Staged Memory Scheduling

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

Carnegie Mellon University, *AMD Research
June 12th 2012
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

Memory Scheduler

Core 1  Core 2  Core 3  Core 4

To DRAM
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 \([\text{Rixner+}, \text{ISCA’00}]\)
  - To maximize memory bandwidth

<table>
<thead>
<tr>
<th>Req 1</th>
<th>Row A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 2</td>
<td>Row B</td>
</tr>
<tr>
<td>Req 3</td>
<td>Row C</td>
</tr>
<tr>
<td>Req 4</td>
<td>Row A</td>
</tr>
<tr>
<td>Req 5</td>
<td>Row B</td>
</tr>
</tbody>
</table>

Currently open row: B
Three Principles of Memory Scheduling

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

<table>
<thead>
<tr>
<th>Req 1</th>
<th>Row A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 2</td>
<td>Row B</td>
</tr>
<tr>
<td>Req 3</td>
<td>Row C</td>
</tr>
<tr>
<td>Req 4</td>
<td>Row A</td>
</tr>
<tr>
<td>Req 5</td>
<td>Row B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Currently open row</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
</tbody>
</table>
Three Principles of Memory Scheduling

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

<table>
<thead>
<tr>
<th>Request</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 1</td>
<td>Row A</td>
</tr>
<tr>
<td>Req 2</td>
<td>Row B</td>
</tr>
<tr>
<td>Req 3</td>
<td>Row C</td>
</tr>
<tr>
<td>Req 4</td>
<td>Row A</td>
</tr>
<tr>
<td>Req 5</td>
<td>Row B</td>
</tr>
</tbody>
</table>

Currently open row: B

Older

Newer
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

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Intensity (MPKI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Intensity (MPKI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
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

Core 1  Core 2  Core 3  Core 4

Memory Scheduler

To DRAM
Introducing the GPU into the System

Memory Scheduler

To DRAM
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

Memory Scheduler

To DRAM
## Problems with Large Monolithic Buffer

<table>
<thead>
<tr>
<th>Req</th>
<th>Req</th>
<th>Req</th>
<th>Req</th>
<th>Req</th>
<th>Req</th>
<th>Req</th>
<th>Req</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
<tr>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
<td>Req</td>
</tr>
</tbody>
</table>

### Memory Scheduler
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

#### 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 centralised 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

Monolithic Scheduler

Core 1
Core 2
Core 3
Core 4
GPU

To DRAM

Memory Scheduler
SMS: Staged Memory Scheduling

Core 1 | Core 2 | Core 3 | Core 4 | GPU

Stage 1
Batch Formation

Stage 2
Batch Scheduler

Stage 3
DRAM Command Scheduler

Bank 1 | Bank 2 | Bank 3 | Bank 4

To DRAM
 SMS: Staged Memory Scheduling

Stage 1
- Batch Formation
  - Core 1
  - Core 2
  - Core 3
  - Core 4
  - GPU

Stage 2
- Batch Scheduler

Stage 3
- DRAM Command Scheduler
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4

To DRAM
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

Batch Formation

Core 1

Core 2

Core 3

Core 4

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1
Batch Formation
Core 1: Row A
Core 2: Row B
Core 3
Core 4

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

Core 1

Core 2

Core 3

Core 4

Batch Formation

Row A

Row B

Row B

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

Core 1

Core 2
Row C

Core 3

Core 4

Batch Formation

Row A

Row B

Row B

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

**Batch Formation**

Core 1

Core 2

Row C

Core 3

Core 4

Next request goes to a different row

Row A

Row B

Row B

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

Batch Formation

Core 1

Row A

Core 2

Row C

Row B

Core 3

Row D

Core 4

Next request goes to a different row

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

Batch Formation

Core 1
Row A
Row A

Core 2
Row B
Row B
Row C

Core 3
Row D

Core 4

Next request goes to a different row

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

Batch Formation

Core 1
Row A
Row A
Row A

Core 2
Row C
Row B
Row C

Core 3
Row D
Row D
Row D

Core 4
Row F

Next request goes to a different row

Time window expires

Batch Boundary

To Stage 2 (Batch Scheduling)
Stage 1: Batch Formation Example

Stage 1

Batch Formation

Core 1  Core 2  Core 3  Core 4

Batch Boundary

Time window expires

Row A  Row A  Row A  Row A

Row C  Row B  Row D  Row F

Row E  Row E  Row D  Row E

Next request goes to a different row

To Stage 2 (Batch Scheduling)
SMS: Staged Memory Scheduling

Stage 1
Batch Formation

Stage 2
Batch Scheduler

Stage 3
DRAM Command Scheduler

Core 1
Core 2
Core 3
Core 4
GPU

To DRAM
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 \( \rightarrow \) Scheduling policy needs to adapt to this

- **Solution**: Hybrid Policy

- At every cycle:
  - With probability \( p \): Shortest Job First \( \rightarrow \) Benefits the CPU
  - With probability \( 1-p \): Round-Robin \( \rightarrow \) Benefits the GPU

- System software can configure \( p \) based on the importance/weight of the GPU
  - Higher GPU importance \( \rightarrow \) Lower \( p \) value
SMS: Staged Memory Scheduling

Stage 1
- Batch Formation
  - Core 1
  - Core 2
  - Core 3
  - Core 4
  - GPU

Stage 2
- Batch Scheduler

Stage 3
- DRAM Command Scheduler
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4

To DRAM
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

Stage 1:
Batch
Formation
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler
Putting Everything Together

Stage 1:
Batch Formation

Stage 2:
Batch Scheduler

Stage 3:
DRAM Command Scheduler

Core 1 → Core 2 → Core 3 → Core 4 → GPU

Bank 1 → Bank 2 → Bank 3 → Bank 4
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy: SJF
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy: SJF
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy: RR
Putting Everything Together

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy: RR
Putting Everything Together

Stage 1:
Batch Formation

Stage 2:
Batch Scheduler

Stage 3:
DRAM Command Scheduler

Current Batch Scheduling Policy
RR
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)

* Based on a Verilog model using 180nm library
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

\[
CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight}
\]
Evaluated System Scenarios

- CPU-focused system
- GPU-focused system
Evaluated System Scenario: CPU Focused

- GPU has low weight (weight = 1)

- Configure SMS such that $p$, SJF probability, is set to 0.9
  - Mostly uses SJF batch scheduling $\rightarrow$ prioritizes latency-sensitive applications (mainly CPU)
Evaluated System Scenario: CPU Focused

- GPU has low weight (weight = 1)

\[ CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight} \]

- Configure SMS such that \( \rho \), SJF probability, is set to 0.9
  - Mostly uses SJF batch scheduling \( \rightarrow \) prioritizes latency-sensitive applications (mainly CPU)
Evaluated System Scenario: CPU Focused

- GPU has low weight (weight = 1)

\[ CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight} \]

- Configure SMS such that \( \rho \), SJF probability, is set to 0.9
  - Mostly uses SJF batch scheduling \( \rightarrow \) prioritizes latency-sensitive applications (mainly CPU)
SJF batch scheduling policy allows latency-sensitive applications to get serviced as fast as possible.
- SJF batch scheduling policy allows latency-sensitive applications to get serviced as fast as possible

**Performance: CPU-Focused System**

- SMS is much less complex than previous schedulers
- +17.2% over ATLAS

**Workload Categories**

- L
- ML
- M
- HL
- HML
- HM
- H
- Avg

**Graph**

- FR-FCFS
- ATLAS
- TCM
- SMS $p=0.9$
Evaluated System Scenario: GPU Focused

- GPU has high weight (weight = 1000)

- Configure SMS such that $p$, SJF probability, is set to 0
  - Always uses round-robin batch scheduling $\rightarrow$ prioritizes memory-intensive applications (GPU)
Evaluated System Scenario: GPU Focused

- GPU has high weight (weight = 1000)

\[ CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight} \]

- Configure SMS such that $p$, SJF probability, is set to 0
  - Always uses round-robin batch scheduling $\rightarrow$ prioritizes memory-intensive applications (GPU)
Evaluated System Scenario: GPU Focused

- GPU has high weight (weight = 1000)

\[ CGWS = CPU_{WS} + GPU_{Speedup} \times GPU_{Weight} \]

- Configure SMS such that $\rho$, SJF probability, is set to 0
  - Always uses round-robin batch scheduling \(\rightarrow\) prioritizes memory-intensive applications (GPU)
Round-robin batch scheduling policy schedules GPU requests more frequently.

Performance: GPU-Focused System

- +1.6% over FR-FCFS

Workload Categories:
- L
- ML
- M
- HL
- HML
- HM
- H
- Avg

Legend:
- FR-FCFS
- ATLAS
- TCM
- SMS

$p=0$
Round-robin batch scheduling policy schedules GPU requests more frequently.

- **Performance: GPU-Focused System**

  - **Workload Categories**
    - L, ML, M, HL, HML, HM, H, Avg

  - **Bar Graph**
    - CGWS
    - FR-FCFS
    - ATLAS
    - TCM
    - SMS

  - **Label:** +1.6% over FR-FCFS

  - **Note:** SMS is much less complex than previous schedulers.

- **Observation:**
  - SMS is much less complex than previous schedulers.
Performance at Different GPU Weights

- **System Performance**

  - **GPUweight**

<table>
<thead>
<tr>
<th>GPUweight</th>
<th>System Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>0.2</td>
</tr>
<tr>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td>1</td>
<td>0.6</td>
</tr>
<tr>
<td>10</td>
<td>0.8</td>
</tr>
<tr>
<td>1000</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Best Previous Scheduler**

Graph showing the relationship between GPU weight and system performance.
Performance at Different GPU Weights

- **Best Previous Scheduler**
- ATLAS
- TCM
- FR-FCFS

System Performance vs. GPU weight
Performance at Different GPU Weights

![Graph showing performance at different GPU weights. The x-axis represents GPU weights ranging from 0.001 to 1000, and the y-axis represents system performance ranging from 0 to 1. Two lines are plotted: the red line represents the Best Previous Scheduler, and the blue line represents the SMS. As the GPU weight increases, both lines approach 1, indicating improved system performance.](image-url)
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 12th 2012
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**

FR-FCFS | SMS-SJF | SMS_SJF-OoO | SMS-RR | SMS-RR-OoO
---|---|---|---|---
0.5 | 5.0 | 6.0 | 3.0 | 3.0

**GPU-Frame Rate**

FR-FCFS | SMS-SJF | SMS_SJF-OoO | SMS-RR | SMS-RR-OoO
---|---|---|---|---
90 | 80 | 90 | 80 | 80
Key Differences Between CPU and GPU

Memory Intensity

<table>
<thead>
<tr>
<th>L2 MPKI</th>
<th>Game01</th>
<th>Game03</th>
<th>Game05</th>
<th>Bench02</th>
<th>gromacs</th>
<th>cactusADM</th>
<th>mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Intensity</td>
<td>~4x difference</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graphic Applications

CPU Applications
MLP and RBL

- Key differences between a CPU application and a GPU application

![Memory Level Parallelism and Row Buffer Locality graphs](image-url)
CPU-GPU Performance Tradeoff

**CPU Performance**

<table>
<thead>
<tr>
<th>SJF Probability</th>
<th>Weighted Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0.5</td>
<td>5</td>
</tr>
<tr>
<td>0.1</td>
<td>5</td>
</tr>
<tr>
<td>0.05</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

**GPU Frame Rate**

<table>
<thead>
<tr>
<th>SJF Probability</th>
<th>Frame Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>0.5</td>
<td>25</td>
</tr>
<tr>
<td>0.1</td>
<td>20</td>
</tr>
<tr>
<td>0.05</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
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)
Unfairness (Lower is better)
Performance at Different Buffer Sizes

[Graph showing CPU performance, unfairness, and GPU frame rate for different buffer sizes (64, 128, 256, 384, 512 MB) for FR-FCFS, PARBS, ATLAS, and TCM.]
CPU and GPU Performance Breakdowns

CPU WS

Frame Rate
CPU-Only Results

![Bar chart showing system performance and unfairness for different models and configurations.](image)
Scalability to Number of Cores

- **CPU Weighted Speedup** (Higher is Better):
  - FR-FCFS
  - ATLAS
  - TCM
  - SMS

- **GPU Frame Rate** (Higher is Better):
  - Number of Cores: 2, 4, 8, 16
  - FR-FCFS: -3.8%, -5.4%, -7.3%, -35.8%
  - ATLAS: -3.8%, -5.4%, -7.3%, -35.8%
  - TCM: -3.8%, -5.4%, -7.3%, -35.8%
  - SMS: -3.8%, -5.4%, -7.3%, -35.8%

- **Unfairness** (Lower is Better):
  - Number of Cores: 2, 4, 8, 16
  - FR-FCFS: 25.6%, 32.3%, 22.8%, 47.4%
  - ATLAS: 25.6%, 32.3%, 22.8%, 47.4%
  - TCM: 25.6%, 32.3%, 22.8%, 47.4%
  - SMS: 25.6%, 32.3%, 22.8%, 47.4%
Scalability to Number of Memory Controllers
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>GPU Max throughput</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>16</td>
<td></td>
<td>1600 ops/cycle</td>
</tr>
<tr>
<td>Number of GPU</td>
<td>1</td>
<td>GPU Texture/Z/Color units</td>
<td>80/128/32</td>
</tr>
<tr>
<td>CPU reorder buffers</td>
<td>128 entries</td>
<td>DRAM Bus</td>
<td>64 bits/channel</td>
</tr>
<tr>
<td>L1 (private) cache size</td>
<td>32KB, 4 ways</td>
<td>DRAM row buffer size</td>
<td>2KB</td>
</tr>
<tr>
<td>L2 (shared) cache size</td>
<td>8MB, 16 ways</td>
<td>MC Request buffer size</td>
<td>300 entries</td>
</tr>
<tr>
<td>ROB Size</td>
<td>128 entries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis to Different SMS Parameters

**Threshold Age**

- CPU-Perf.
- Unfairness
- FrameRate

**DCS FIFO Size**

- CPU-Perf.
- Unfairness
- FrameRate

<table>
<thead>
<tr>
<th>Normalized Value</th>
<th>0%</th>
<th>50%</th>
<th>100%</th>
<th>200%</th>
<th>300%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Normalized Value</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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