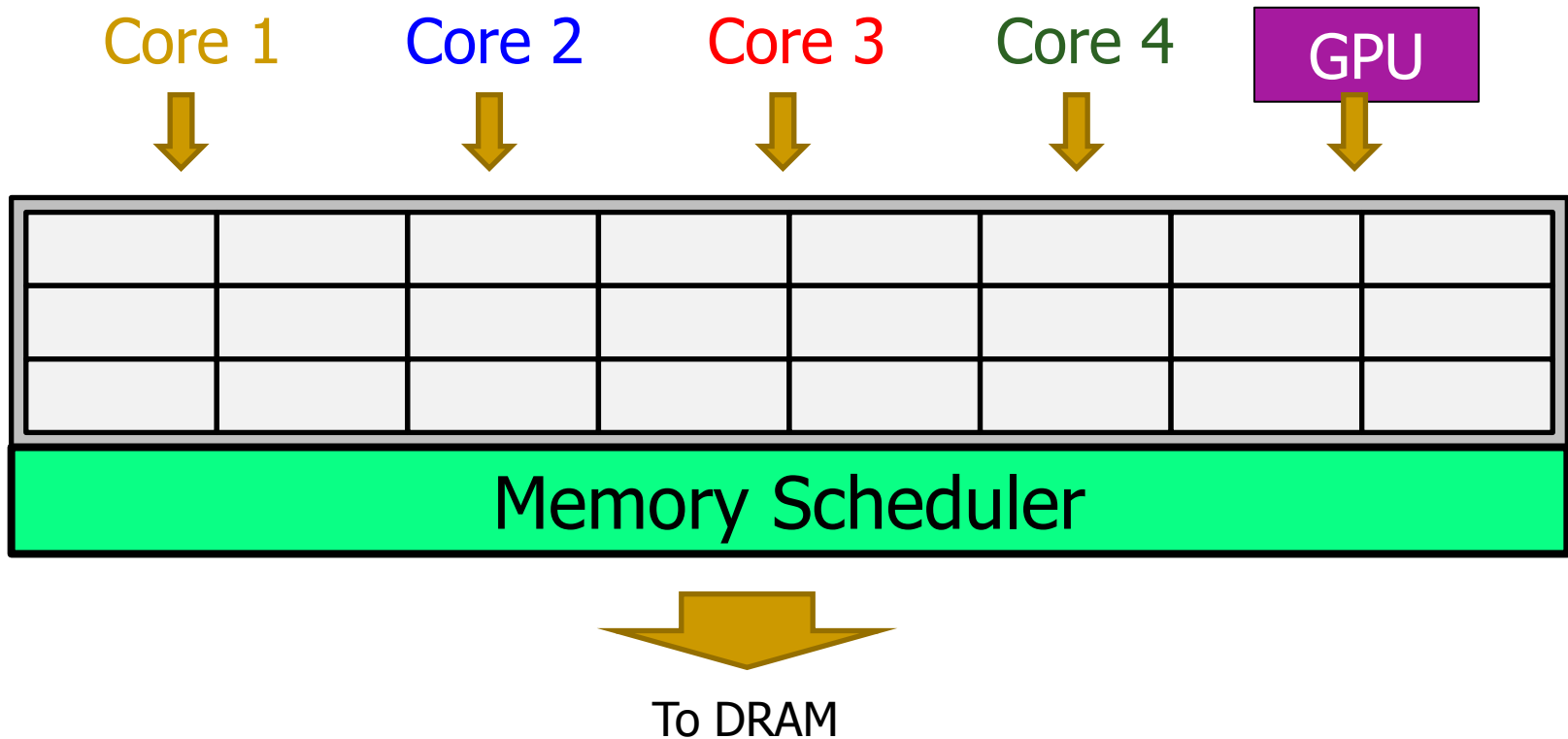
---



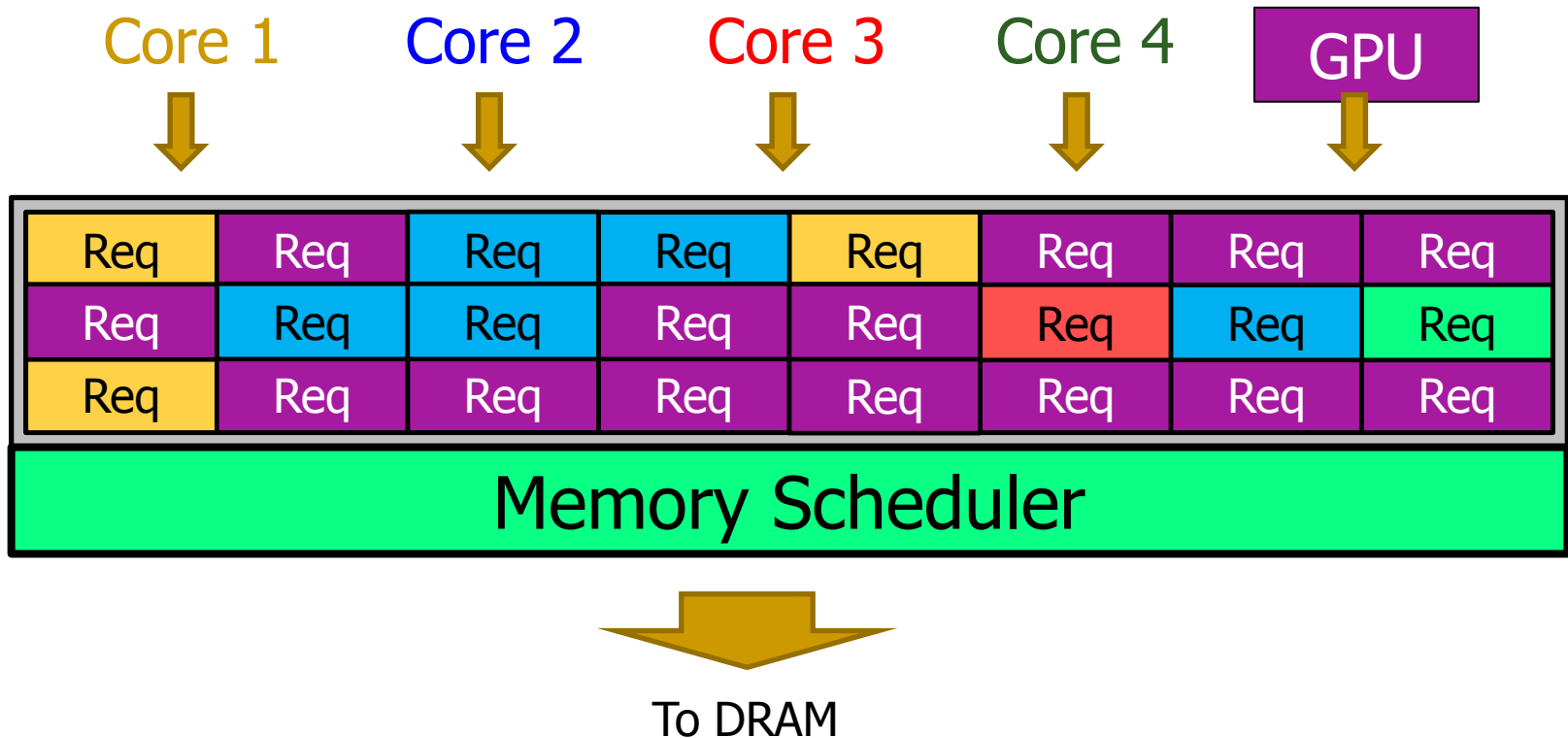


# Introducing the GPU into the System

---



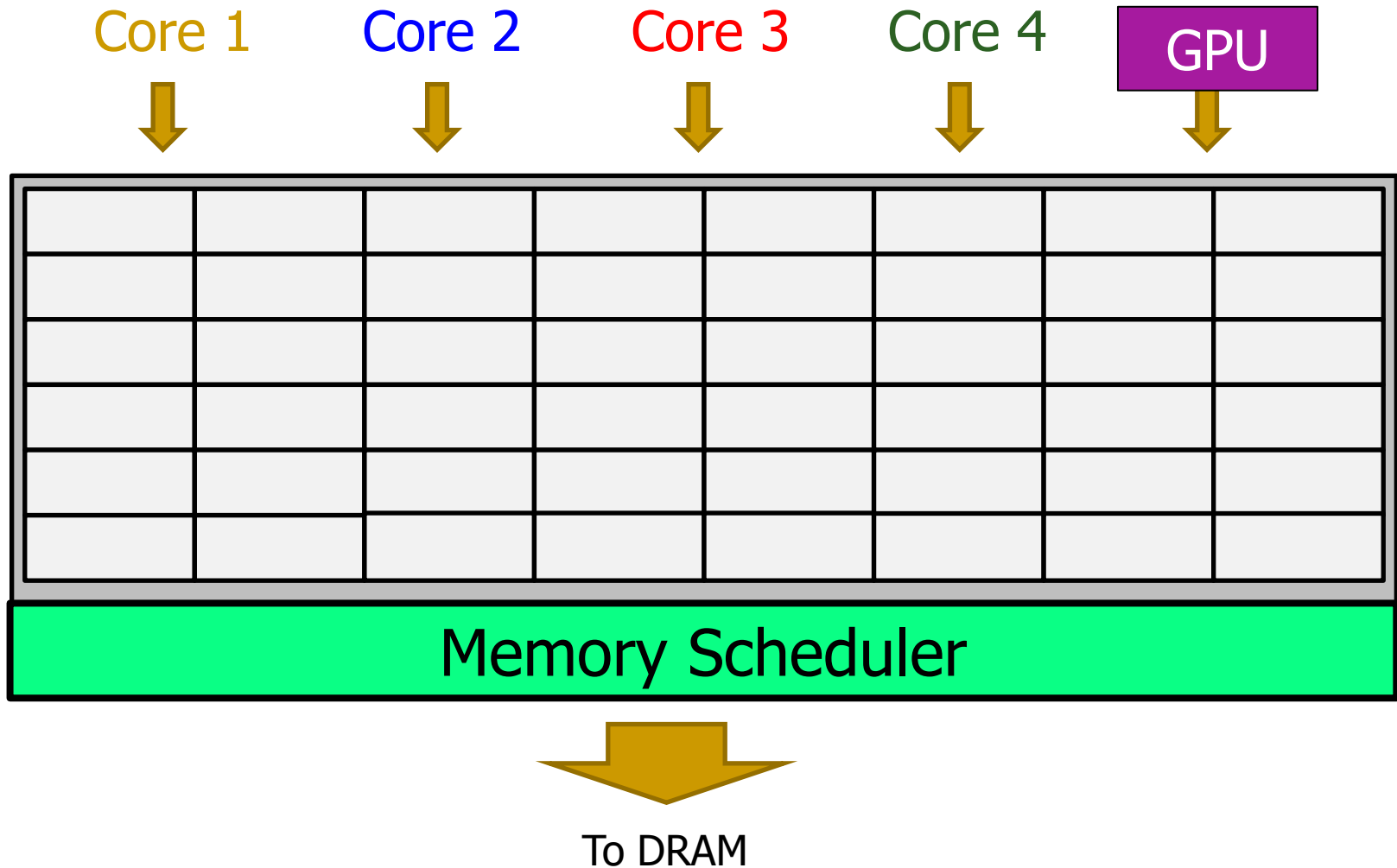
# Introducing the GPU into the System



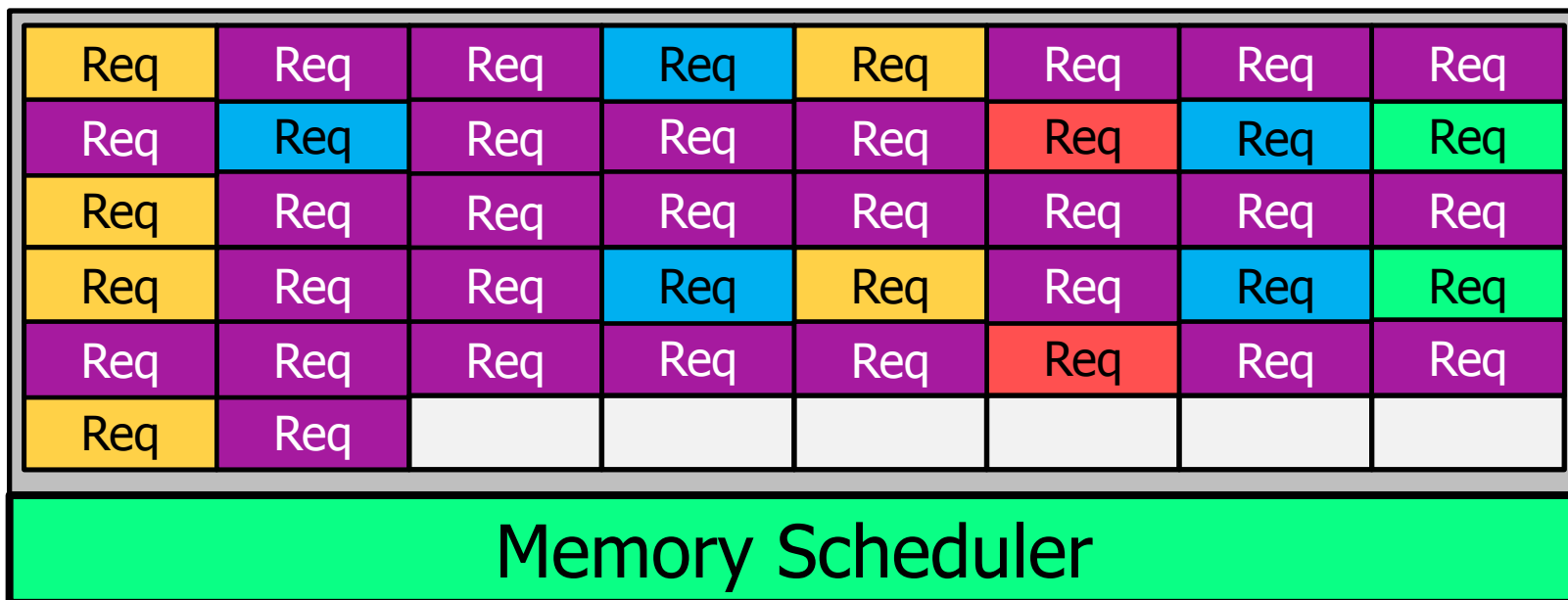
- GPU occupies a significant portion of the request buffers
  - Limits the MC's visibility of the CPU applications' differing memory behavior → can lead to a **poor scheduling decision**

# Naïve Solution: Large Monolithic Buffer

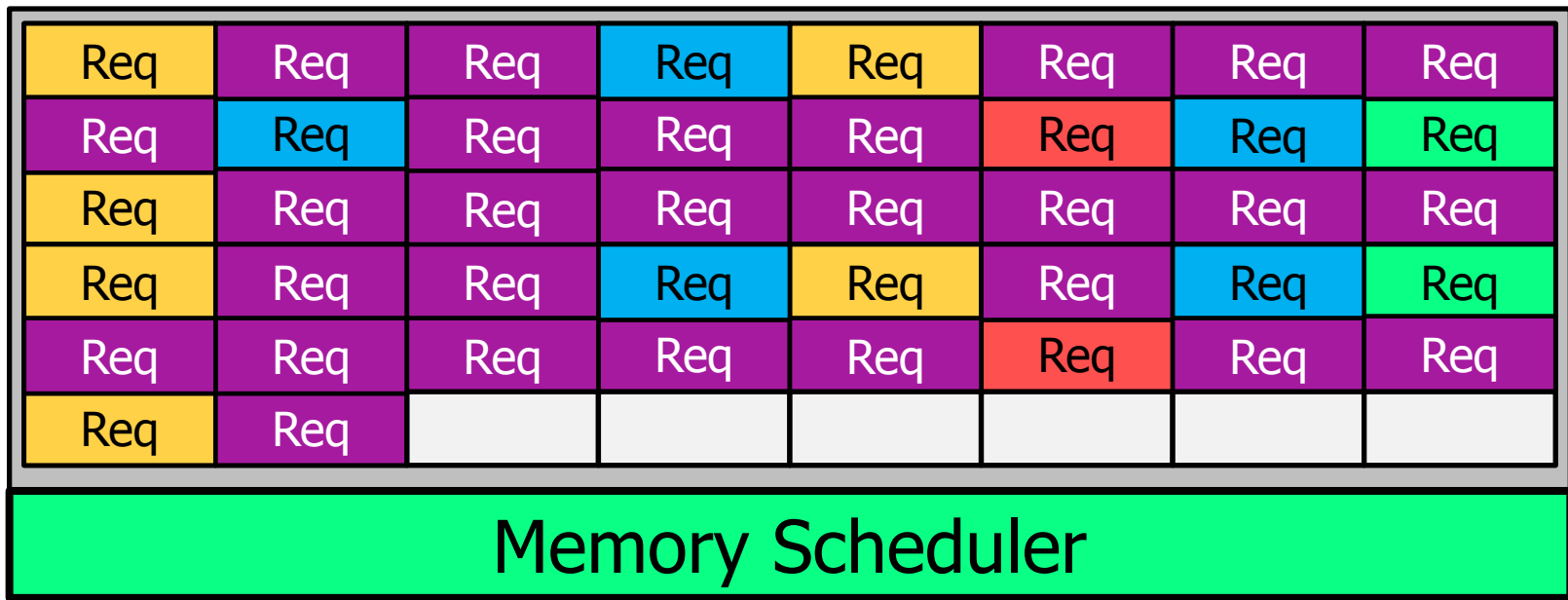
---



# Problems with Large Monolithic Buffer



# Problems with Large Monolithic Buffer



- A large buffer requires more complicated logic to:
  - Analyze memory requests (e.g., determine row buffer hits)
  - Analyze application characteristics
  - Assign and enforce priorities
- This leads to high complexity, high power, large die area

# Problems with Large Monolithic Buffer

---

Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req						

More Complex Memory Scheduler

# Our Goal

---

- Design a new memory scheduler that is:
  - **Scalable** to accommodate a large number of requests
  - **Easy to implement**
  - **Application-aware**
  - Able to provide high **performance and fairness**, especially in heterogeneous CPU-GPU systems

# Outline

---

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- **Observations**
- Staged Memory Scheduling
  - 1) Batch Formation
  - 2) Batch Scheduler
  - 3) DRAM Command Scheduler
- Results
- Conclusion



# Key Functions of a Memory Controller

---

- Memory controller must consider three different things concurrently when choosing the next request:
  - 1) Maximize row buffer hits
    - Maximize memory bandwidth
  - 2) Manage contention between applications
    - Maximize system throughput and fairness
  - 3) Satisfy DRAM timing constraints
- Current systems use a **centralized memory controller design** to accomplish these functions
  - **Complex, especially with large request buffers**

# Key Idea: Decouple Tasks into Stages

---

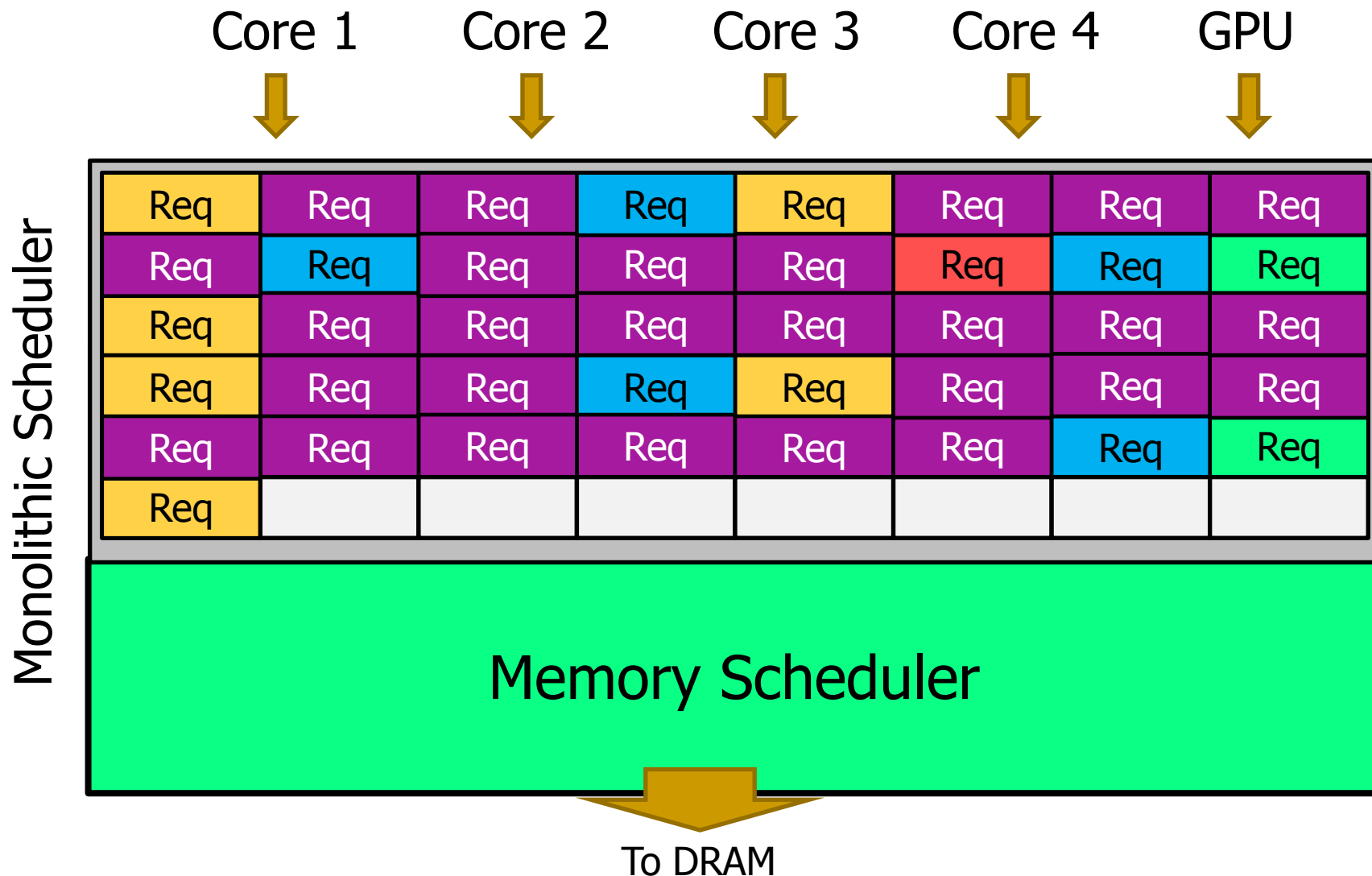
- Idea: **Decouple the functional tasks** of the memory controller
  - Partition tasks across several simpler HW structures (stages)
- 1) Maximize row buffer hits
  - **Stage 1: Batch formation**
  - Within each application, **groups requests to the same row into batches**
- 2) Manage contention between applications
  - **Stage 2: Batch scheduler**
  - **Schedules batches from different applications**
- 3) Satisfy DRAM timing constraints
  - **Stage 3: DRAM command scheduler**
  - **Issues requests** from the already-scheduled order to each bank

# Outline

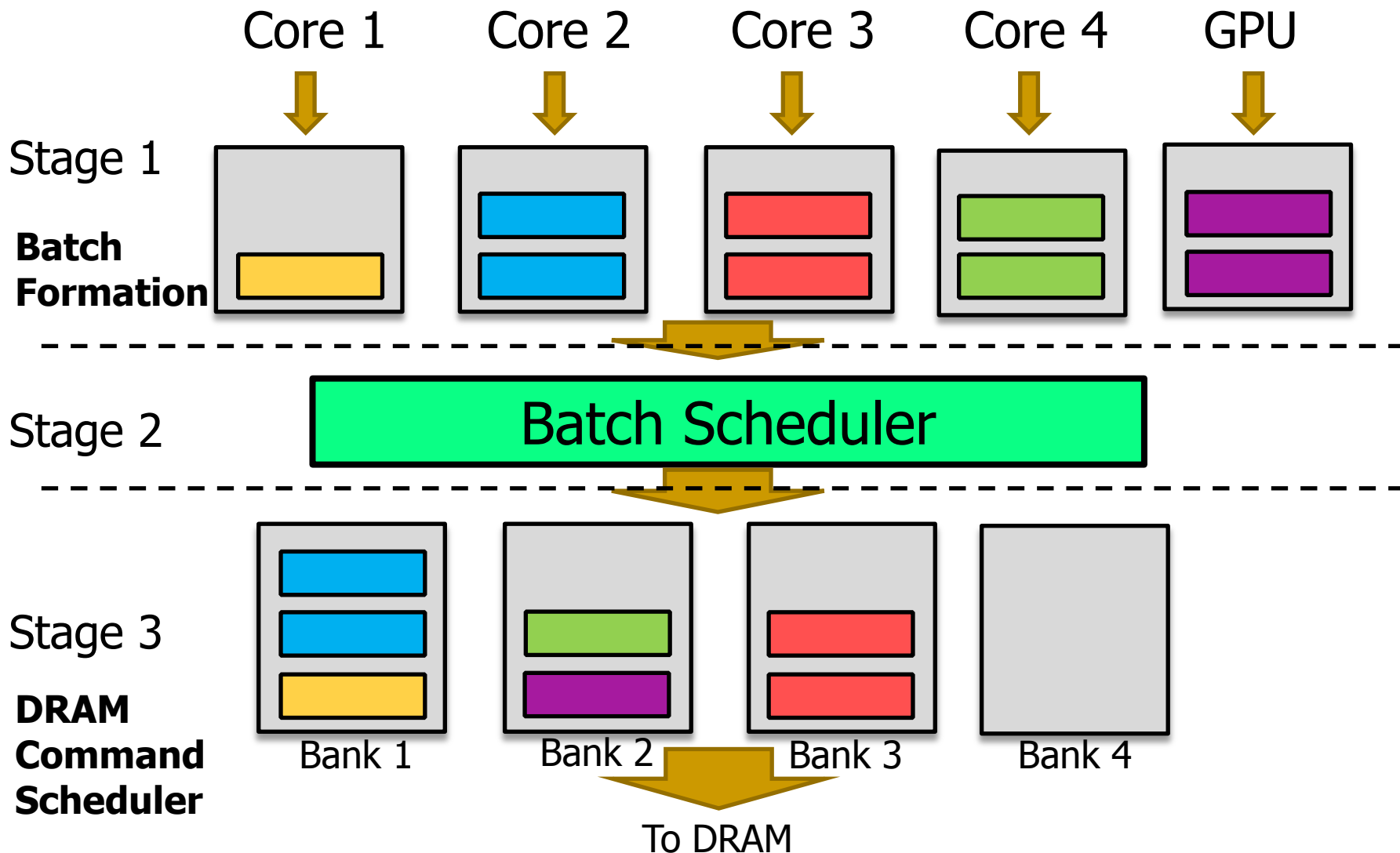
---

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  - 1) Batch Formation
  - 2) Batch Scheduler
  - 3) DRAM Command Scheduler
- Results
- Conclusion

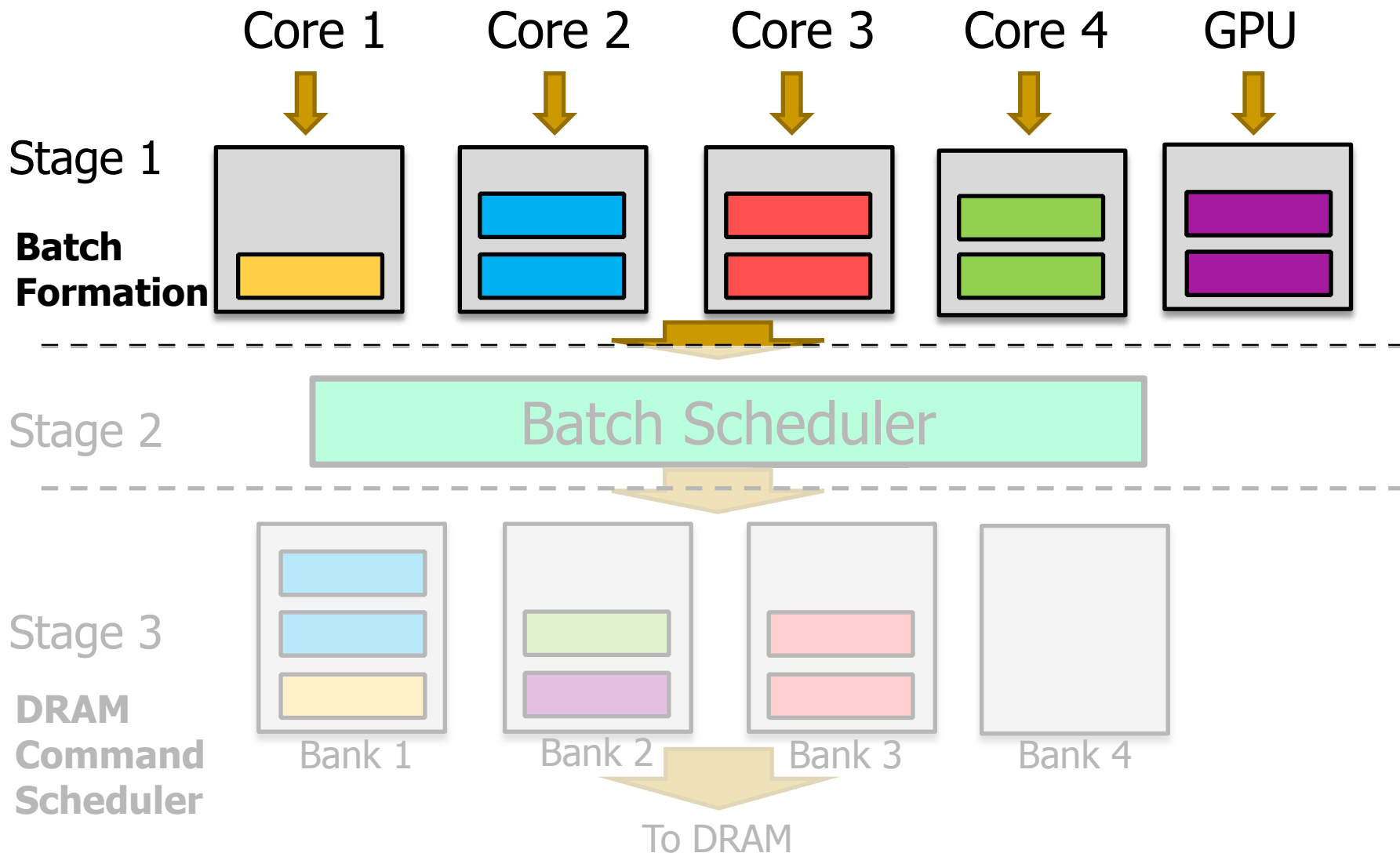
# SMS: Staged Memory Scheduling



# SMS: Staged Memory Scheduling



# SMS: Staged Memory Scheduling



# Stage 1: Batch Formation

---

- Goal: **Maximize row buffer hits**
- At each core, we want to **batch requests that access the same row** within a **limited time window**
- A batch is ready to be scheduled under two conditions
  - 1) When the next request accesses a different row
  - 2) When the time window for batch formation expires
- Keep this stage simple by using **per-core FIFOs**

# Stage 1: Batch Formation Example

---

Stage 1

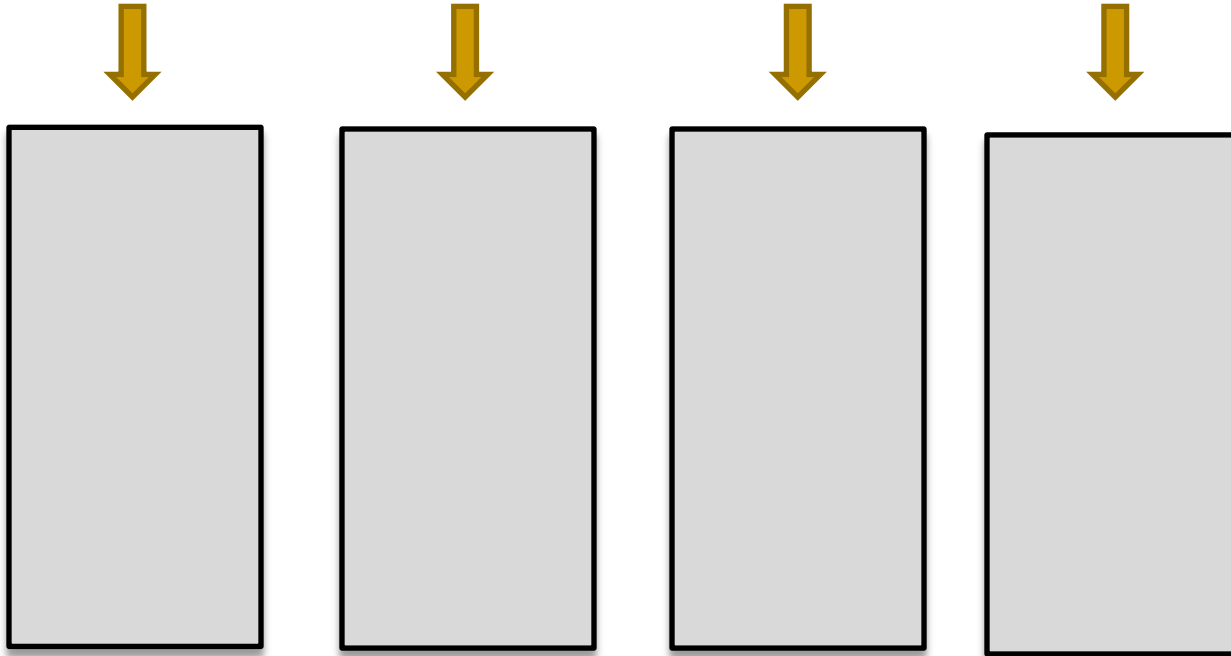
Core 1

Core 2

Core 3

Core 4

**Batch  
Formation**



---

To Stage 2 (Batch Scheduling)

---

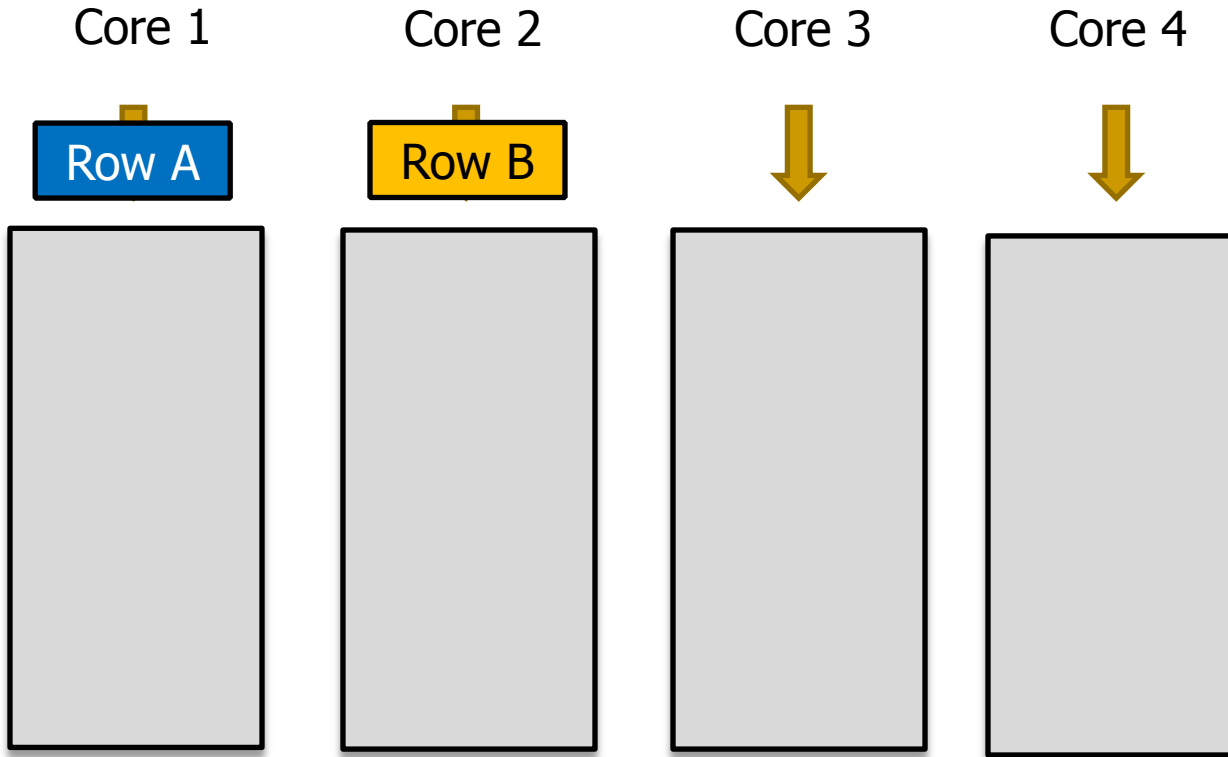


# Stage 1: Batch Formation Example

---

Stage 1

**Batch  
Formation**

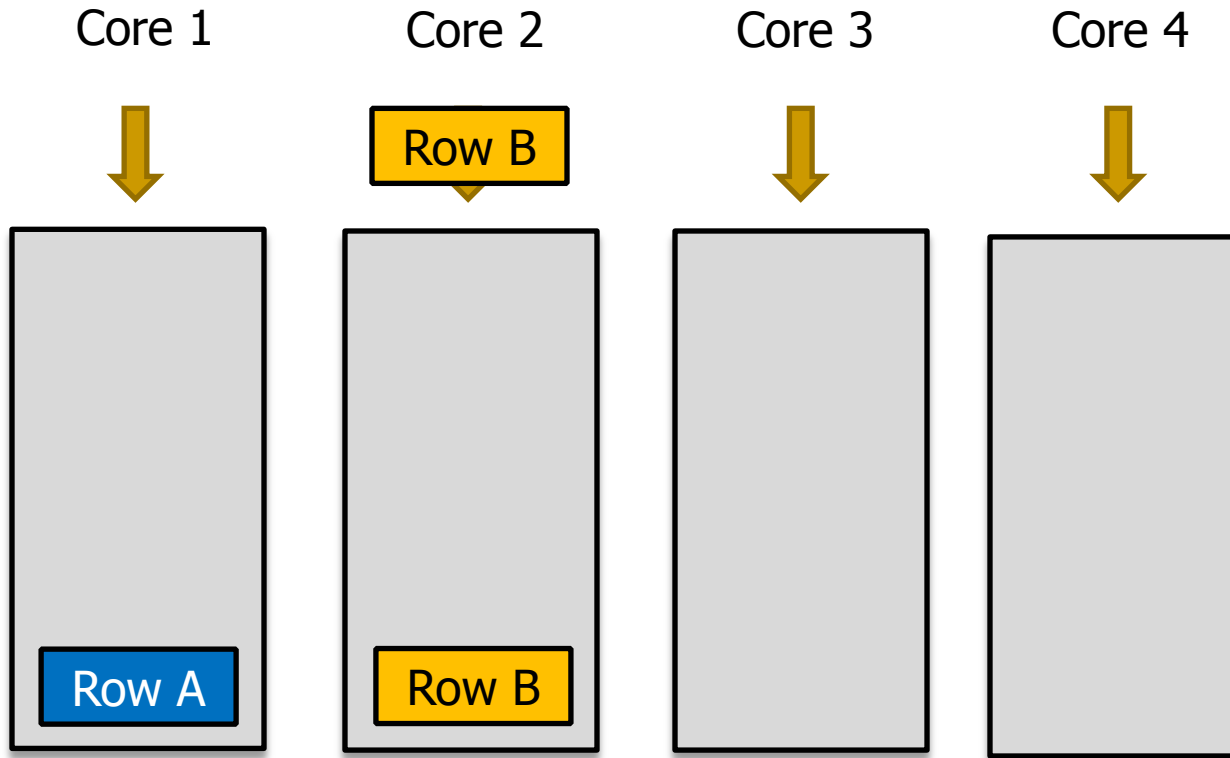


# Stage 1: Batch Formation Example

---

Stage 1

**Batch Formation**



To Stage 2 (Batch Scheduling)

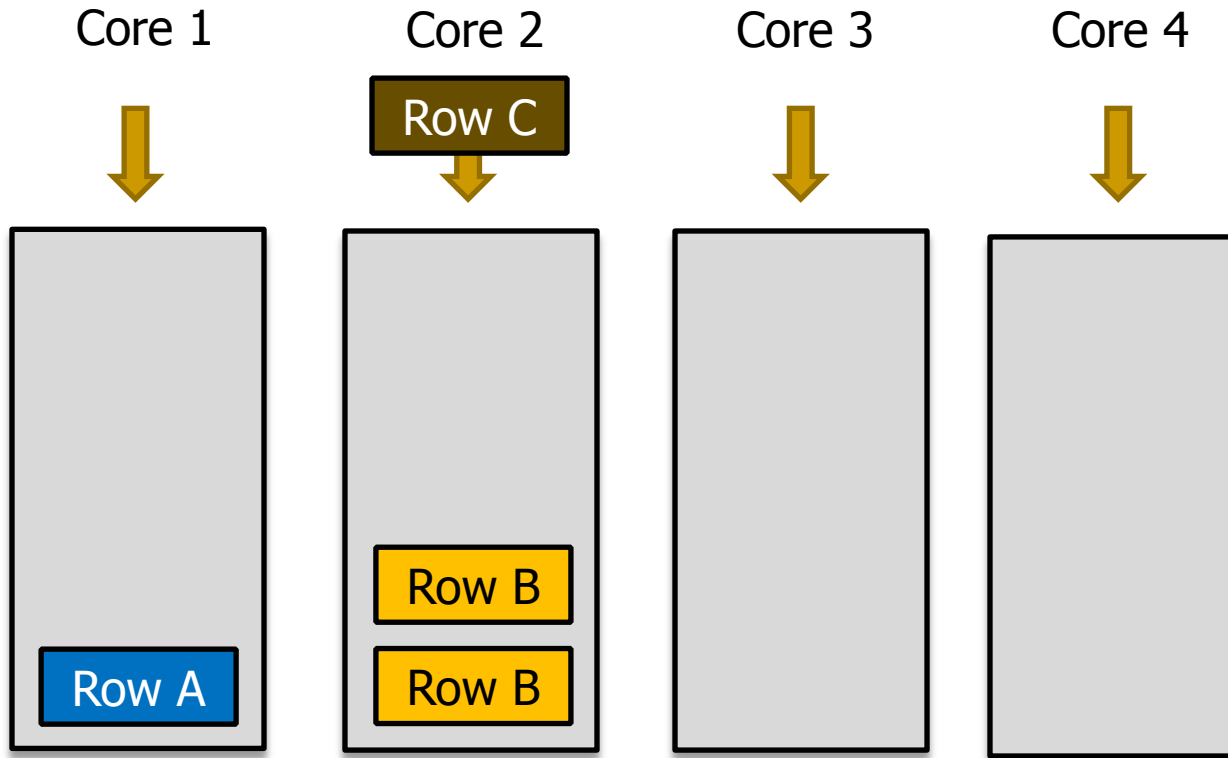
---

# Stage 1: Batch Formation Example

---

Stage 1

**Batch  
Formation**



To Stage 2 (Batch Scheduling)

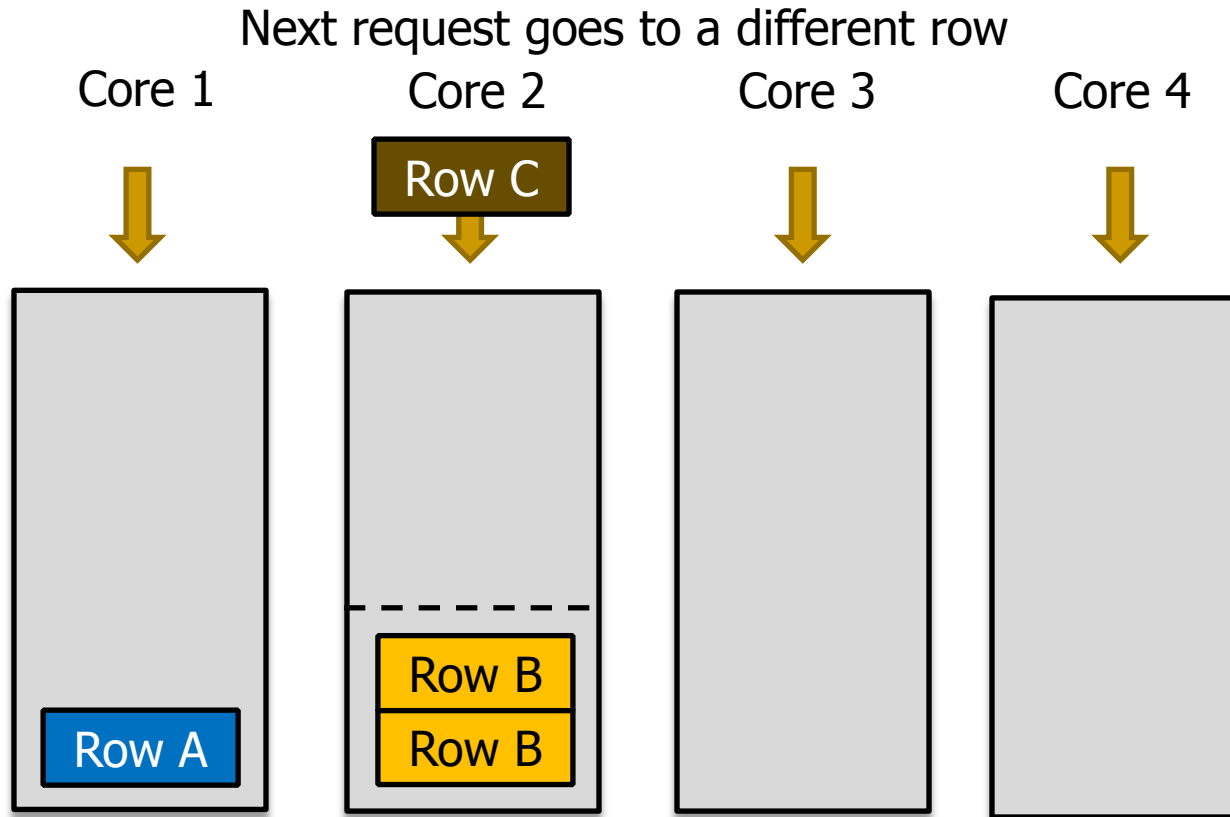
---

# Stage 1: Batch Formation Example

---

Stage 1

**Batch  
Formation**



To Stage 2 (Batch Scheduling)

---

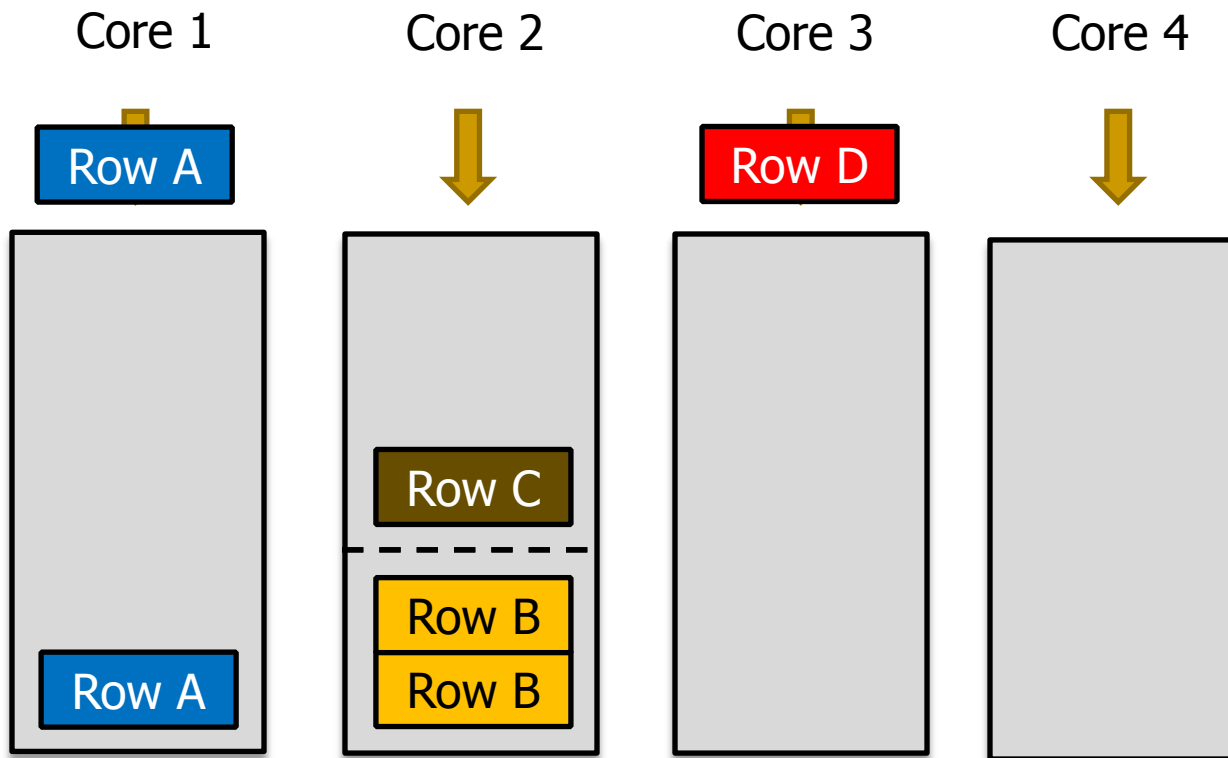
# Stage 1: Batch Formation Example

---

Stage 1

Next request goes to a different row

**Batch Formation**



To Stage 2 (Batch Scheduling)

---

# Stage 1: Batch Formation Example

---

Stage 1

Next request goes to a different row

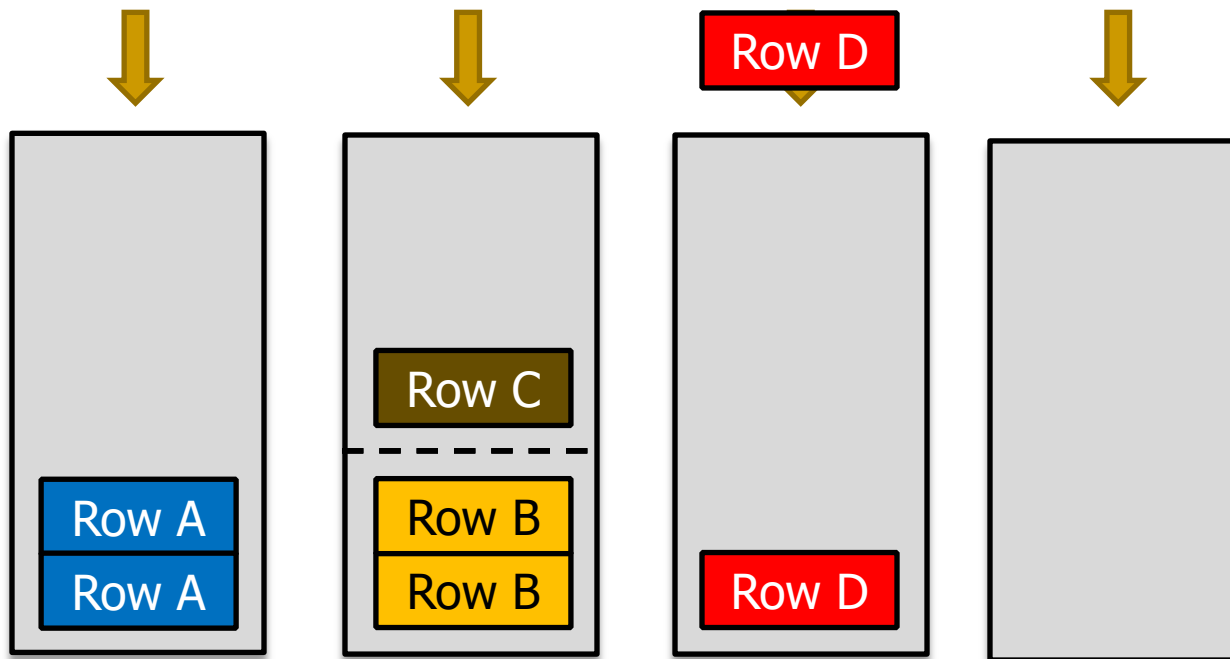
Core 1

Core 2

Core 3

Core 4

**Batch Formation**



To Stage 2 (Batch Scheduling)

---

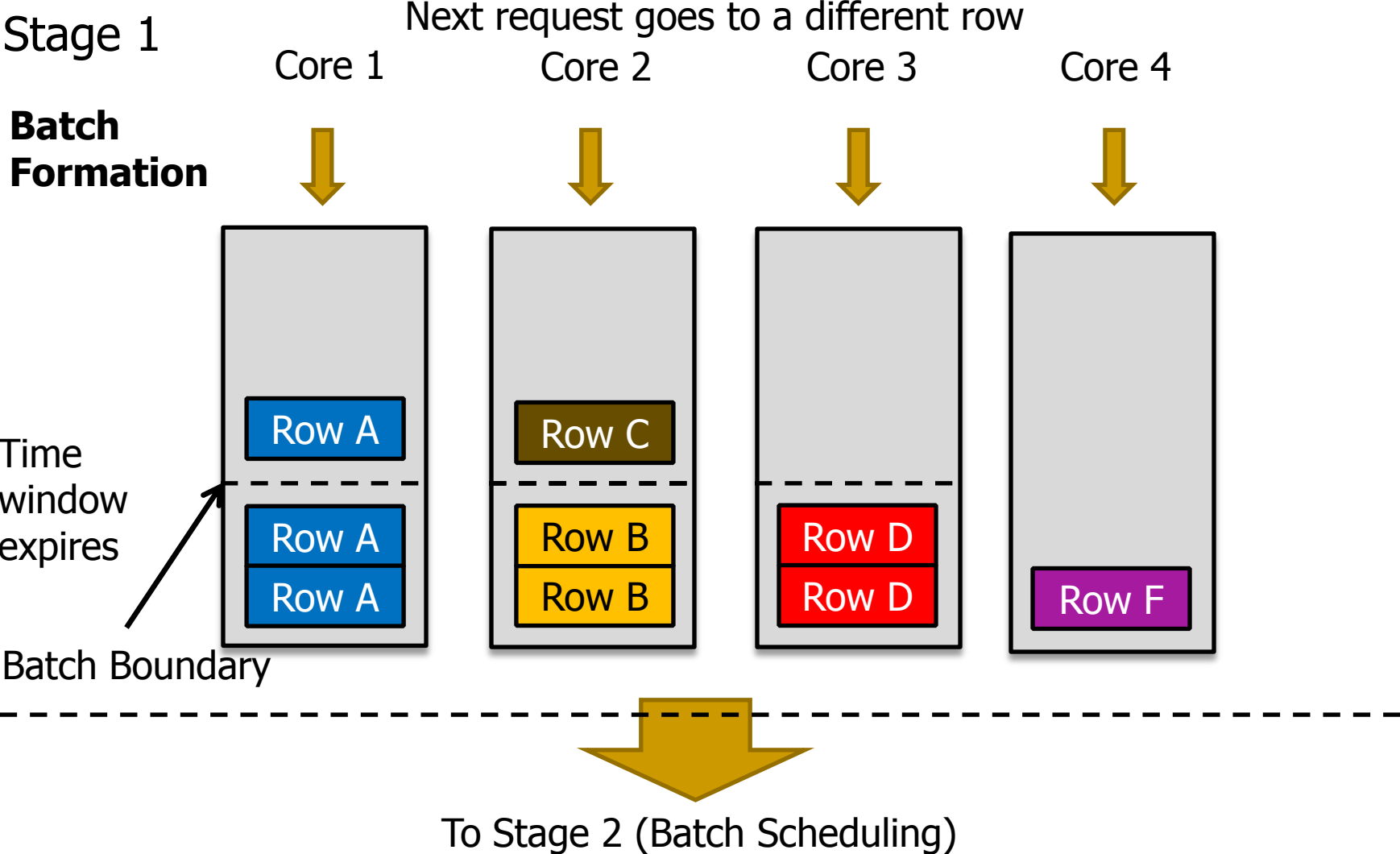
# Stage 1: Batch Formation Example

Stage 1

**Batch Formation**

Time window expires

Batch Boundary



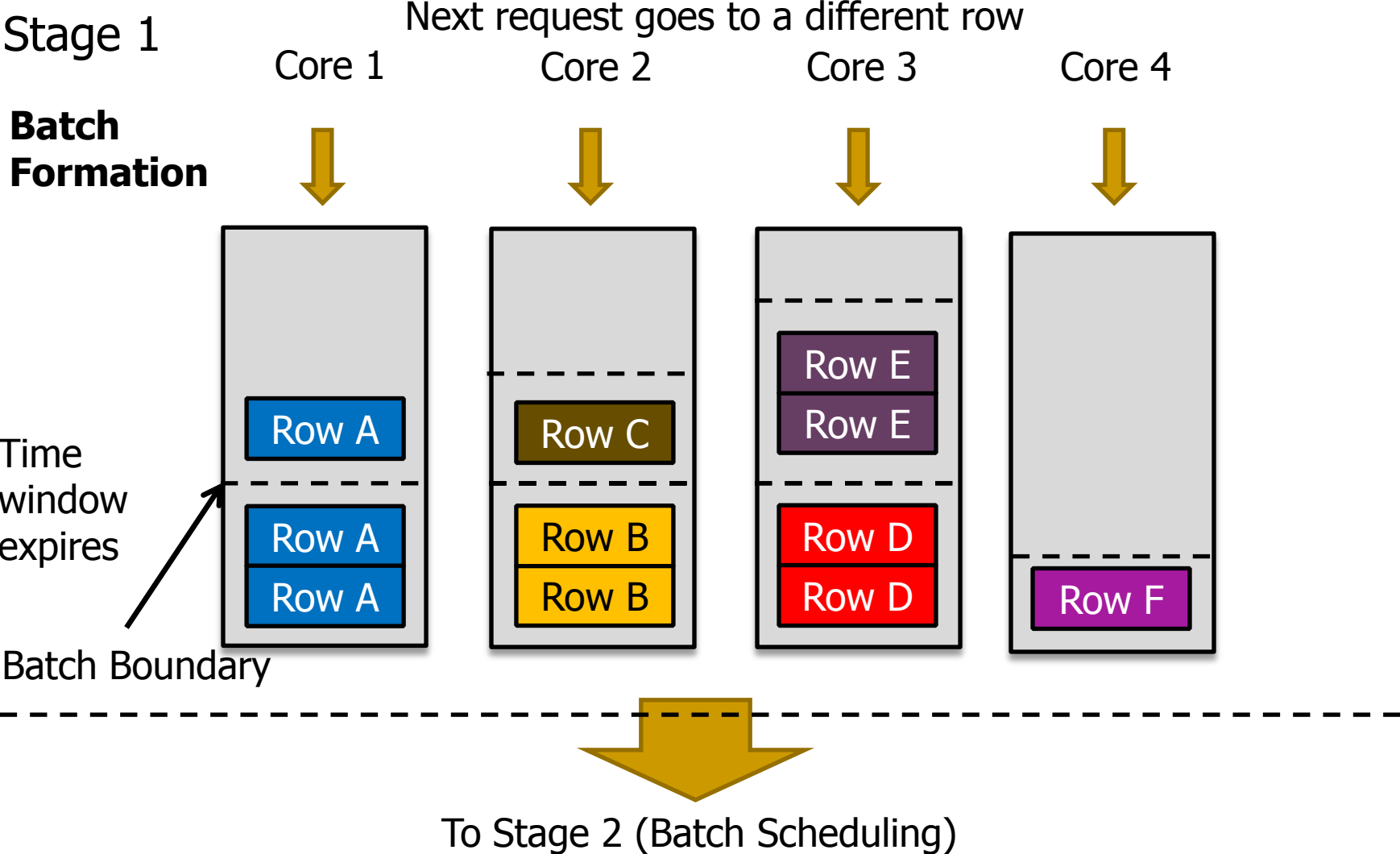
# Stage 1: Batch Formation Example

Stage 1

**Batch Formation**

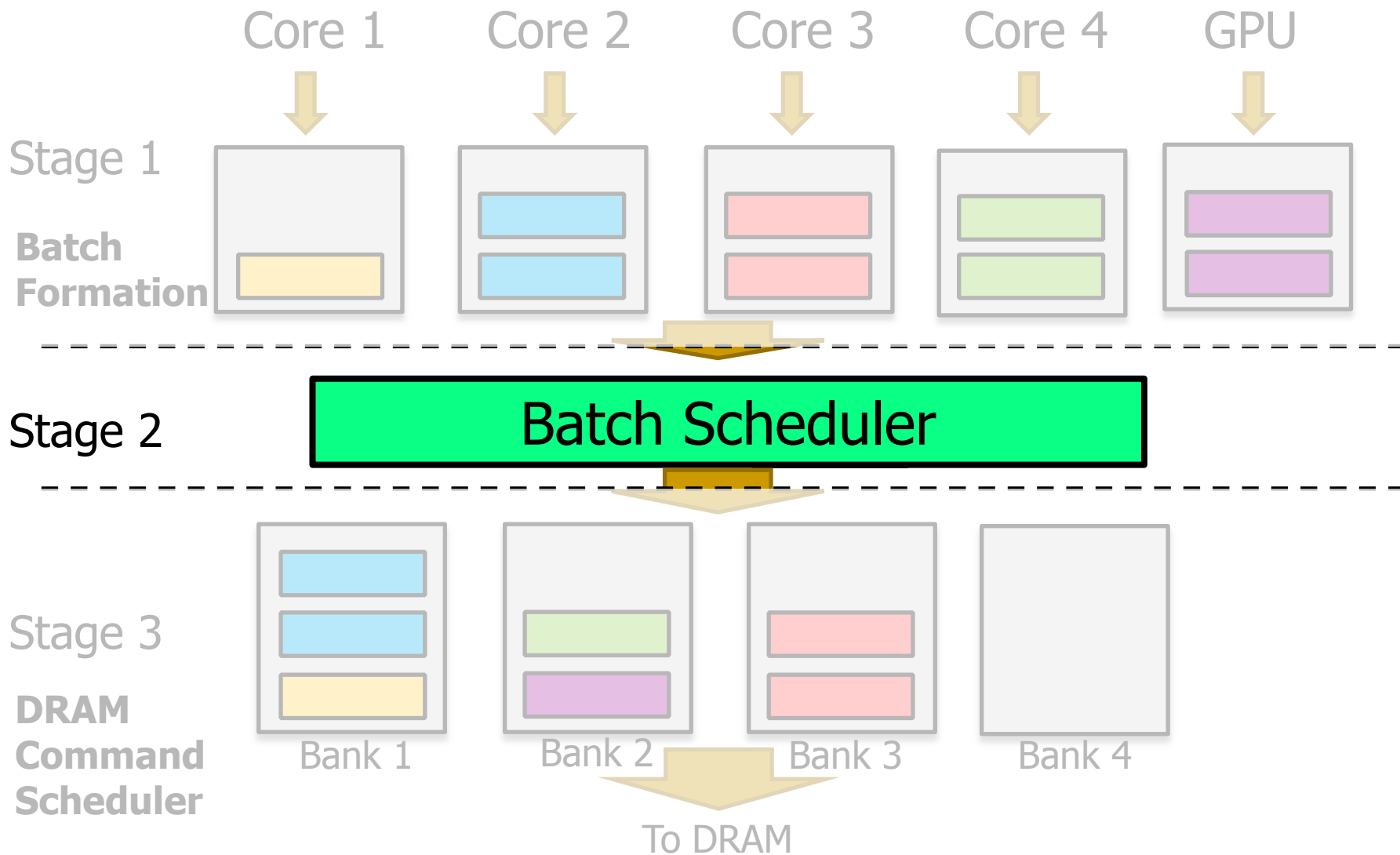
Time window expires

Batch Boundary





# SMS: Staged Memory Scheduling



# Stage 2: Batch Scheduler

---

- Goal: **Minimize interference between applications**
- Stage 1 forms batches **within each application**
- Stage 2 schedules batches **from different applications**
  - Schedules the oldest batch from each application
- Question: Which application's batch should be scheduled next?
- Goal: Maximize system performance and fairness
  - To achieve this goal, the batch scheduler chooses between two different policies

# Stage 2: Two Batch Scheduling Algorithms

---

## ■ **Shortest Job First (SJF)**

- ❑ Prioritize the applications with the fewest outstanding memory requests because **they make fast forward progress**
- ❑ **Pro:** Good system performance and fairness
- ❑ **Con:** GPU and memory-intensive applications get deprioritized

## ■ **Round-Robin (RR)**

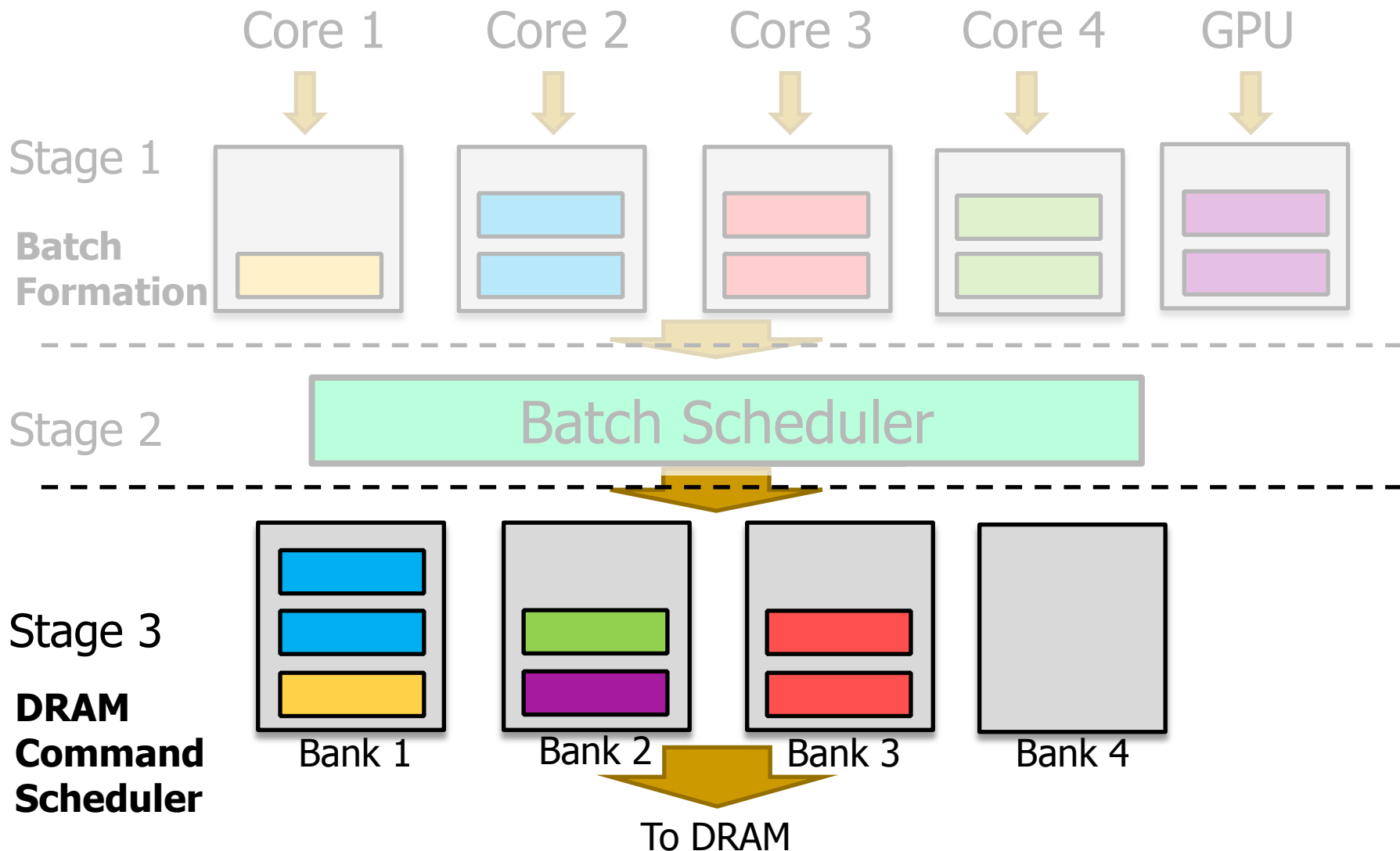
- ❑ Prioritize the applications in a round-robin manner to ensure that **memory-intensive applications can make progress**
- ❑ **Pro:** GPU and memory-intensive applications are treated fairly
- ❑ **Con:** GPU and memory-intensive applications significantly slow down others

# Stage 2: Batch Scheduling Policy

---

- The importance of the GPU varies between systems and over time → Scheduling policy needs to adapt to this
- **Solution:** Hybrid Policy
- At every cycle:
  - With probability  $p$  : Shortest Job First → Benefits the CPU
  - With probability  $1-p$  : Round-Robin → Benefits the GPU
- System software can configure  $p$  based on the importance/weight of the GPU
  - Higher GPU importance → Lower  $p$  value

# SMS: Staged Memory Scheduling



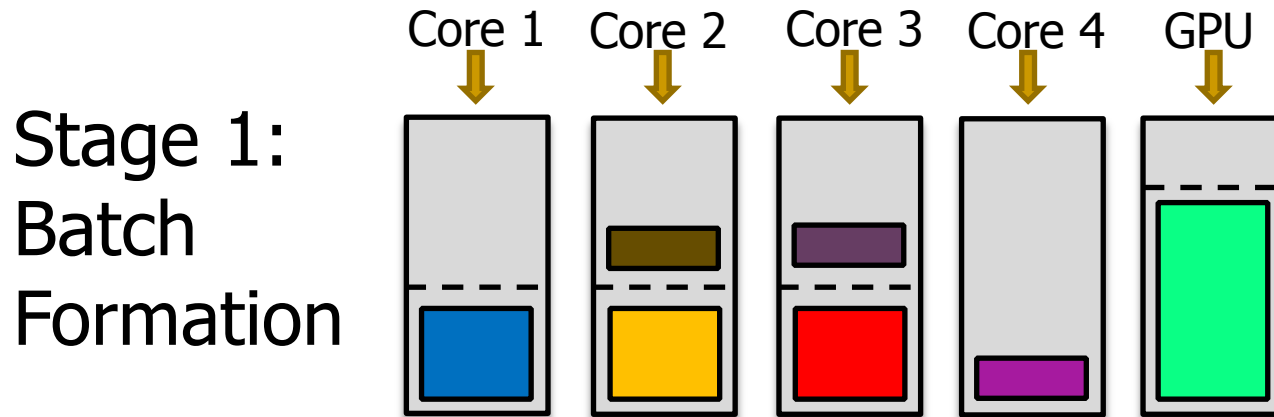
# Stage 3: DRAM Command Scheduler

---

- High level policy decisions have already been made by:
  - Stage 1: Maintains row buffer locality
  - Stage 2: Minimizes inter-application interference
- Stage 3: No need for further scheduling
- Only goal: **service requests while satisfying DRAM timing constraints**
- Implemented as **simple per-bank FIFO queues**

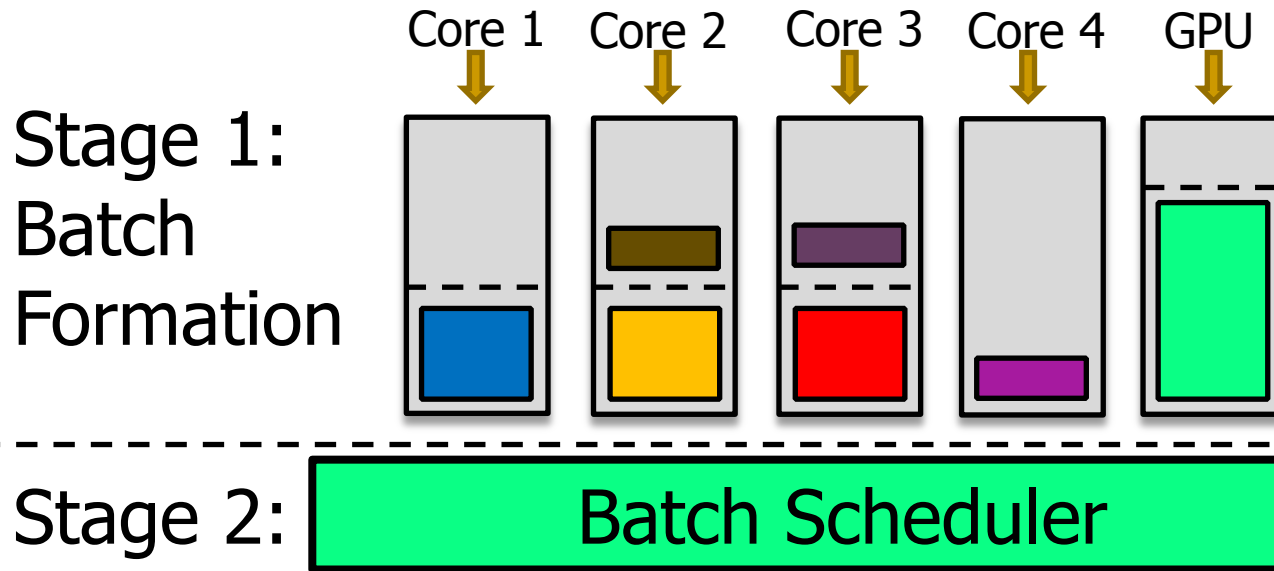
# Putting Everything Together

---



# Putting Everything Together

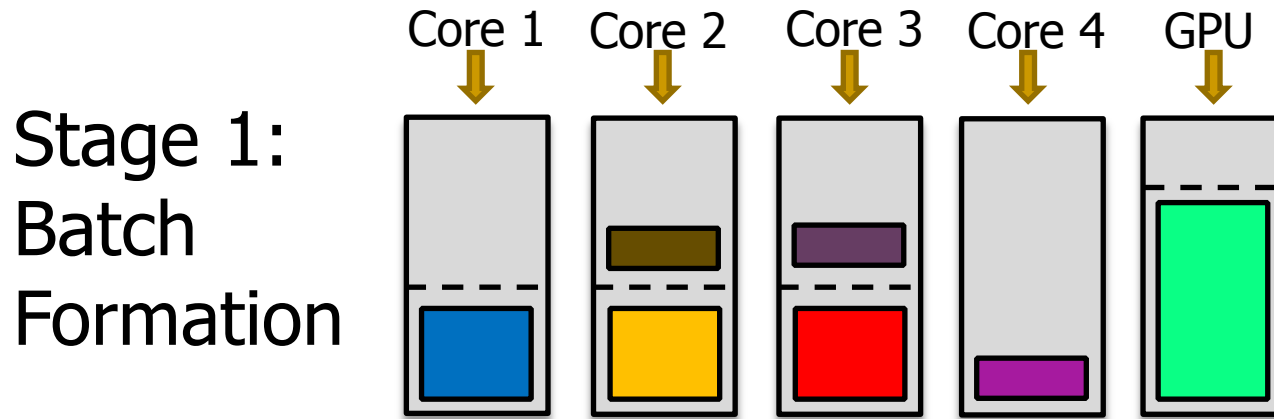
---





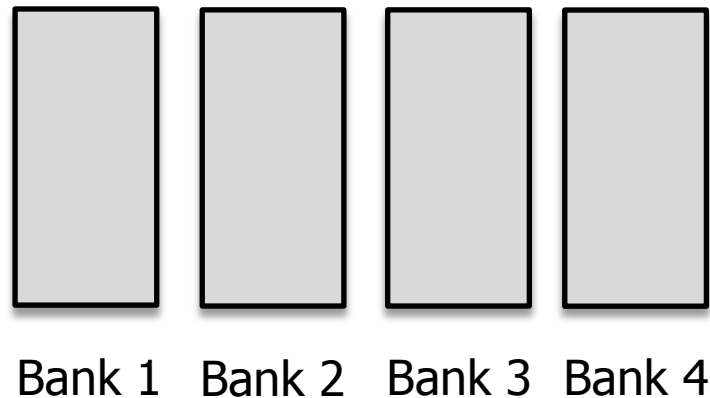
# Putting Everything Together

---

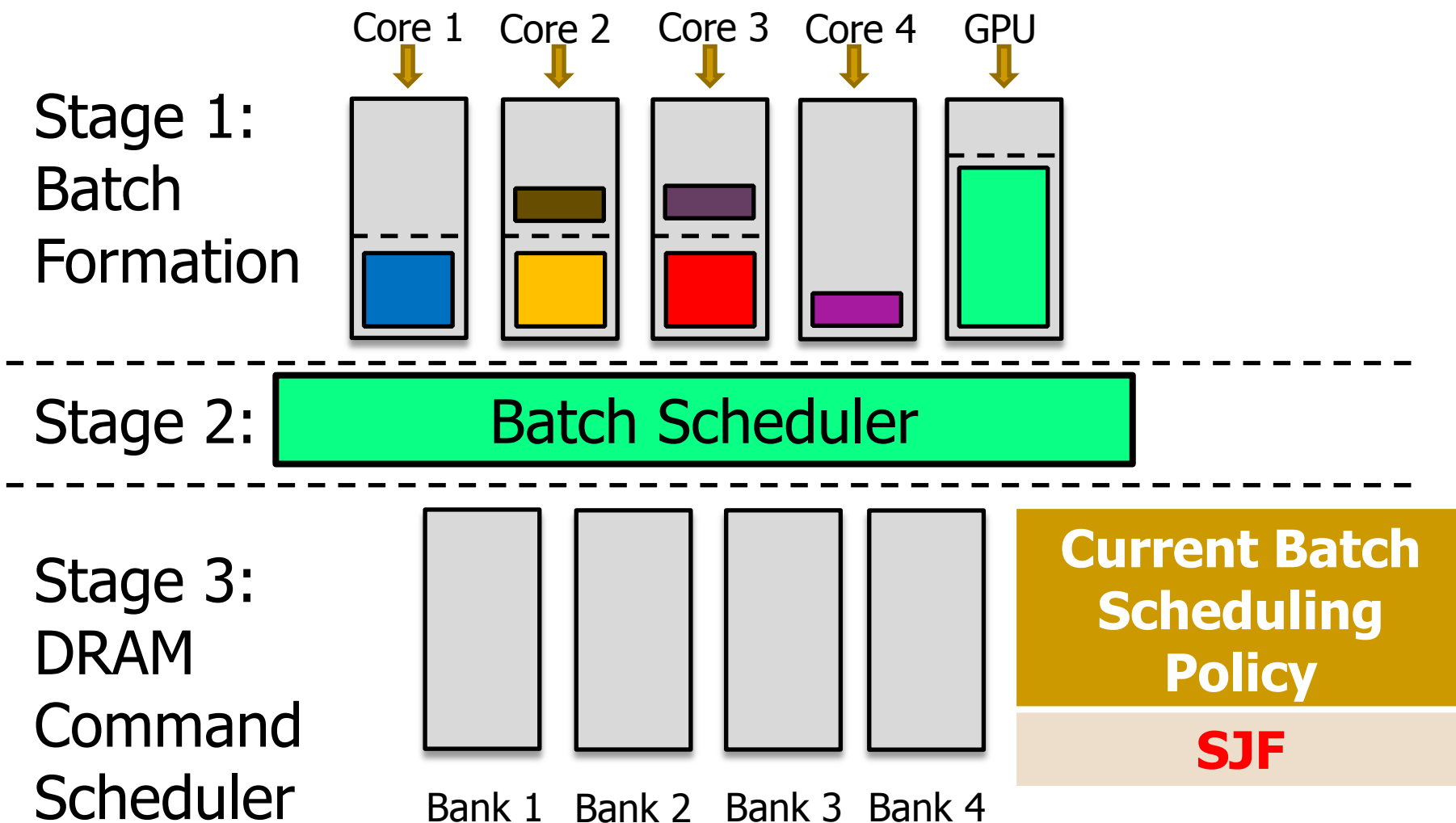


Stage 2: **Batch Scheduler**

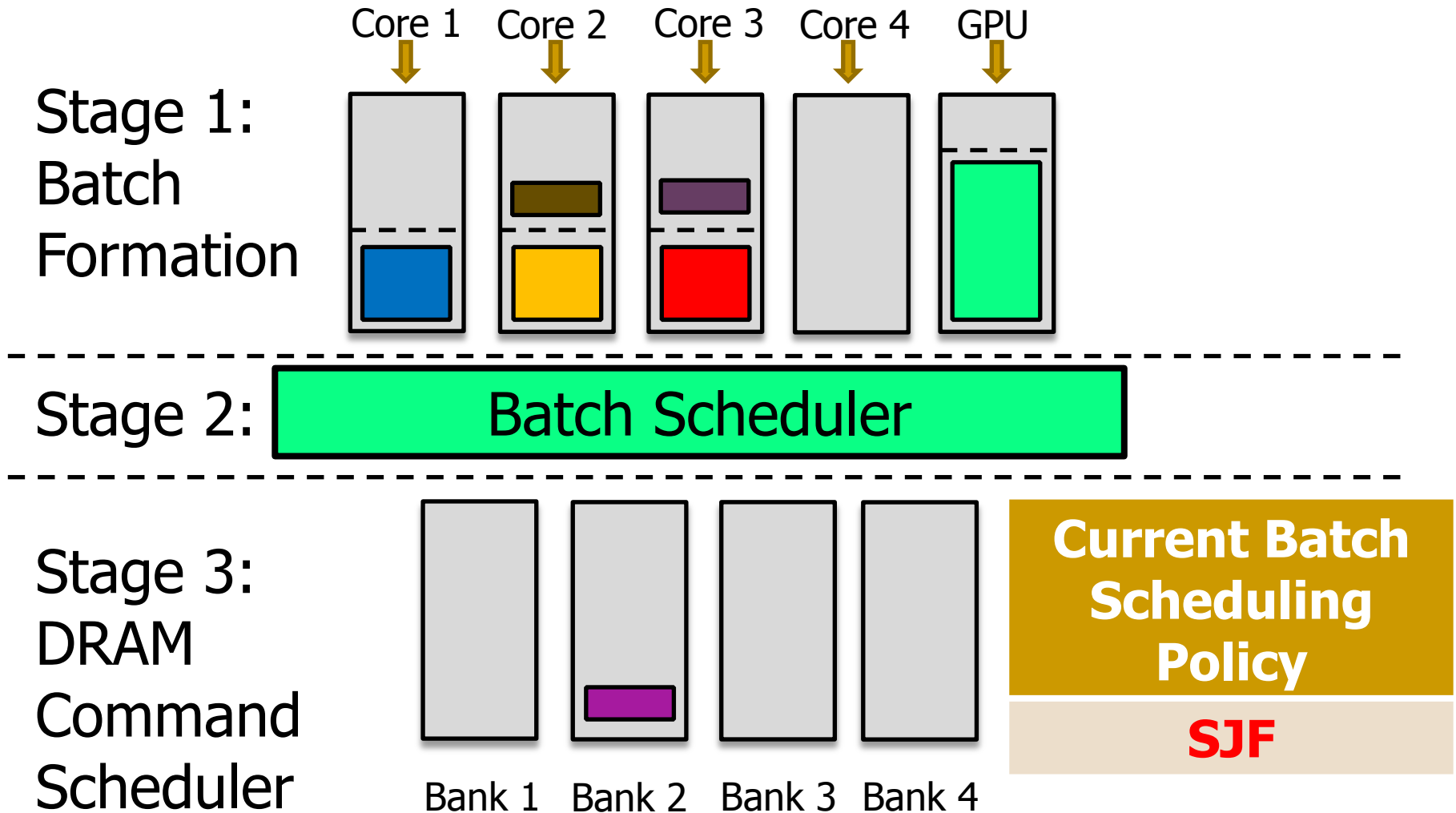
Stage 3:  
DRAM  
Command  
Scheduler



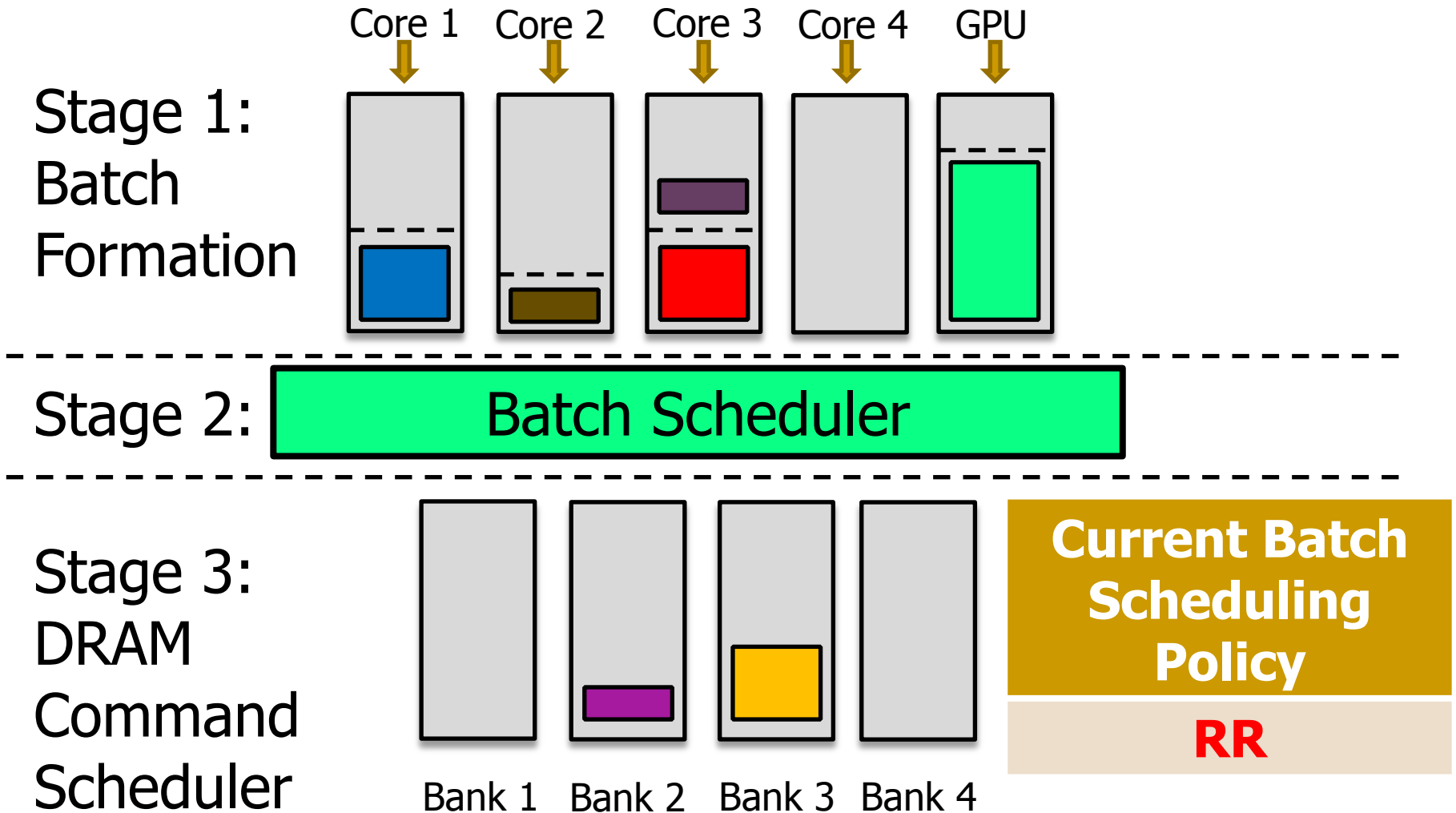
# Putting Everything Together



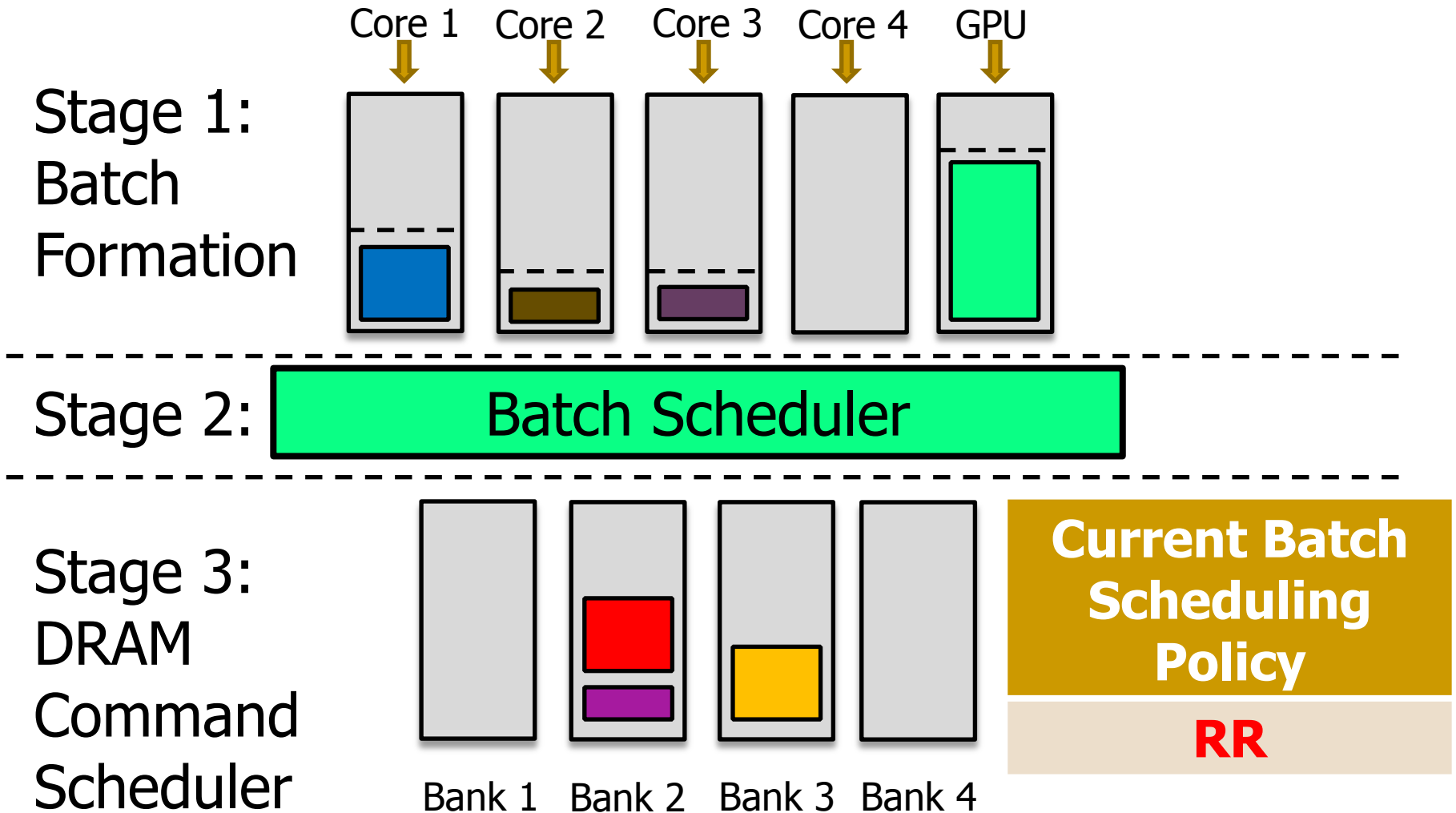
# Putting Everything Together



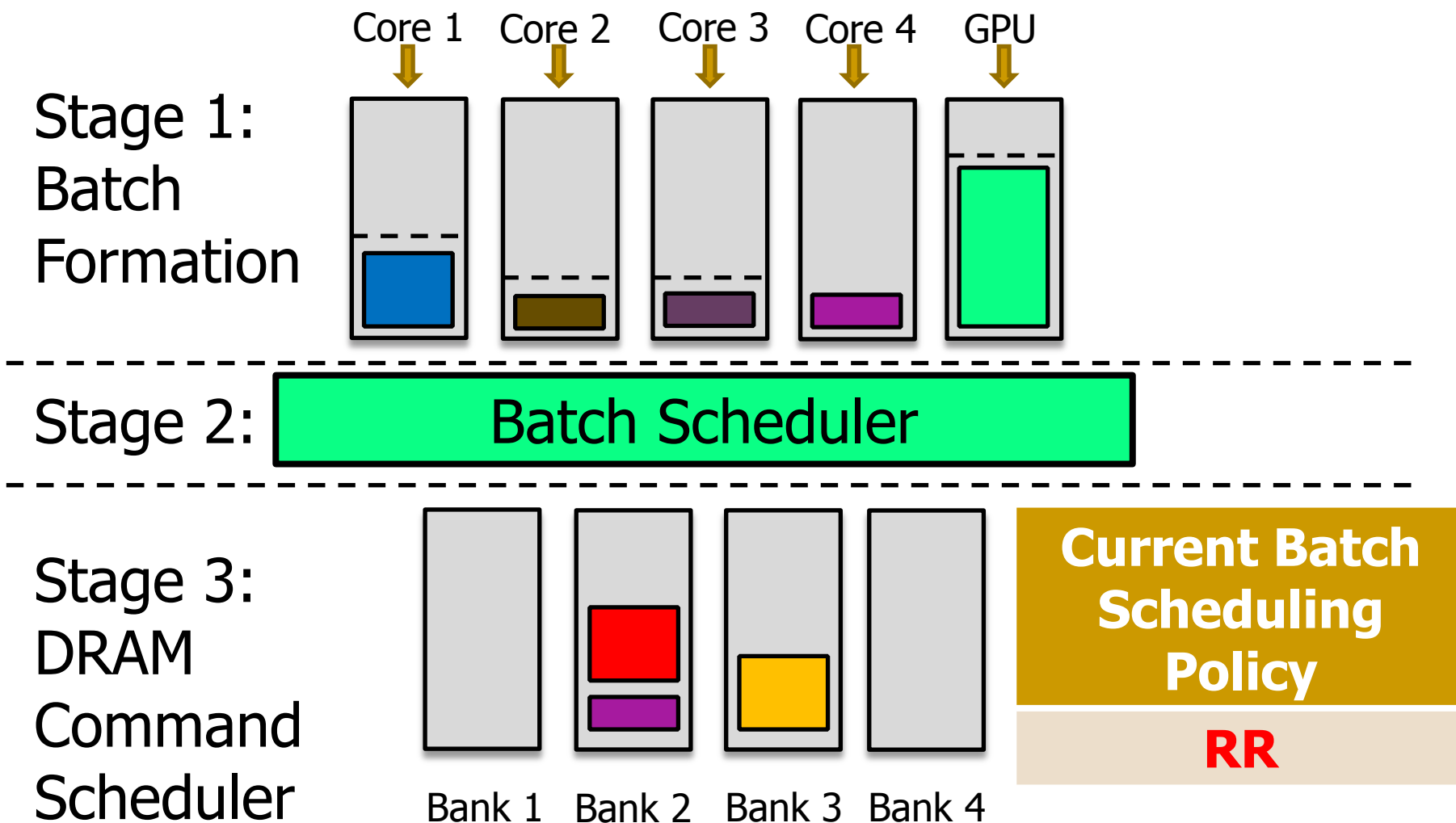
# Putting Everything Together



# Putting Everything Together



# Putting Everything Together



# Complexity

---

- Compared to a row hit first scheduler, SMS consumes\*
  - 66% less area
  - 46% less static power
  
- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer size (buffering is distributed across stages)

# Outline

---

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  - 1) Batch Formation
  - 2) Batch Scheduler
  - 3) DRAM Command Scheduler
- **Results**
- Conclusion



# Methodology

---

- Simulation parameters
  - 16 OoO CPU cores, 1 GPU modeling AMD Radeon™ 5870
  - DDR3-1600 DRAM 4 channels, 1 rank/channel, 8 banks/channel
  
- Workloads
  - CPU: SPEC CPU 2006
  - GPU: Recent games and GPU benchmarks
  - 7 workload categories based on the memory-intensity of CPU applications
    - Low memory-intensity (L)
    - Medium memory-intensity (M)
    - High memory-intensity (H)

# Comparison to Previous Scheduling Algorithms

---

- **FR-FCFS [Rixner+, ISCA'00]**
  - Prioritizes row buffer hits
  - Maximizes DRAM throughput
  - **Low multi-core performance** ← Application unaware
- **ATLAS [Kim+, HPCA'10]**
  - Prioritizes latency-sensitive applications
  - Good multi-core performance
  - **Low fairness** ← Deprioritizes memory-intensive applications
- **TCM [Kim+, MICRO'10]**
  - Clusters low and high-intensity applications and treats each separately
  - Good multi-core performance and fairness
  - **Not robust** ← Misclassifies latency-sensitive applications

# Evaluation Metrics

---

- CPU performance metric: Weighted speedup
- GPU performance metric: Frame rate speedup
- CPU-GPU system performance: CPU-GPU weighted speedup

# Evaluation Metrics

---

- CPU performance metric: Weighted speedup

$$CPU_{WS} = \sum \frac{IPC_{Shared}}{IPC_{Alone}}$$

- GPU performance metric: Frame rate speedup

$$GPU_{Speedup} = \frac{FrameRate_{Shared}}{FrameRate_{Alone}}$$

- CPU-GPU system performance: CPU-GPU weighted speedup

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

# Evaluated System Scenarios

---

- CPU-focused system
- GPU-focused system

# Evaluated System Scenario: CPU Focused

---

- GPU has **low** weight (weight = 1)
  
  
  
  
  
  
  
  
  
  
- Configure SMS such that  $\rho$ , SJF probability, is set to 0.9
  - **Mostly uses SJF** batch scheduling → prioritizes latency-sensitive applications (mainly CPU)

# Evaluated System Scenario: CPU Focused

---

- GPU has **low** weight (weight = 1)

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

- Configure SMS such that  $\rho$ , SJF probability, is set to 0.9
  - **Mostly uses SJF** batch scheduling → prioritizes latency-sensitive applications (mainly CPU)

# Evaluated System Scenario: CPU Focused

---

- GPU has **low** weight (weight = 1)

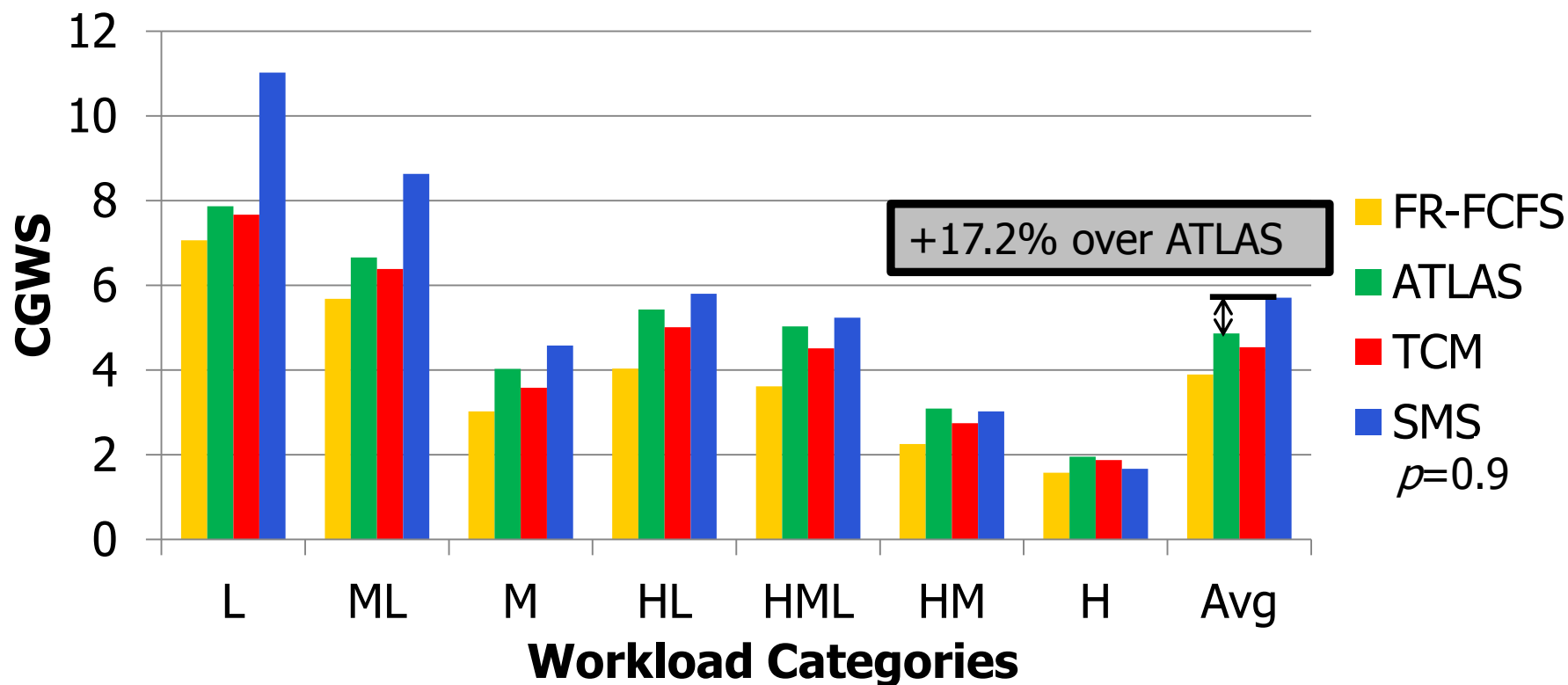
$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

**1**

- Configure SMS such that  $\rho$ , SJF probability, is set to 0.9
  - **Mostly uses SJF** batch scheduling → prioritizes latency-sensitive applications (mainly CPU)

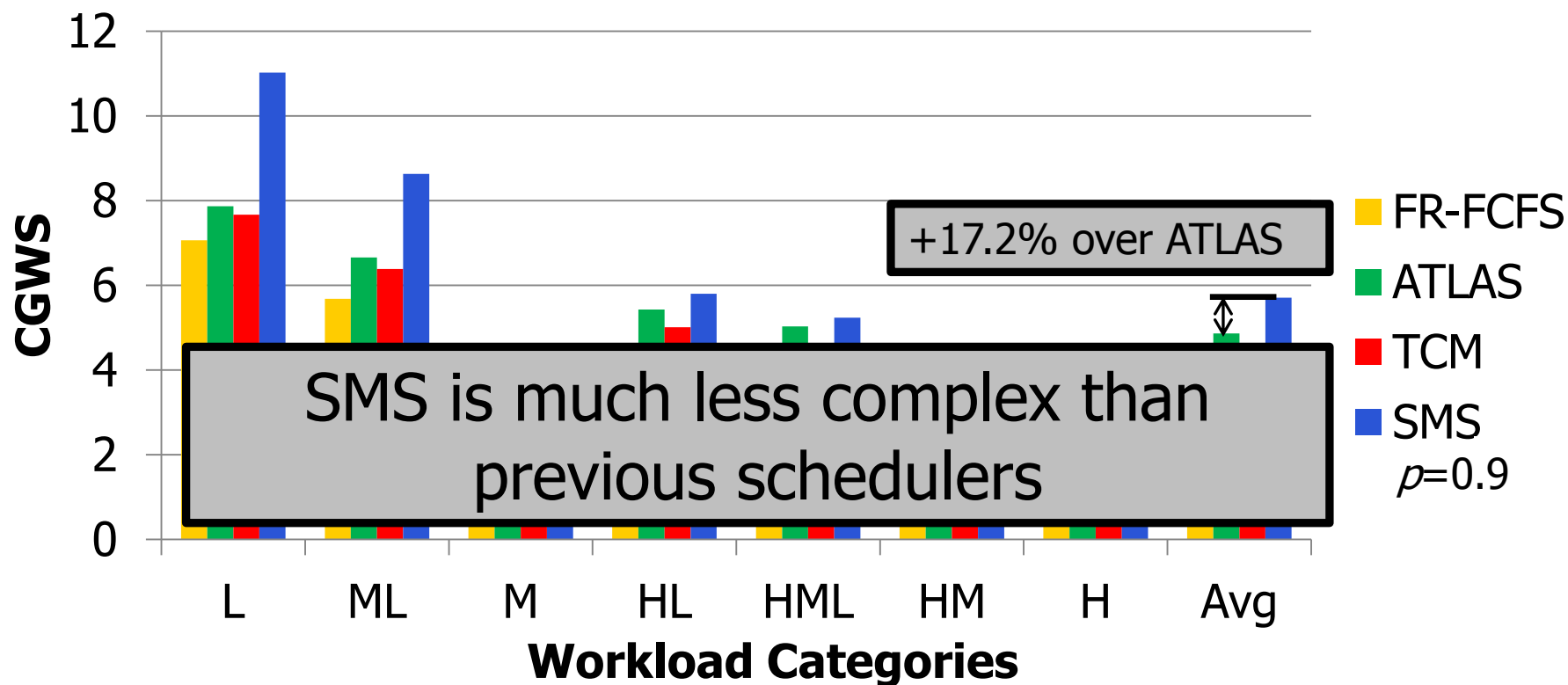


# Performance: CPU-Focused System



- SJF batch scheduling policy allows latency-sensitive applications to get serviced as fast as possible

# Performance: CPU-Focused System



- SJF batch scheduling policy allows latency-sensitive applications to get serviced as fast as possible

# Evaluated System Scenario: GPU Focused

---

- GPU has **high** weight (weight = 1000)
  
- Configure SMS such that  $\rho$ , SJF probability, is set to 0
  - **Always uses round-robin** batch scheduling → prioritizes memory-intensive applications (GPU)

# Evaluated System Scenario: GPU Focused

---

- GPU has **high** weight (weight = 1000)

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

- Configure SMS such that  $\rho$ , SJF probability, is set to 0
  - **Always uses round-robin** batch scheduling → prioritizes memory-intensive applications (GPU)

# Evaluated System Scenario: GPU Focused

---

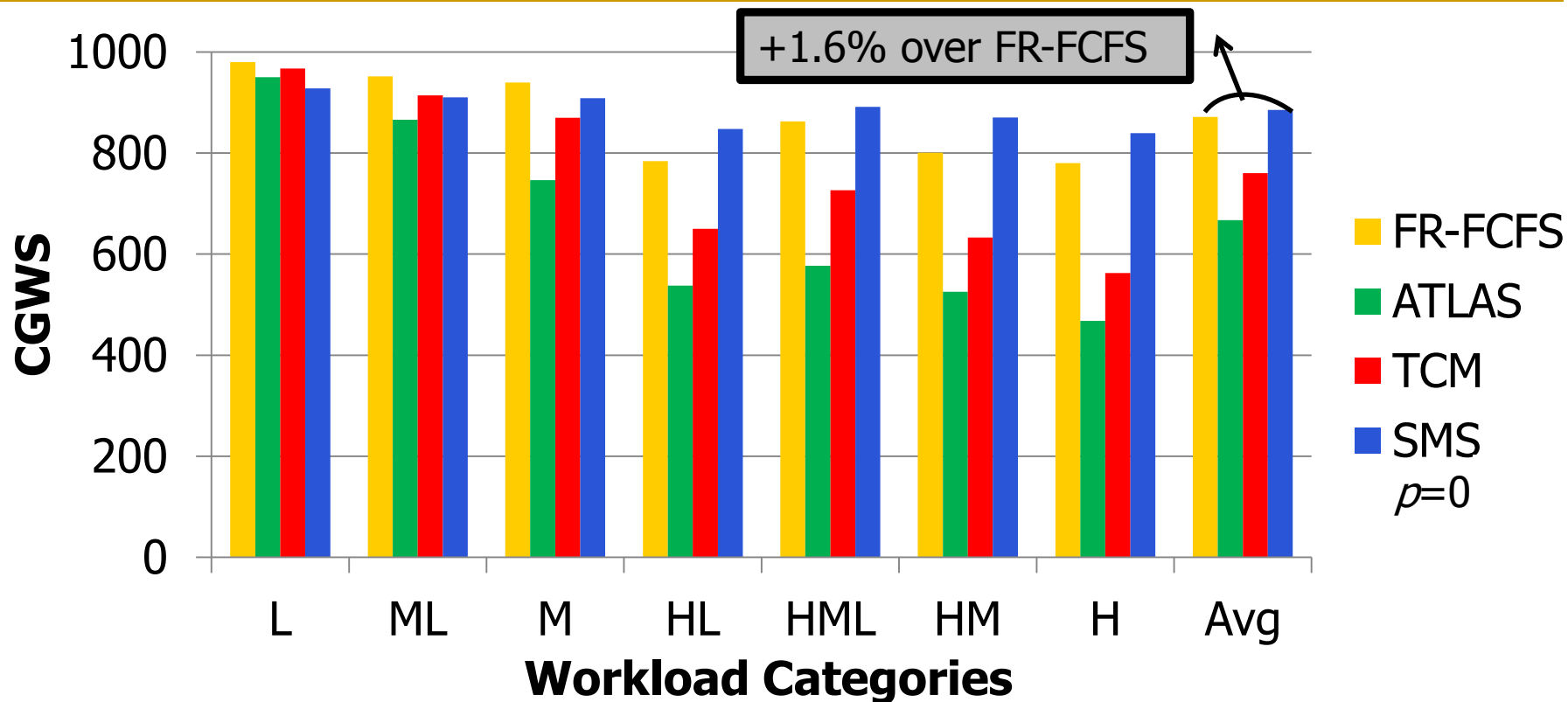
- GPU has **high** weight (weight = 1000)

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

**1000**

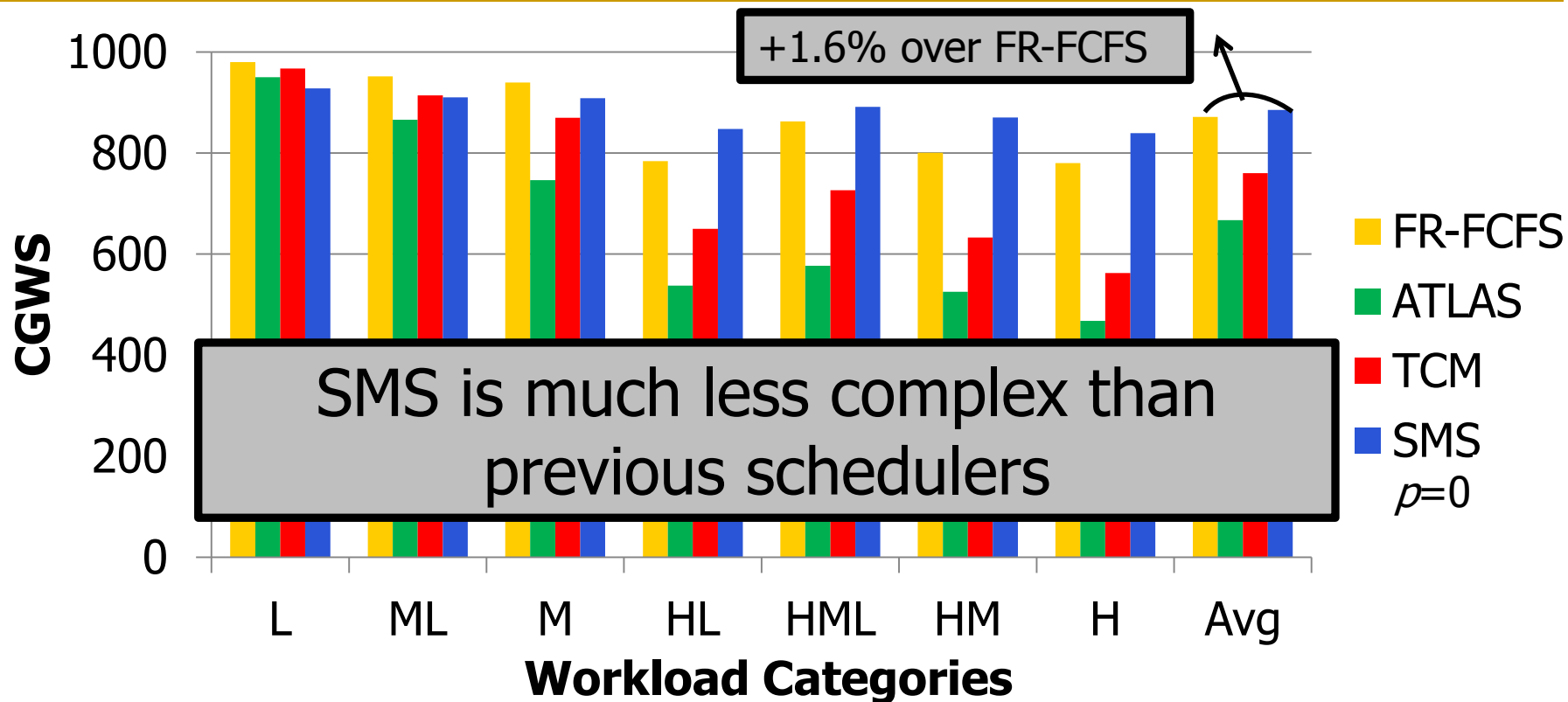
- Configure SMS such that  $\rho$ , SJF probability, is set to 0
  - **Always uses round-robin** batch scheduling → prioritizes memory-intensive applications (GPU)

# Performance: GPU-Focused System



- Round-robin batch scheduling policy schedules GPU requests more frequently

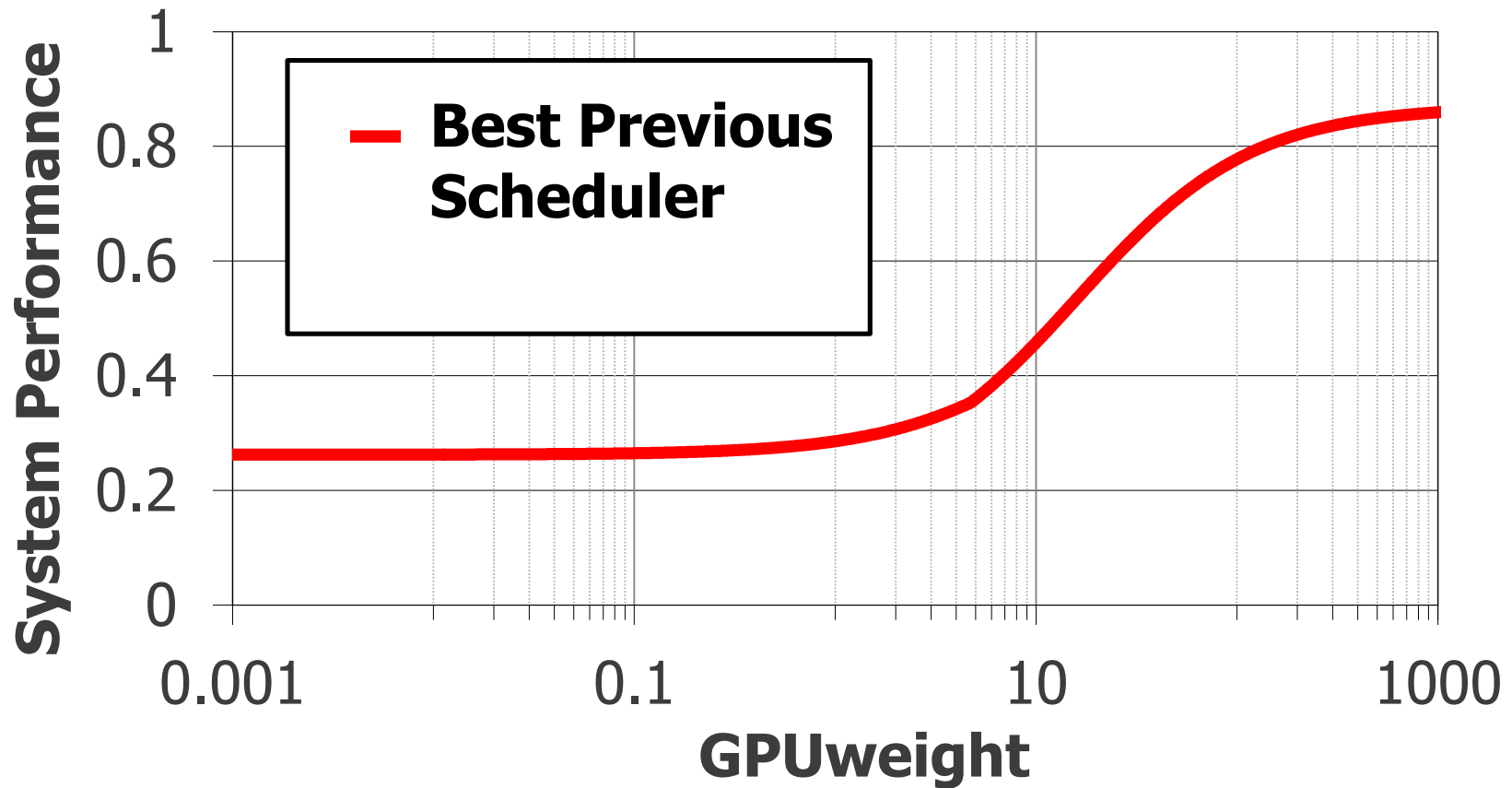
# Performance: GPU-Focused System



- Round-robin batch scheduling policy schedules GPU requests more frequently

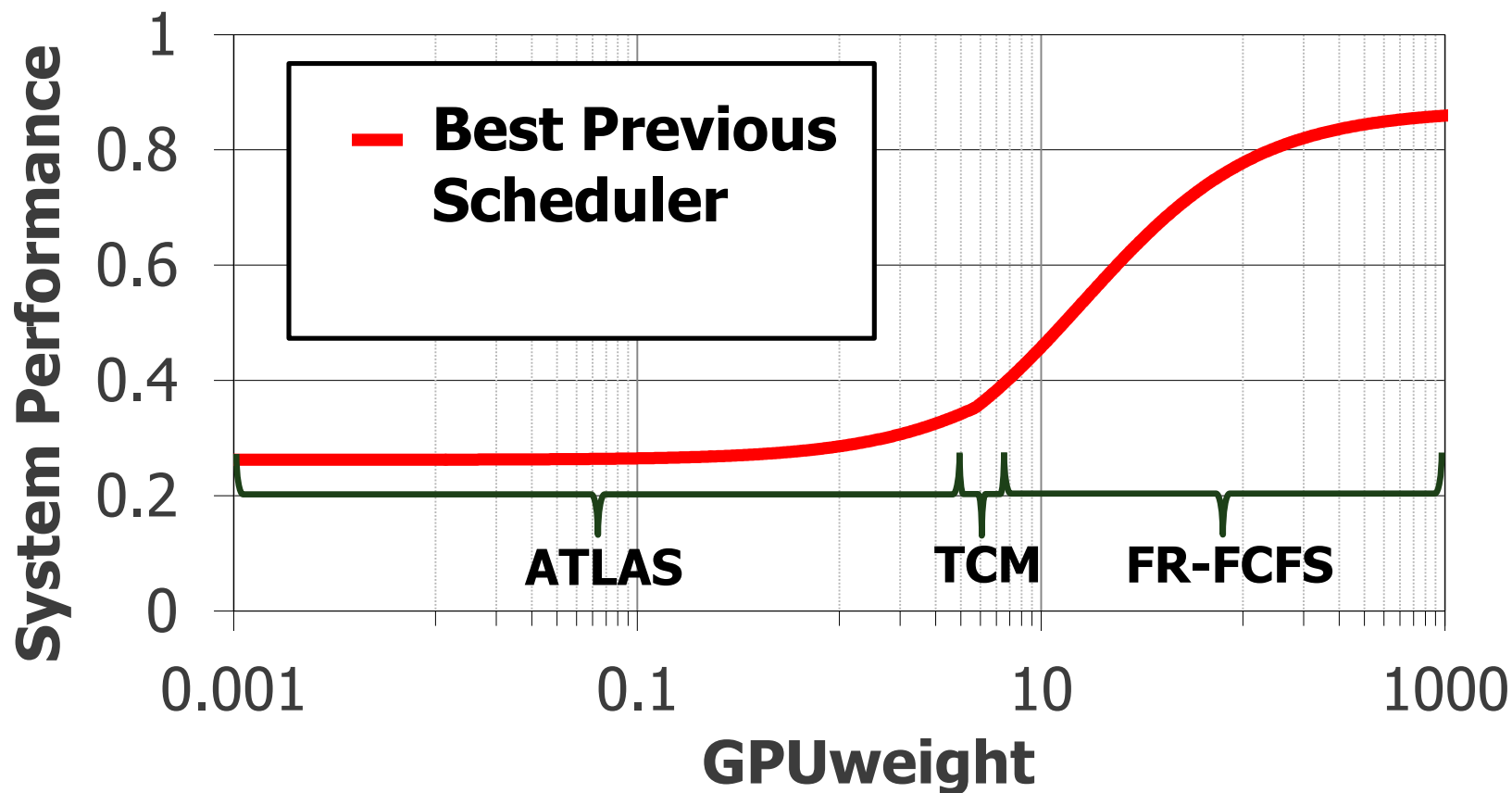
# Performance at Different GPU Weights

---



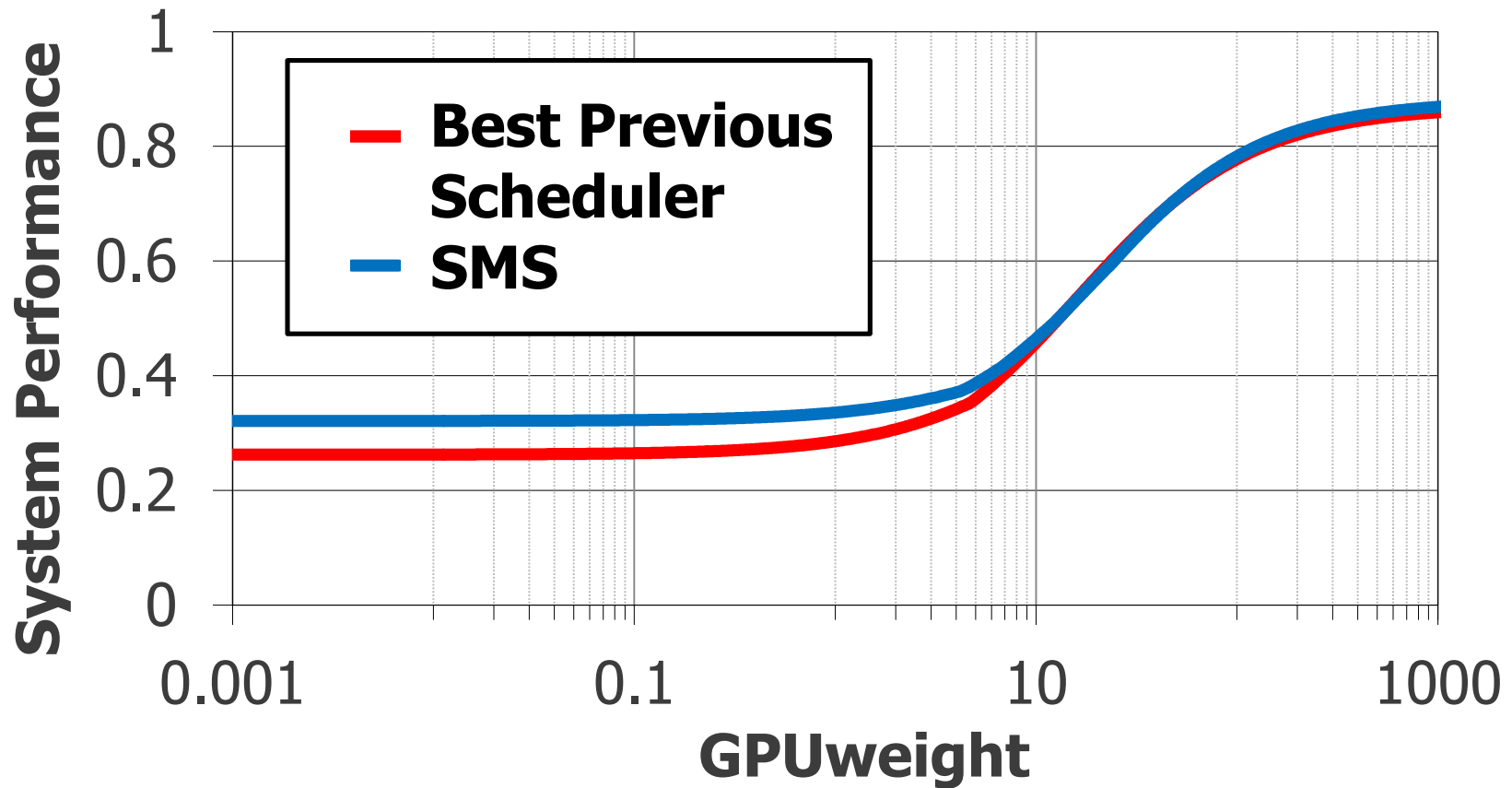


# Performance at Different GPU Weights

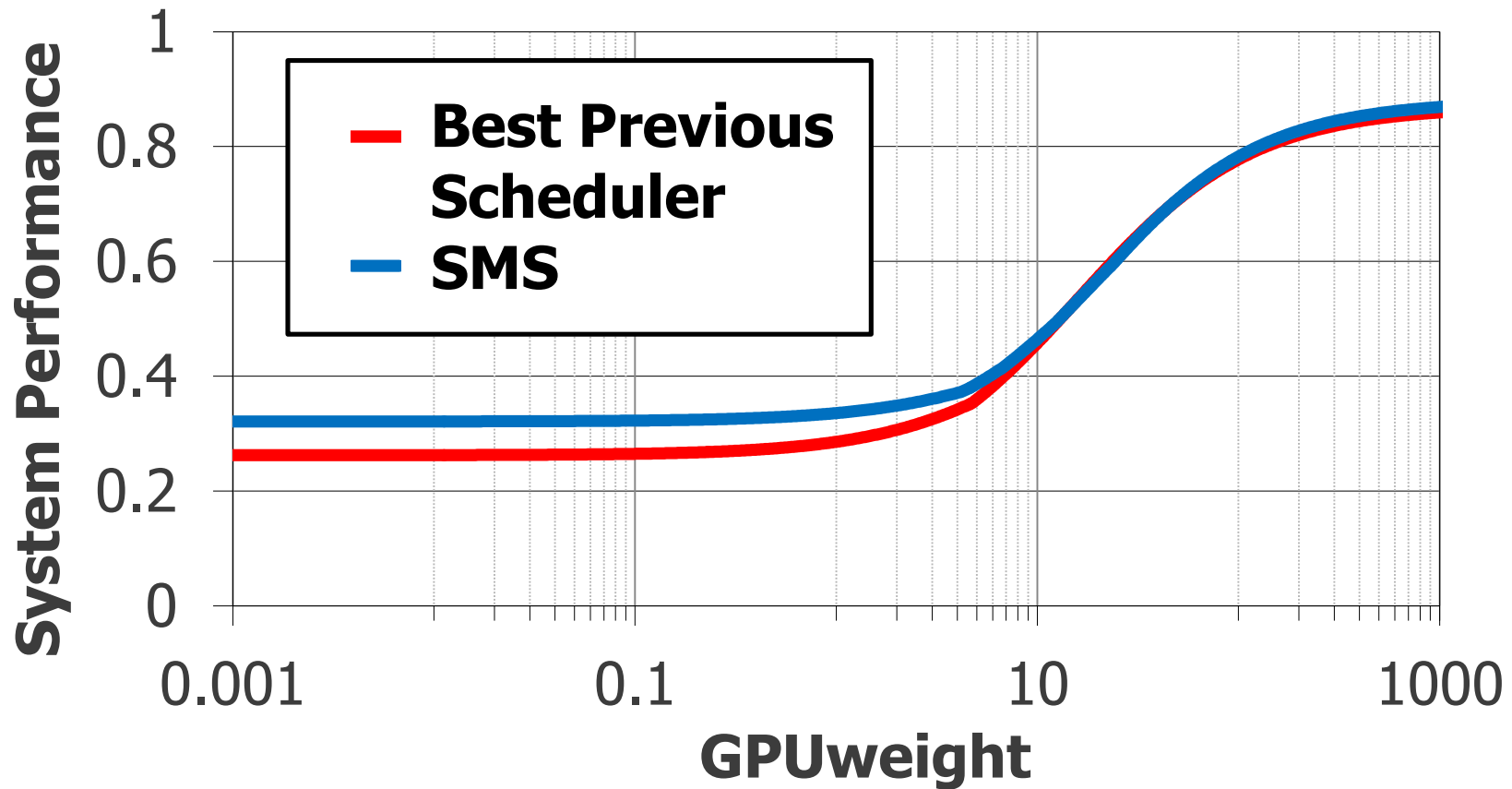


# Performance at Different GPU Weights

---



# Performance at Different GPU Weights



- At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight

# Additional Results in the Paper

---

- Fairness evaluation
  - 47.6% improvement over the best previous algorithms
- Individual CPU and GPU performance breakdowns
- CPU-only scenarios
  - Competitive performance with previous algorithms
- Scalability results
  - SMS' performance and fairness scales better than previous algorithms as the number of cores and memory channels increases
- Analysis of SMS design parameters

# Outline

---

- Background
- Motivation: CPU-GPU Systems
- Our Goal
- Observations
- Staged Memory Scheduling
  - 1) Batch Formation
  - 2) Batch Scheduler
  - 3) DRAM Command Scheduler
- Results
- Conclusion

# Conclusion

---

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with **large request buffers**
  - **Problem:** Existing monolithic application-aware memory scheduler designs are **hard to scale** to large request buffer size
  - **Solution:** Staged Memory Scheduling (SMS)  
**decomposes the memory controller into three simple stages:**
    - 1) Batch formation: maintains row buffer locality
    - 2) Batch scheduler: reduces interference between applications
    - 3) DRAM command scheduler: issues requests to DRAM
  - Compared to state-of-the-art memory schedulers:
    - **SMS is significantly simpler and more scalable**
    - **SMS provides higher performance and fairness**
-

# Staged Memory Scheduling

**Rachata Ausavarungnirun**, Kevin Chang, Lavanya Subramanian,  
Gabriel H. Loh\*, Onur Mutlu

Carnegie Mellon University, \*AMD Research  
June 12<sup>th</sup> 2012

**SAFARI**

**Carnegie Mellon**

**AMD** 

# Backup Slides

---



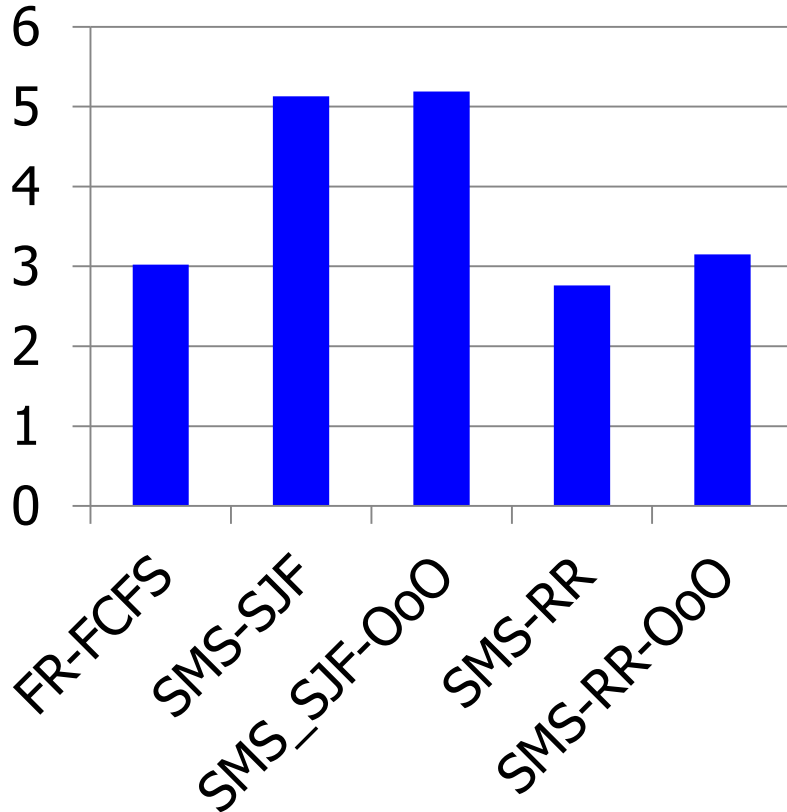
# Row Buffer Locality on Batch Formation

---

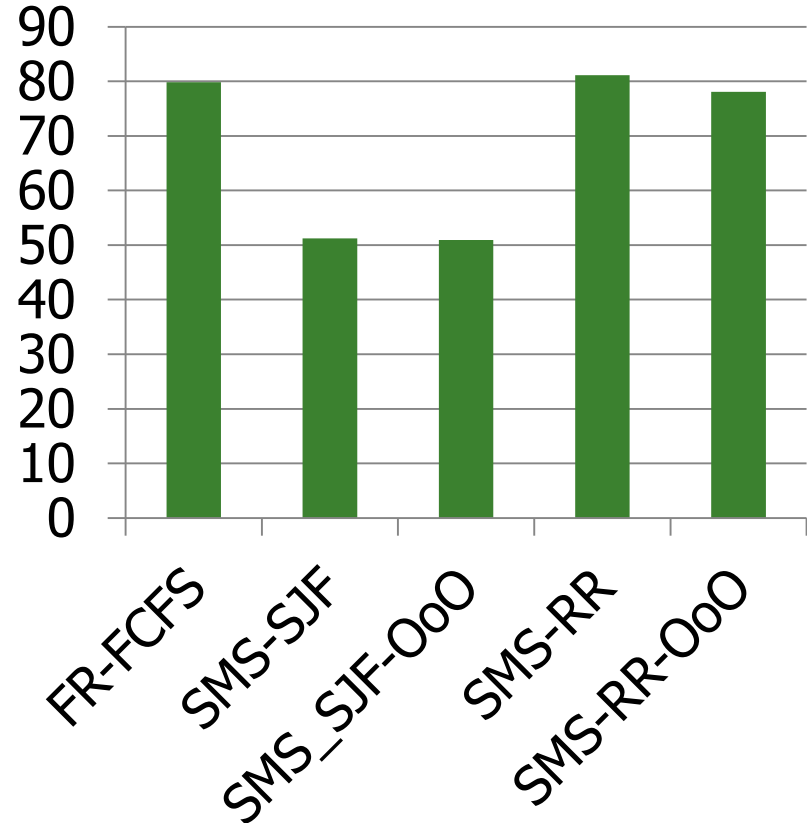
- OoO batch formation improves the performance of the system by:
  - ~3% when the batch scheduler uses SJF policy most of the time
  - ~7% when the batch scheduler uses RR most of the time
- However, OoO batch formation is more complex
  - OoO buffering instead of FIFO queues
  - Need to fine tune the time window of the batch formation based on application characteristics (only 3%-5% performance gain without fine tuning)

# Row Buffer Locality on Batch Formation

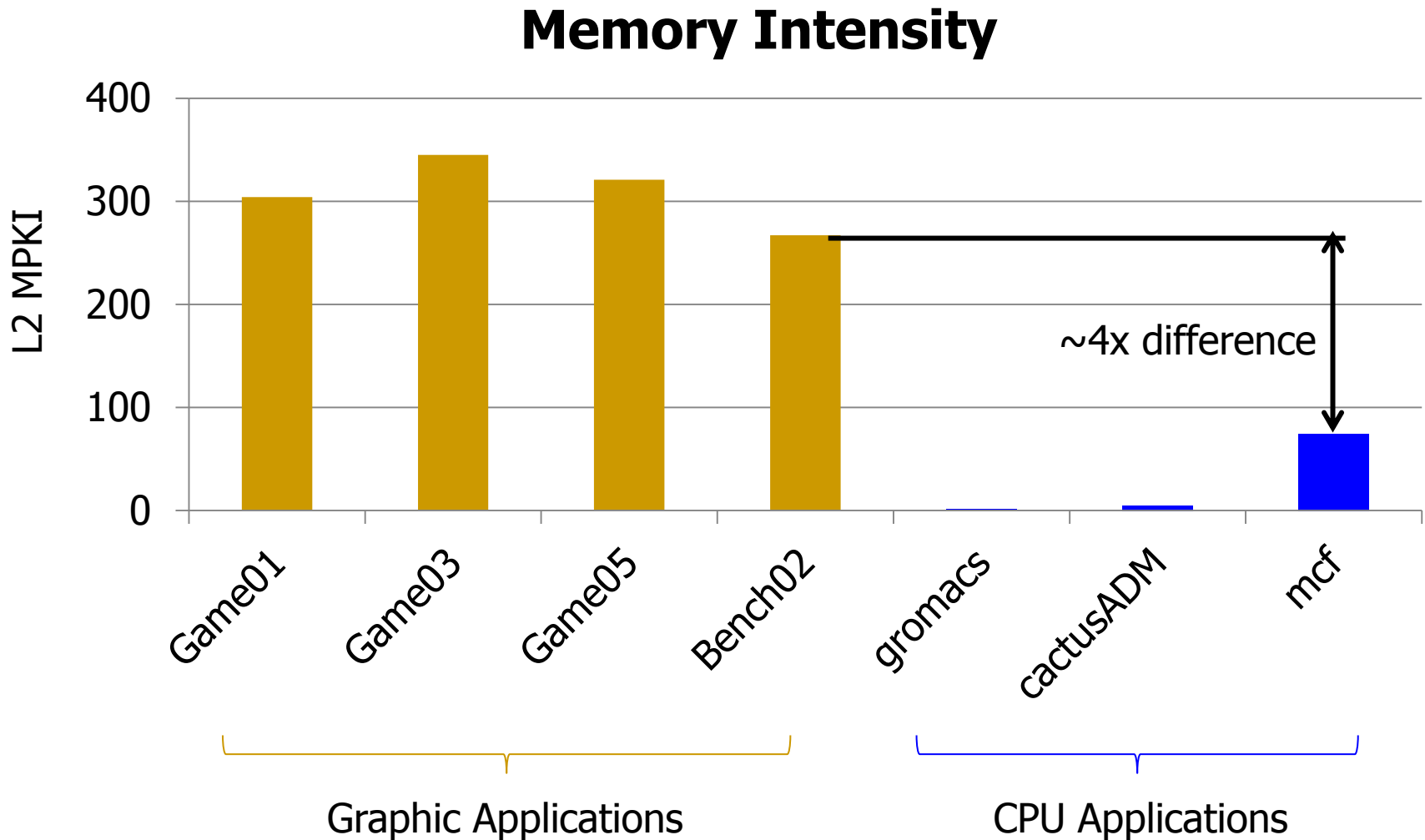
## CPU-WS



## GPU-Frame Rate



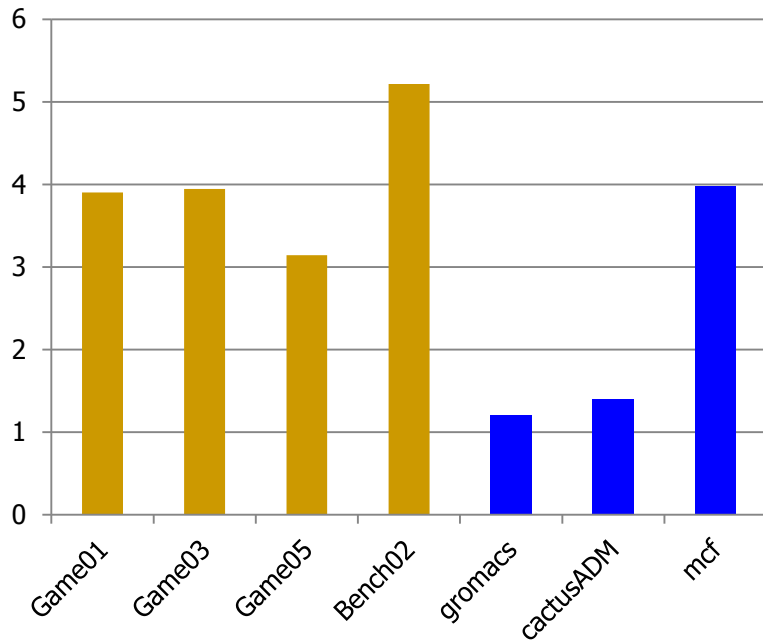
# Key Differences Between CPU and GPU



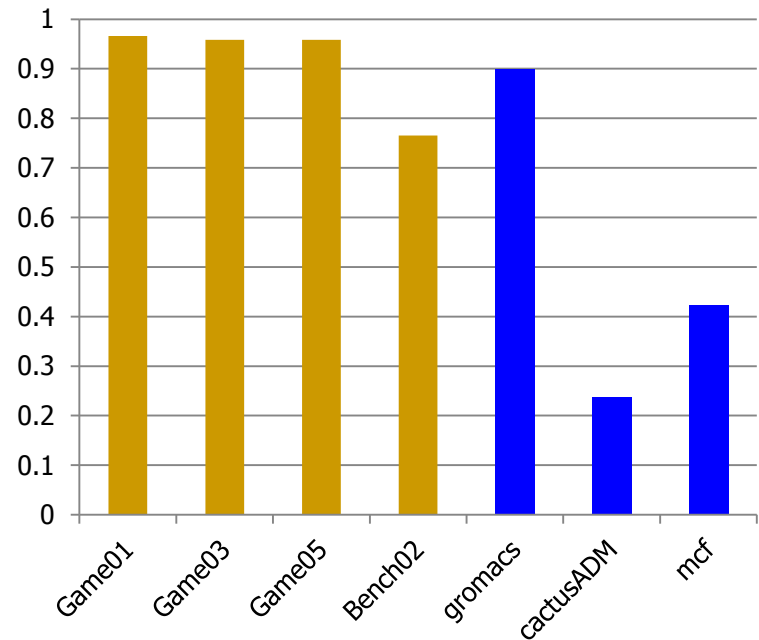
# MLP and RBL

- Key differences between a CPU application and a GPU application

## Memory Level Parallelism



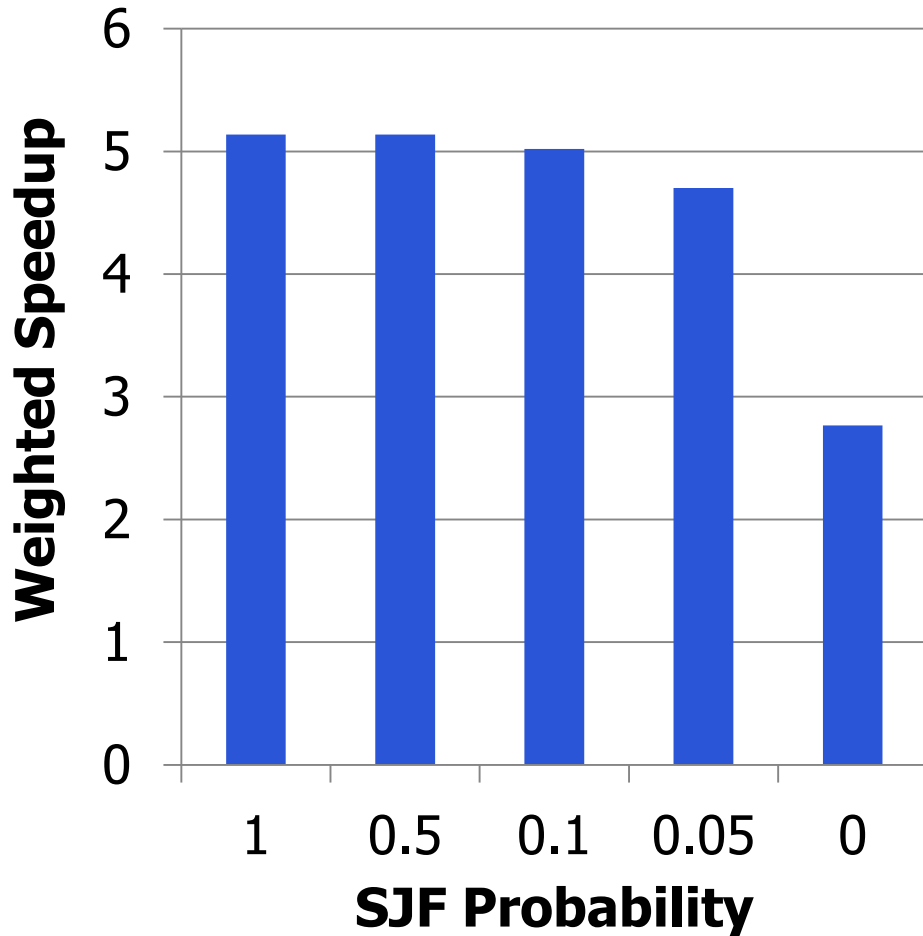
## Row Buffer Locality



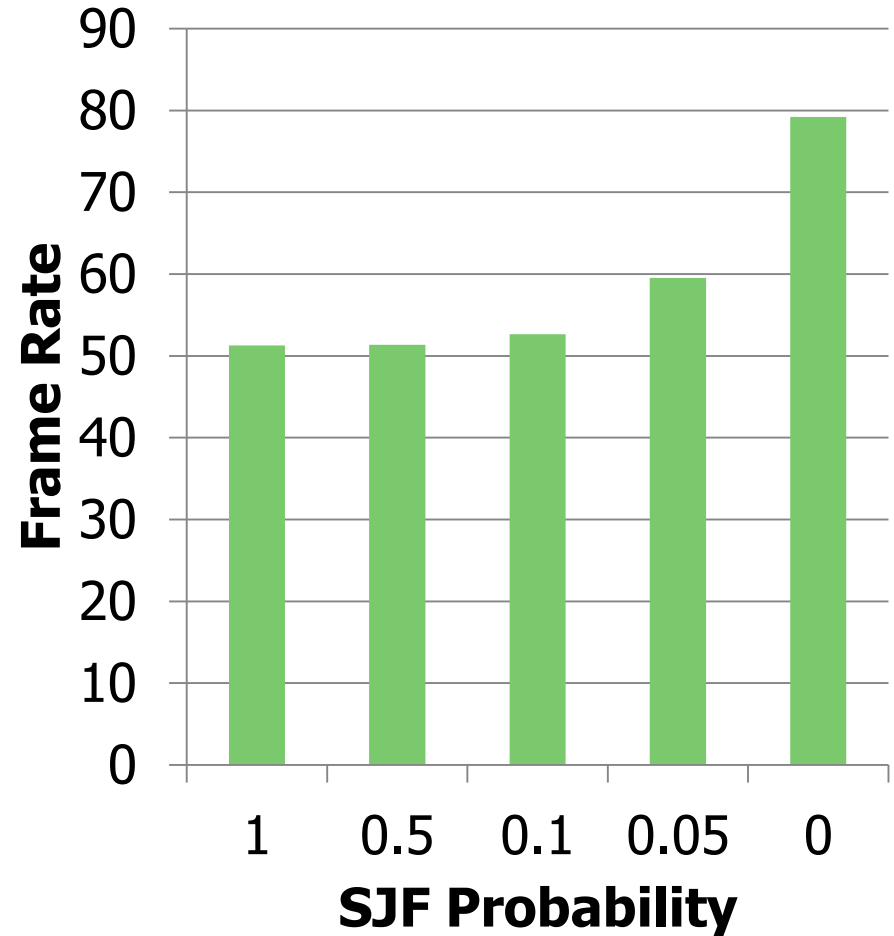
# CPU-GPU Performance Tradeoff

---

## CPU Performance



## GPU Frame Rate



# Dealing with Multi-Threaded Applications

---

- Batch formation: Groups requests from each application in a per-thread FIFO
- Batch scheduler: Detects critical threads and prioritizes them over non-critical threads
  - Previous works have shown how to detect and schedule critical threads
    - 1) Bottleneck Identification and Scheduling in MT applications [Joao+, ASPLOS'12]
    - 2) Parallel Application Memory Scheduling [Ebrahimi, MICRO'11]
- DRAM command scheduler: Stays the same

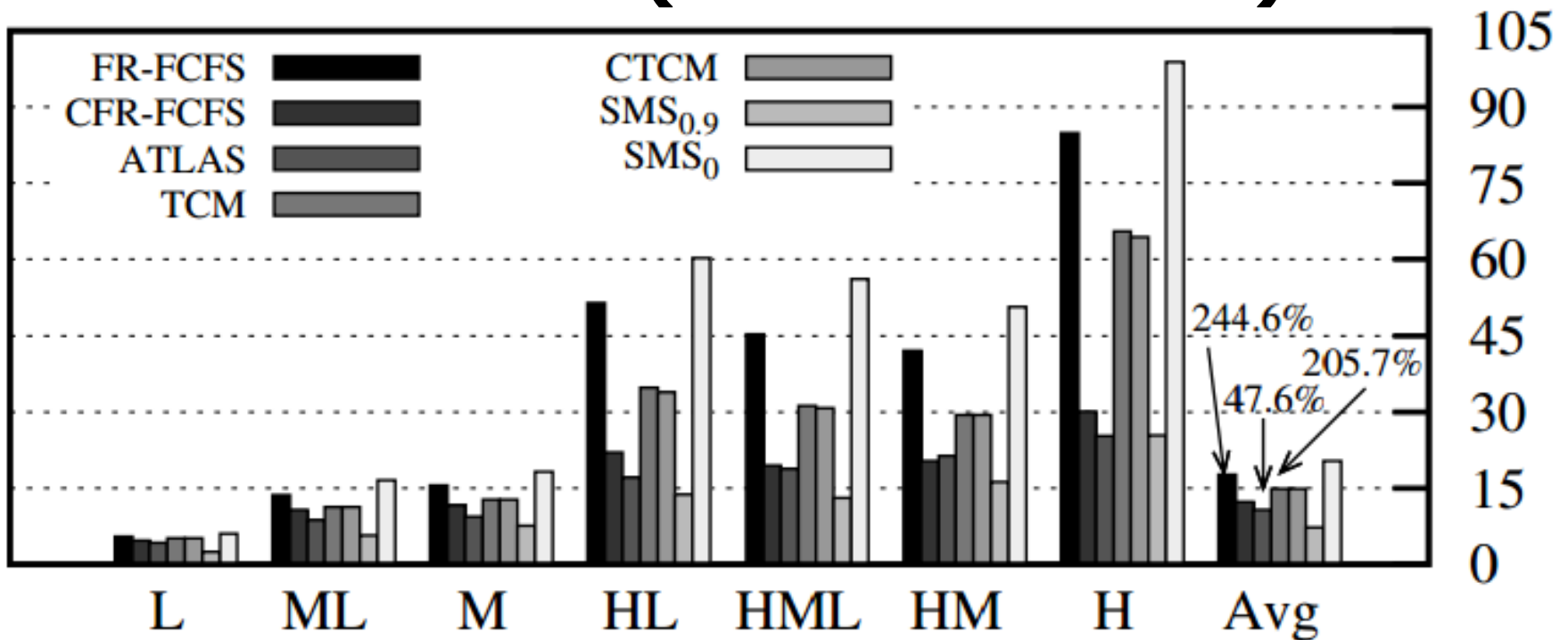
# Dealing with Prefetch Requests

---

- Previous works have proposed several solutions:
  - Prefetch-Aware Shared-Resource Management for Multi-Core Systems [Ebrahimi+, ISCA'11]
  - Coordinated Control of Multiple Prefetchers in Multi-Core Systems [Ebrahimi+, MICRO'09]
  - Prefetch-aware DRAM Controller [Lee+, MICRO'08]
- Handling Prefetch Requests in SMS:
  - SMS can handle prefetch requests before they enter the memory controller (e.g., source throttling based on prefetch accuracy)
  - SMS can handle prefetch requests by prioritizing/deprioritizing prefetch batch at the batch scheduler (based on prefetch accuracy)

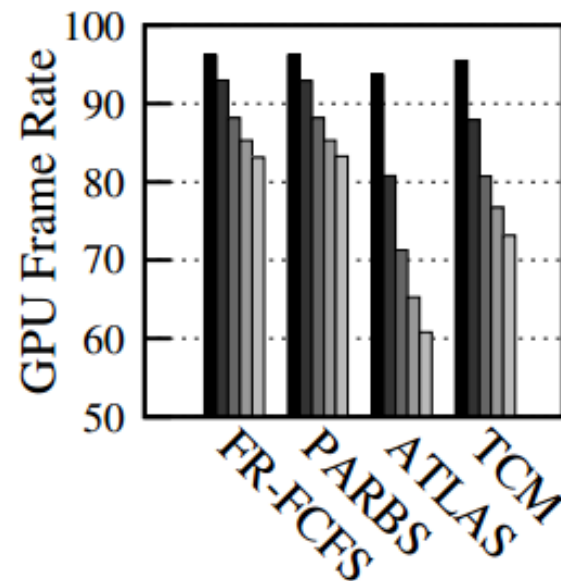
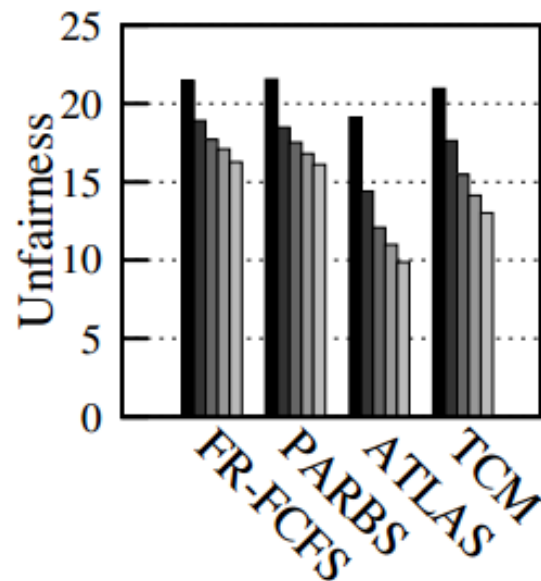
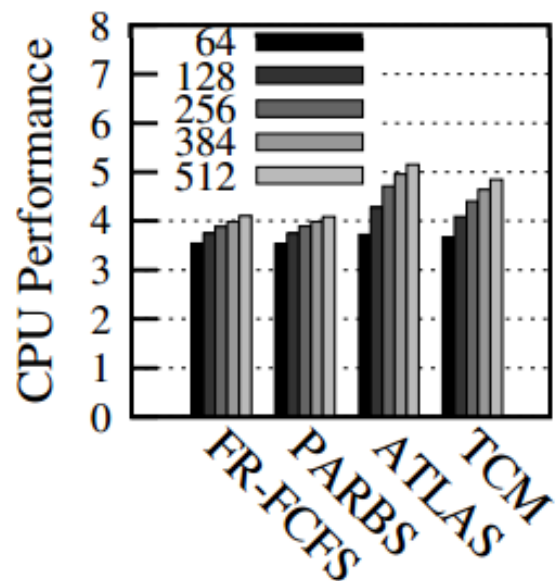
# Fairness Evaluation

## Unfairness (Lower is better)



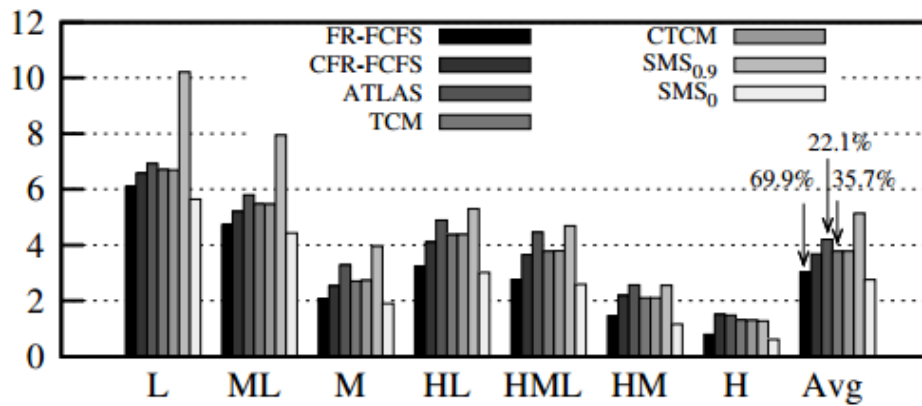


# Performance at Different Buffer Sizes

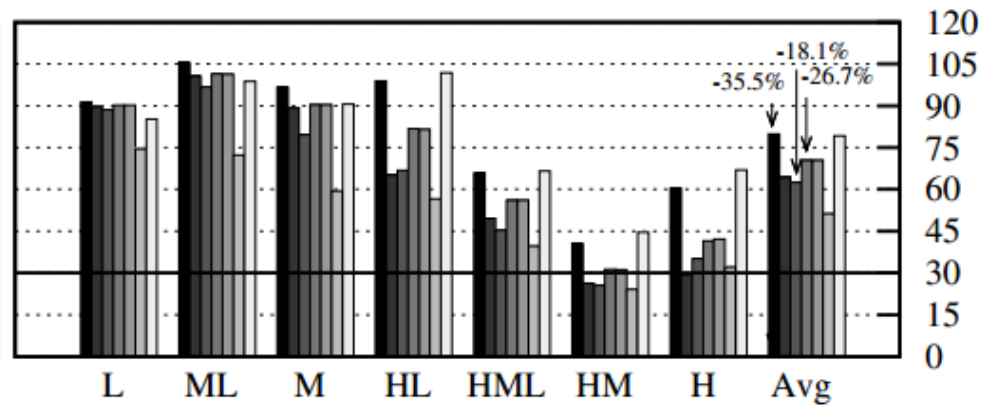


# CPU and GPU Performance Breakdowns

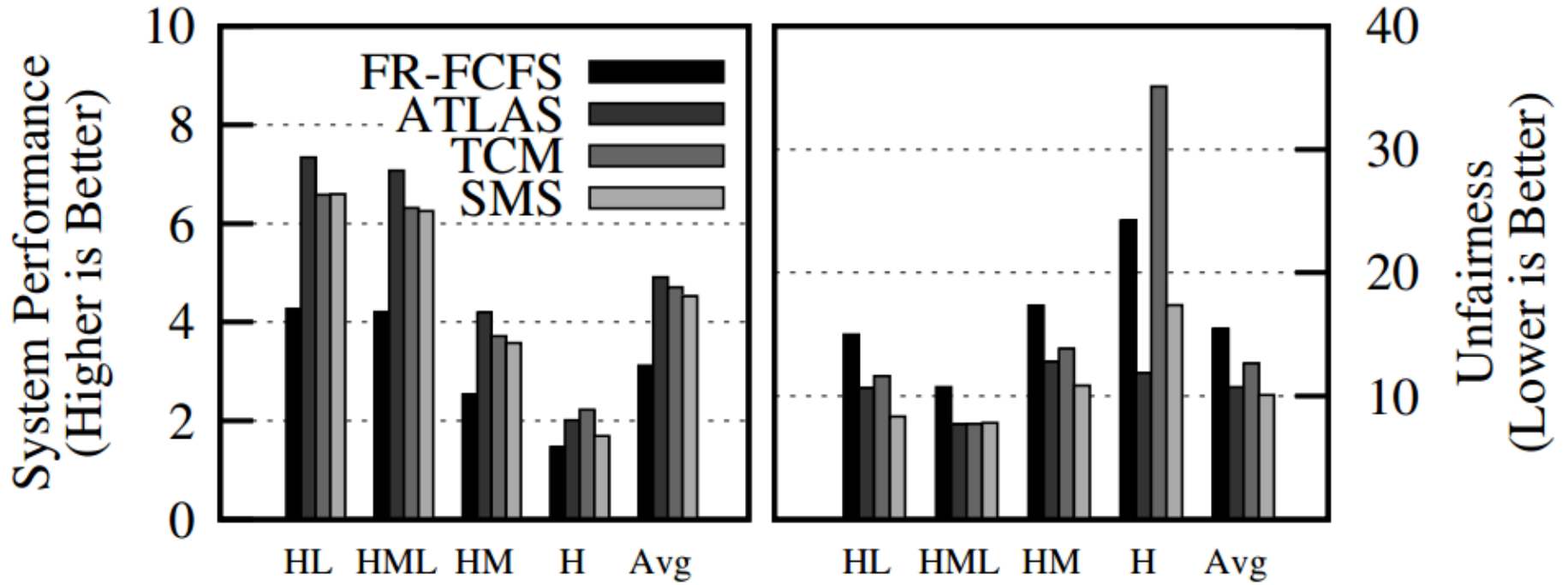
## CPU WS



## Frame Rate

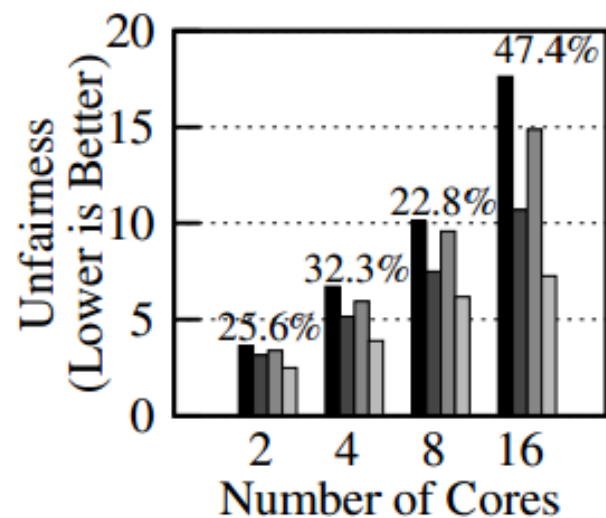
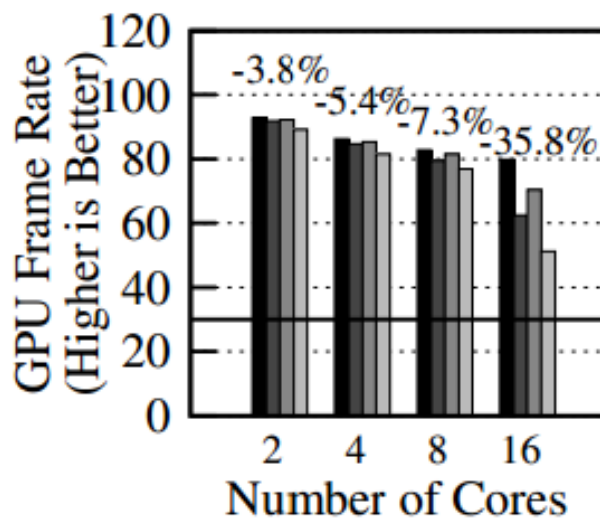
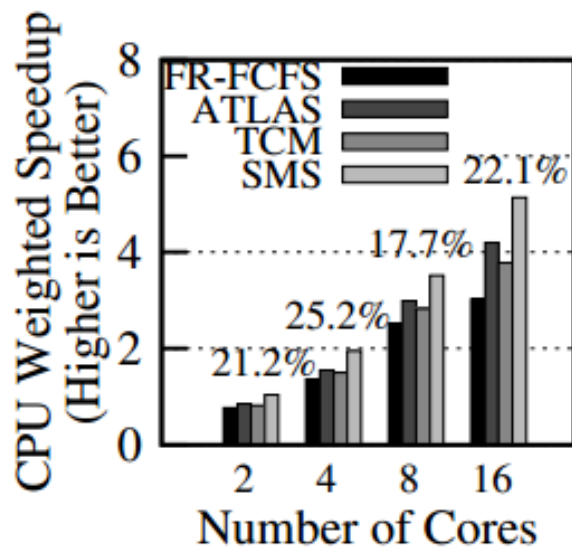


# CPU-Only Results

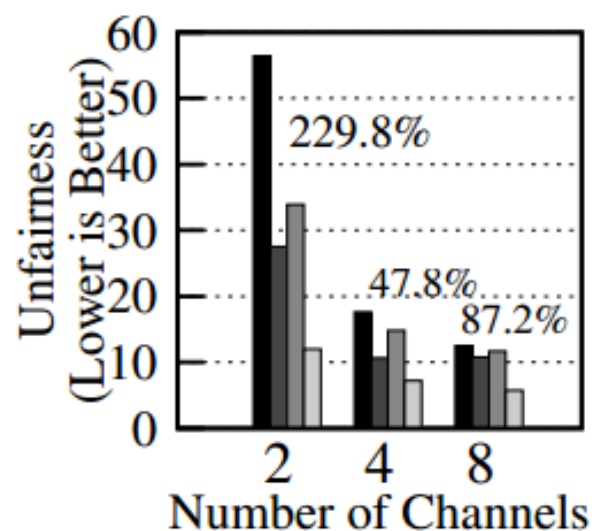
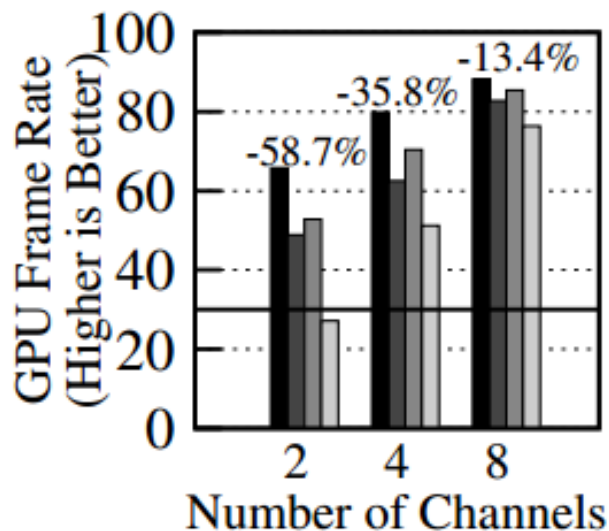
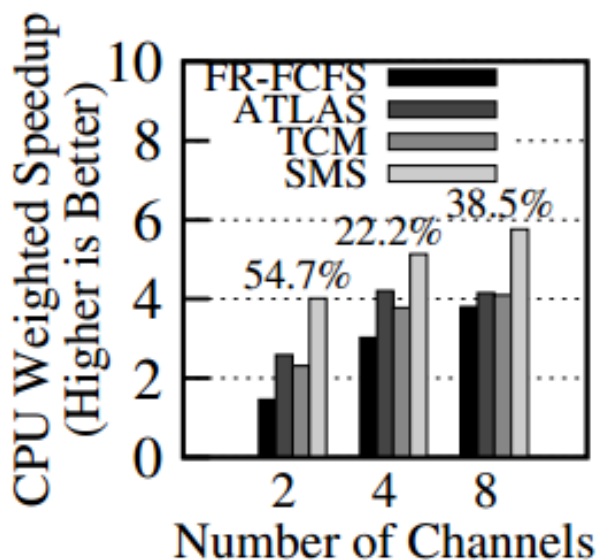


# Scalability to Number of Cores

---



# Scalability to Number of Memory Controllers



# Detailed Simulation Methodology

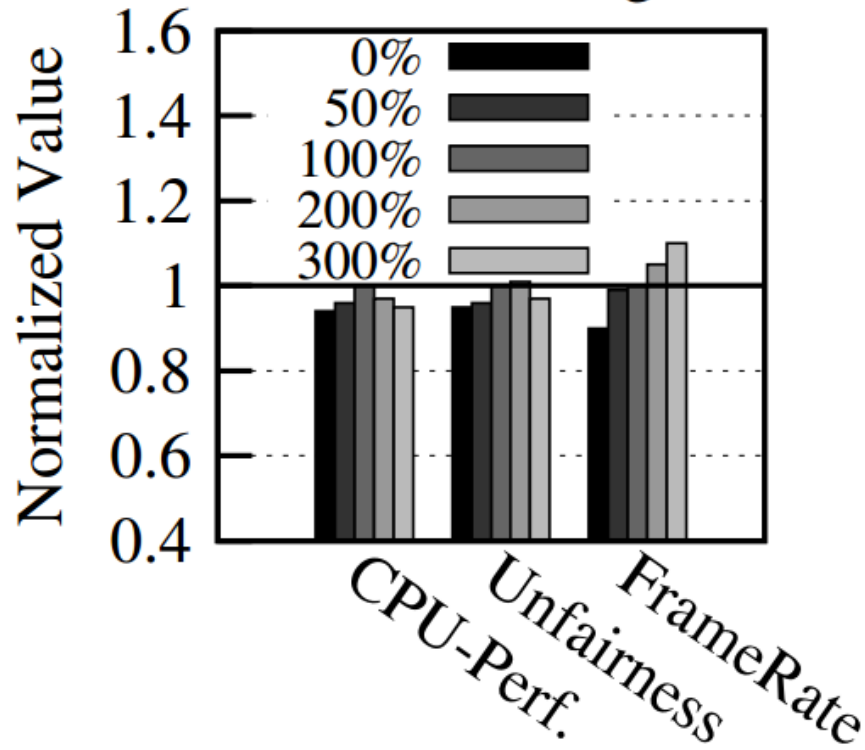
---

Number of cores	16	GPU Max throughput	1600 ops/cycle
Number of GPU	1	GPU Texture/Z/Color units	80/128/32
CPU reorder buffers	128 entries	DRAM Bus	64 bits/channel
L1 (private) cache size	32KB, 4 ways	DRAM row buffer size	2KB
L2 (shared) cache size	8MB, 16 ways	MC Request buffer size	300 entries
ROB Size	128 entries		

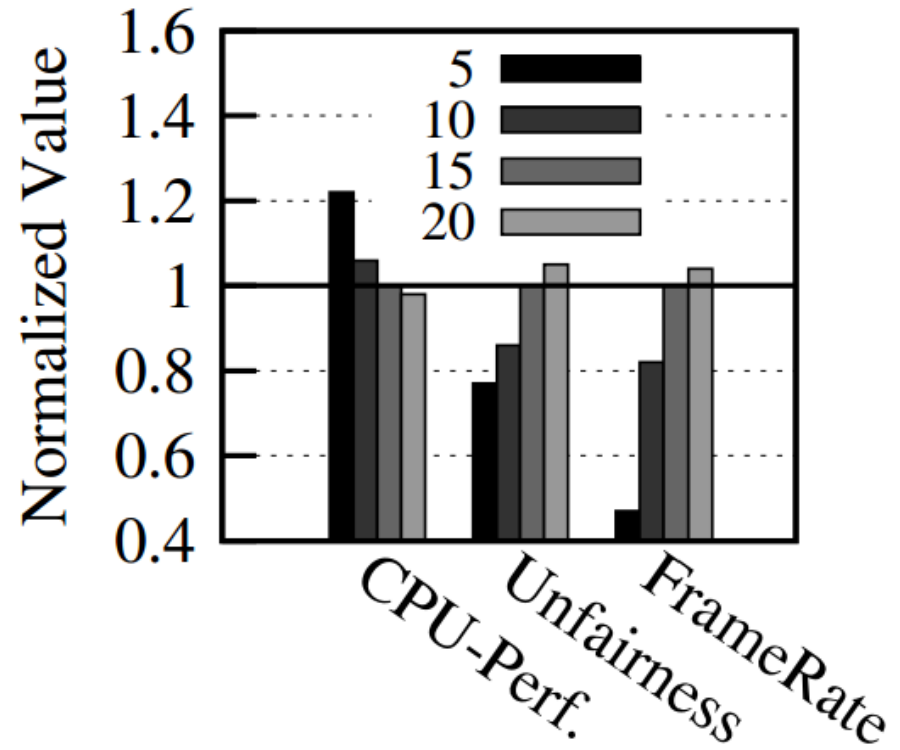
# Analysis to Different SMS Parameters

---

### Threshold Age



### DCS FIFO Size



# Global Bypass

---

- What if the system is lightly loaded?
  - Batching will increase the latency of requests
- Global Bypass
  - Disable the batch formation when the number of total requests is lower than a threshold

