GPUSync
A Framework for Real-Time GPU Management

Glenn Elliott
gelliott@cs.unc.edu

Bryan Ward
bcw@cs.unc.edu

James Anderson
anderson@cs.unc.edu

The University of North Carolina at Chapel Hill

Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135; ARO grant W911NF-09-1-0535; and an NSF graduate fellowship.
GPGPU

• Can execute general program code, a technique called GPGPU

• Can outperform a CPU by several orders of magnitude in data-parallel applications
Growth of theoretical peak GFLOPs for GPUs and CPUs
Growth of theoretical peak GFLOPs for GPUs and CPUs
Real-Time GPGPU Applications

- Potential in any data-parallel application with real-time constraints

- Example: Advanced Driver Assistance Systems
  - Computer vision:
    - Pedestrian, vehicle, sign detection, etc.
  - LIDAR processing
  - Sensor fusion
Target Platform

• We want to develop a system using components available today

• Current state of technology motivates the following platform:

  • Multicore system with one or more GPUs

  • Soft real-time (bounded deadline tardiness)
Task Model and Assumptions

- **Sporadic** task model
  - A task releases recurrent work as jobs
  - A **job requires one GPU** to execute a sequence of operations
  - A task may use a **different GPU per job**
  - A task may have **affinity** with its most recently used GPU
Real-Time GPU Challenges

• Managed by an operating system driver

• Often closed source

• Not originally designed for real-time

• Not directly schedulable like a CPU

• GPU operations are non-preemptive

• Explicit DMA operations for data movement
Example Hardware Architecture
Example Hardware Architecture

System Memory

Multicore Processor
- CPU₀
- CPU₁
- CPU₂
- CPU₃
- CPU₄
- CPU₅

Memory Controller

I/O Hub

May be on one chip
Example Hardware Architecture

- System Memory
- Memory Controller
- Multicore Processor
  - CPU0
  - CPU1
  - CPU2
  - CPU3
  - CPU4
  - CPU5
- I/O Hub
- Hierarchical, full-duplex, packet-switched bus
- PCIe Switch
  - GPU0
  - PCIe Switch
  - GPU1
  - PCIe Switch
  - GPU2
  - PCIe Switch
  - GPU3
Example Hardware Architecture
Copy Engines: DMA data between System and GPU Memory

Example Hardware Architecture
Example Hardware Architecture

Copy Engines: DMA data between System and GPU Memory

Execution Engine: Executes program code
GPU Engines:
1. Independent
2. Non-preemptive

Execution Engine: Executes program code
GPGPU Execution Pattern

CPU

GPU: CE₀

GPU: EE

GPU: CE₁

time

scheduled
GPU commands
suspended
GPGPU Execution Pattern

CPU

GPU: EE

GPU: CE₁

time

Issue commands to GPU and wait for completion

scheduled
GPU commands
suspended
GPGPU Execution Pattern

CPU

GPU: CE₀

GPU: EE

GPU: CE₁

time

Input

suspended

scheduled

GPU commands
GPGPU Execution Pattern

CPU

GPU: CE₀

GPU: EE

GPU: CE₁

time

input

scheduled
GPU commands
suspended
GPGPU Execution Pattern

CPU

GPU: CE₀

GPU: EE

GPU: CE₁

time

input

work

scheduled
GPU commands
suspended
GPGPU Execution Pattern

CPU

GPU: CE₀

GPU: EE

GPU: CE₁

input

work

scheduled

GPU commands

suspended

time
GPGPU Execution Pattern

CPU

GPU: CE₀
input

GPU: EE
work

GPU: CE₁
output

time

scheduled
GPU commands
suspended
GPGPU Execution Pattern

CPU

GPU: CE₀

GPU: EE

GPU: CE₁

time

input

work

output

scheduled
GPU commands
suspended
Affinity and State Migration

System Memory

Memory Controller

Multicore Processor

CPU0  CPU1  CPU2

CPU3  CPU4  CPU5

I/O Hub

PCIe Switch

GPU0

PCIe Switch

GPU1

PCIe Switch

GPU2

PCIe Switch

GPU3
Affinity and State Migration

System Memory

Memory Controller

Multicore Processor

- CPU₀
- CPU₁
- CPU₂
- CPU₃
- CPU₄
- CPU₅

I/O Hub

PCle Switch

State

GPU₁

PCle Switch

GPU₂

PCle Switch

GPU₃
Affinity and State Migration

System Memory Migration:
Data copied from GPU to GPU via System Memory
Affinity and State Migration
Affinity and State Migration

- **System Memory**
- **Memory Controller**
- **Multicore Processor**
  - CPU₀
  - CPU₁
  - CPU₂
  - CPU₃
  - CPU₄
  - CPU₅
- **I/O Hub**
- **PCIe Switch**
- **State**
- **GPU₁**
- **GPU₂**
- **GPU₃**
Affinity and State Migration

Peer-to-Peer Migration:
Data copied directly between GPUs (more efficient than system memory)
Affinity and State Migration

System Memory

Memory Controller

Multicore Processor
- CPU\(_0\)
- CPU\(_1\)
- CPU\(_2\)
- CPU\(_3\)
- CPU\(_4\)
- CPU\(_5\)

I/O Hub

PCIe Switch

GPU\(_0\)

State

PCIe Switch

GPU\(_2\)

PCIe Switch

GPU\(_3\)
Affinity and State Migration
Affinity and State Migration

Peer-to-Peer Migration:
Distant migrations are more costly than near ones
Affinity and State Migration
Affinity and State Migration
Affinity and State Migration

System Memory

Memory Controller

Multicore Processor

CPU₀  CPU₁  CPU₂
CPU₃  CPU₄  CPU₅

I/O Hub

PCIe Switch

GPU₀

PCIe Switch

GPU₁

PCIe Switch

GPU₂

State
Scheduling Problem

• What does a job need scheduled?
  • CPU time
  • Copy engine time
  • Execution engine time

• Constraints:
  • Copy and execution engines must be of same GPU
  • Job must execute on CPU to issue GPU commands
Scheduling Questions

1. Fixed- or dynamic-priority?
2. Partitioned, clustered, or global CPU scheduling?
3. Partitioned, clustered, or global GPU allocation?
4. How many jobs may use a GPU at once?
5. How do we distribute GPUs among jobs?
6. How do we arbitrate engine access?
7. Should copy engines have designated roles?
8. Should we break large memory copies up into pieces?
9. System memory or peer-to-peer migrations?
10. Techniques to maintain GPU affinity?
1. Fixed- or dynamic-priority?
2. Partitioned, clustered, or global CPU scheduling?
3. Partitioned, clustered, or global GPU allocation?
4. How many jobs may use a GPU at once?
5. How do we distribute GPUs among jobs?
6. How do we arbitrate engine access?
7. Should copy engines have designated roles?
8. Should we break large memory copies up into pieces?
9. System memory or peer-to-peer migrations?
10. Techniques to maintain GPU affinity?
1. Fixed- or dynamic-priority?
2. Partitioned, clustered, or global CPU scheduling?
3. Partitioned, clustered, or global GPU allocation?
4. How many jobs may use a GPU at once?
5. How do we distribute GPUs among jobs?
6. How do we arbitrate engine access?
7. Should copy engines have designated roles?
8. Should we break large memory copies up into pieces?
9. System memory or peer-to-peer migrations?
10. Techniques to maintain GPU affinity?
1. Fixed- or dynamic-priority?
2. Partitioned, clustered, or global
   CPU scheduling?
3. Partitioned, clustered, or global
   GPU allocation?
4. How many jobs may use a GPU at once?
5. How do we distribute GPUs among
   jobs?
6. How do we arbitrate engine access?
7. Should copy engines have designated
   roles?
8. Should we break large memory
    copies up into pieces?
9. System memory or peer-to-peer
    migrations?
10. Techniques to maintain GPU
    affinity?
1. Fixed- or dynamic-priority?
2. Partitioned, clustered, or global CPU scheduling?
3. Partitioned, clustered, or global GPU allocation?
4. How many jobs may use a GPU at once?
5. How do we distribute GPUs among jobs?
6. How do we arbitrate engine access?
7. Should copy engines have designated roles?
8. Should we break large memory copies up into pieces?
9. System memory or peer-to-peer migrations?
10. Techniques to maintain GPU affinity?
Reasonable Answers

1. Fixed- or dynamic-priority?
   \{rate monotonic, earliest-deadline-first\}

2. Partitioned, clustered, or global CPU scheduling?
   \{partitioned, clustered, global\}

3. Partitioned, clustered, or global GPU allocation?
   \{partitioned, clustered, global\}

4. How many jobs may use a GPU at once?
   \{1, \#engines, \(x\), unlimited\}

5. How do we distribute GPUs among jobs?
   \{do nothing, FIFO, priority-ordered, hybrid\}

6. How do we arbitrate engine access?
   \{FIFO, priority-ordered\}

7. Should copy engines have designated roles?
   \{no, yes: send & recv, yes: send/recv & P2P\}

8. Should we break large memory copies up into pieces?
   \{no, yes\}

9. System memory or peer-to-peer migrations?
   \{system memory, peer-to-peer\}

10. Techniques to maintain GPU affinity?
   \{no, yes\}
Reasonable Answers

At least 9,360 reasonable combinations!
(>20,000 with other configuration tweaks)

Which is best for:
Schedulability?
Response time?
Jitter?
GPUSync

- **Configurable testbed** for determining the **best** real-time GPGPU configurations

- Synchronization-based
  - Treat GPUs and engines as **shared resources**

- Features fall into three categories:
  1. **Allocation/arbitration** of GPUs/engines
  2. **Budget enforcement** mechanisms
  3. **Integration** with closed-source drivers and GPGPU runtime
GPU Allocation: High-level Design

$g$: number of GPUs \hspace{1cm} k$: total number of GPU tokens \\
$\rho$: number of tokens per GPU
GPU Allocation: High-level Design

Job requests GPU

**GPU Allocator**
- Manages $k = \rho \cdot g$ tokens

**Engine Locks**
- $\rho$ tokens per GPU
- $EE_0$, $CE_{0,0}$, $CE_{0,1}$

$g$ : number of GPUs
$k$ : total number of GPU tokens
$\rho$ : number of tokens per GPU
GPU Allocation: High-level Design

Job requests GPU

GPU Allocator

Manages

\[ k = \rho \cdot g \text{ tokens} \]

Engine Locks

\[ \rho \]

\[ \rho \]

\[ \rho \]

EE_0

CE_{0,0}

CE_{0,1}

GPU

GPU Allocator employs Cost Predictor

\[ g : \text{number of GPUs} \]

\[ k : \text{total number of GPU tokens} \]

\[ \rho : \text{number of tokens per GPU} \]
GPU Allocation: High-level Design

Job requests GPU

GPU Allocator employs Cost Predictor

Token granted, assigning GPU to job

A

B

C

k = ρ \cdot g tokens

\( g \): number of GPUs
\( k \): total number of GPU tokens
\( ρ \): number of tokens per GPU
GPU Allocation: High-level Design

- **Job requests GPU** (A)
- **GPU Allocator employs Cost Predictor**
- **Manages $k = \rho \cdot g$ tokens** (B)
- **Token granted, assigning GPU to job** (C)
- **Job competes for engine lock** (D)

$g$: number of GPUs  
$k$: total number of GPU tokens  
$\rho$: number of tokens per GPU
GPU Allocator employs Cost Predictor

Token granted, assigning GPU to job

Job competes for engine lock

Job uses an engine it has locked

Job requests GPU

GPU Allocator manages $k = \rho \cdot g$ tokens

$g$: number of GPUs  
$k$: total number of GPU tokens

$\rho$: number of tokens per GPU
Schedule w/ GPUSync

CPU

GPU\_x: CE\_0

GPU\_x: EE

GPU\_x: CE\_1

<table>
<thead>
<tr>
<th>time</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
<th>t_8</th>
<th>t_9</th>
<th>t_10</th>
<th>t_11</th>
<th>t_12</th>
</tr>
</thead>
</table>

- \( \text{scheduled} \)
- \( \text{holds token} \)
- \( \text{holds eng. lock} \)
- \( \text{suspended} \)
Schedule w/ GPUSync

$t_1$: Token requested

CPU

$\text{GPU}_x$: $\text{CE}_0$

$\text{GPU}_x$: $\text{EE}$

$\text{GPU}_x$: $\text{CE}_1$

<table>
<thead>
<tr>
<th>time</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
<th>$t_{10}$</th>
<th>$t_{11}$</th>
<th>$t_{12}$</th>
</tr>
</thead>
</table>
Schedule w/ GPUSync

- $t_1$: Token requested
- $t_2$: Token for GPU_x granted

CPU

GPU_x: CE_0

GPU_x: EE

GPU_x: CE_1

time: $t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \ t_8 \ t_9 \ t_{10} \ t_{11} \ t_{12}$

- scheduled
- holds token
- holds eng. lock
- suspended
Schedule w/ GPUSync

- $t_1$: Token requested
- $t_2$: Token for GPU$_x$ granted
- $t_3$: Copy engine lock requested

CPU

GPU$_x$: CE$_0$

GPU$_x$: EE

GPU$_x$: CE$_1$

Time

$[t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}]$
Schedule w/ GPUSync

- $t_1$: Token requested
- $t_2$: Token for GPU$_x$ granted
- $t_3$: Copy engine lock requested
- $t_4$: Copy engine lock granted

- GPU$_x$: CE$_0$
- GPU$_x$: EE
- GPU$_x$: CE$_1$

Nodes:
- scheduled
- holds token
- holds eng. lock
- suspended

Time:
- $t_1$, $t_2$, $t_3$, $t_4$
- $t_5$, $t_6$, $t_7$
- $t_8$, $t_9$, $t_{10}$
- $t_{11}$, $t_{12}$
Schedule w/ GPUSync

$t_1$: Token requested

$t_2$: Token for $\text{GPU}_x$ granted

$t_3$: Copy engine lock requested

$t_4$: Copy engine lock granted

$t_5$: Copy engine lock released

$\text{CPU}$

$\text{GPU}_x: \text{CE}_0$

$\text{GPU}_x: \text{EE}$

$\text{GPU}_x: \text{CE}_1$

$time$:

$t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}$
Schedule w/ GPUSync

- \( t_1 \): Token requested
- \( t_2 \): Token for \( \text{GPU}_x \) granted
- \( t_3 \): Copy engine lock requested
- \( t_4 \): Copy engine lock granted
- \( t_5 \): Copy engine lock released
- \( t_{12} \): Token for \( \text{GPU}_x \) released

CPU

\( \text{GPU}_x: \text{CE}_0 \)

\( \text{GPU}_x: \text{EE} \)

\( \text{GPU}_x: \text{CE}_1 \)

time

- \( t_1, t_2, t_3, t_4 \)
- \( t_5, t_6, t_7 \)
- \( t_8, t_9, t_{10} \)
- \( t_{11}, t_{12} \)
Schedule w/ GPUSync

GPU Critical Section

CPU

GPU<sub>x</sub>: CE<sub>0</sub>

GPU<sub>x</sub>: EE

GPU<sub>x</sub>: CE<sub>1</sub>

t<sub>1</sub> t<sub>2</sub> t<sub>3</sub> t<sub>4</sub> t<sub>5</sub> t<sub>6</sub> t<sub>7</sub> t<sub>8</sub> t<sub>9</sub> t<sub>10</sub> t<sub>11</sub> t<sub>12</sub>

- **t<sub>2</sub>:** Token for GPU<sub>x</sub> granted
- **t<sub>12</sub>:** Token for GPU<sub>x</sub> released

- **scheduled**
- **holds token**
- **holds eng. lock**
- **suspended**
GPU Allocator

- Each GPU is associated with $\rho$ tokens
- GPU Allocator assigns GPU tokens to jobs
  - Job assigned GPU of corresponding token
  - Manages $k = \rho \cdot g$ tokens
- Configurable hybrid priority-/FIFO-ordered queueing

$g$ : number of GPUs  $\rho$ : number of tokens per GPU  $k$ : total number of GPU tokens
Token Queues

$max \text{ length } f$

$g$ : number of GPUs  
$\rho$ : number of tokens per GPU  
$f$ : maximum FIFO length  
$k$ : total number of GPU tokens
Token Queues

- **$g$**: number of GPUs
- **$\rho$**: number of tokens per GPU
- **$f$**: maximum FIFO length
- **$k$**: total number of GPU tokens

**GPU requests spill into priority-ordered queue when FIFOs are full**

- Single priority-ordered queue
- Fixed-length FIFO queues

**Max length $f$**

- $\rho$ tokens
- $\rho \cdot g$ tokens

**Cost Predictor**
- **Engine Locks**
- **GPU Allocator**
Token Queues

GPU requests spill into priority-ordered queue when FIFOs are full

System designer selects $f$ and $\rho$

max length $f$

$\rho$ tokens

$\rho \cdot g$ tokens

single priority-ordered queue

fixed-length FIFO queues

$g$: number of GPUs
$\rho$: number of tokens per GPU
$f$: maximum FIFO length
$k$: total number of GPU tokens
Token Queues

GPU requests spill into priority-ordered queue when FIFOs are full

max length $f$

System designer selects $f$ and $\rho$

$\rho$ tokens

$\rho \cdot g$ tokens

single priority-ordered queue

fixed-length FIFO queues

Large $\rho$ increases GPU engine parallelism

$g$: number of GPUs
$\rho$: number of tokens per GPU
$f$: maximum FIFO length
$k$: total number of GPU tokens
Token Queues

Smaller $f$ promotes migration/load balancing

GPU requests spill into priority-ordered queue when FIFOs are full

max length $f$

System designer selects $f$ and $\rho$

GPU requests spill into priority-ordered queue when FIFOs are full

single priority-ordered queue

fixed-length FIFO queues

$g$: number of GPUs

$\rho$: number of tokens per GPU

$f$: maximum FIFO length

$k$: total number of GPU tokens

Large $\rho$ increases GPU engine parallelism
Values of $f$ and $\rho$ may affect schedulability analysis. Possible trade-offs between parallelism, performance, and schedulability.

$g$: number of GPUs
$\rho$: number of tokens per GPU
$f$: maximum FIFO length
$k$: total number of GPU tokens
Cost Predictor

- GPU Allocator **may choose** from multiple non-full FIFOs when enqueuing a request
  - Available FIFOs may correspond to different GPUs
- Cost Predictor **estimates the cost of migration**
  - Based upon **historical observations**
- Selects FIFO that **minimizes expected job response time** of GPU-requesting job
  - **May migrate** in order to **avoid heavily contended GPU**
Engine Locks

• Job **competes with other token holders** for GPU engines

  • At most $\rho - 1$ competitors for any engine request

• Job may only hold **one engine lock at a time**

  • Exception: peer-to-peer state migration (see paper for details)

• Engine locks may be either **FIFO-** or **priority-ordered**
Budget Enforcement

- **Non-preemption of GPU engines** raises budget enforcement challenges

- Can only **recover** and **isolate** budget overruns

- Three techniques:
  1. **Signal** the user application
  2. **Early releasing** (use budget and deadline of next job)
  3. **Bandwidth inheritance** (we’ll skip this in the interest of time)
Early Releasing

Job with 10ms budget, 15ms period, implicit-deadline
Early Releasing

Job with 10ms budget, 15ms period, implicit-deadline

Budget

- scheduled
- holds eng. lock
- holds token
- suspended
- release
- deadline
- job completion
Early Releasing

Job with 10ms budget, 15ms period, implicit deadline.

Consume budget of next job with postponed deadline.
Early Releasing
Job with 10ms budget, 15ms period, implicit deadline

Budget exhausted

CPU

GPU\textsubscript{x}: CE/EE

Budget

Remaining budget discarded

Consume budget of next job with postponed deadline

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- scheduled
- holds eng. lock
- holds token
- suspended
- release
- job completion
- deadline
Early Releasing

Job with 10ms budget, 15ms period, implicit deadline

- Budget exhausted
- Remaining budget discarded
- Consume budget of next job with postponed deadline

```
CPU
```

```
GPU_x: CE/EE
```

```
Budget
```

```
time
```

- scheduled
- holds eng. lock
- holds token
- suspended
- release
- job completion
- deadline
Evaluation
Implementation

• GPUSync implemented in Litmus\textsuperscript{RT}, a real-time patch to Linux co-developed at UNC and MPI

  • Adds \textasciitilde20k lines of code to Litmus\textsuperscript{RT}

• NVIDIA driver and used the CUDA 5.0 runtime

• Test Platform:

  • 2x 6-core Westmere-class CPUs

  • 8 NVIDIA Quadro K5000 (two copy engines each)
High-level Evaluation Questions

• Do GPUSync configuration parameters affect observed performance?

• Do tasks receive needed CPU/GPU processor time?

• Does affinity-aware token assignment reduce migration frequency?

• How do migrations affect performance?
Test Setup

• Clustered earliest deadline first (C-EDF) scheduler

• Task set: 28 CPU-only tasks, 34 GPU-using tasks

• Three scenarios:
  
  • **Normal**: Job execution time adheres to budget
  
  • **Aberrant**: Some GPU-using jobs randomly use 10x execution engine time
  
  • **Aberrant w/ Budget Enforcement**: Use early-releasing policy
Budget Results

Let’s look at the total allocated Execution Engine time given to a particular GPU-using task.
Budget Results

$T_1$: Needs 2.5ms of Execution Engine time every 15ms:

0.166 “execution engine utilization”
Budget Results

T₁: Needs 2.5ms of Execution Engine time every 15ms:

0.166 “execution engine utilization”

After 150 seconds of real time, T₁ has received ~22.5 seconds of Execution Engine time
Budget Results

T₁: Needs 2.5ms of Execution Engine time every 15ms:

0.166 “execution engine utilization”

After 150 seconds of real time, T₁ has received
~22.5 seconds of Execution Engine time
**Budget Results**

$T_1$: Needs 2.5ms of Execution Engine time every 15ms:

*0.166 “execution engine utilization”*

<table>
<thead>
<tr>
<th></th>
<th>Desired</th>
<th>Normal</th>
<th>Aberrant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budget</td>
<td>0.153</td>
<td>0.165</td>
<td>0.169</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Budget Enforced:** 0.153
- **Normal:** 0.165
- **Aberrant:** 0.169

Equations:

- [1] C-EDF $T_1 = 0.165x - 170.30$
- [2] C-EDF $T_1$ Abr. = 0.169x - 214.08
- [3] C-EDF $T_1$ Abr. w/ Budget = 0.153x - 13.03
T₁: Needs 2.5ms of Execution Engine time every 15ms: 0.166 “execution engine utilization”
Migration Frequency

Observed migrations over 180 seconds

- Task T1: 11857 (GPU Reuse), 110 (Near Migration), 0 (Distant Migration)
- Task T2: 2382 (GPU Reuse), 11 (Near Migration), 4 (Distant Migration)
Migration Frequency

Observed migrations over 180 seconds

<table>
<thead>
<tr>
<th>Task</th>
<th>GPU Reuse</th>
<th>Near Migration</th>
<th>Distant Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>11857</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>T2</td>
<td>2382</td>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>

GPU reuse dominates.
Migration Frequency

- **Number of Migrations**
  - Task T1: 11857
  - Task T2: 2382

- **GPU Reuse** dominates
- Near migrations favored over distant ones

**Observed migrations over 180 seconds**
Affinity-aware token assignment reduced migrations (arbitrary assignment would lead to ~50% distant migrations).
Computer Vision Evaluation

- **30 tasks** performed “feature tracking” on a separate video stream at various frame rates

  - Modified vision code by Hwangbo et al. (CMU) to run with GPUSync/LITMUSRT

- **Three** GPU cluster sizes: 1, 2, and 4

- Tested **priority-** and **FIFO-** ordered engine locks

- Tested **peer-to-peer** and **system memory** migration methods
Observed Response Times, FIFO Engine Locks

CDF of Response Time As % of Period for C-EDF, FIFO Queues

Response Time (% of period)

Probability(Response Time) ≤ x

[1] g = 1
[2] g = 2 (P2P)
[3] g = 2 (SysMem)
[4] g = 4 (P2P)
[5] g = 4 (SysMem)

(59408%)

(47820%)
Observed Response Times, FIFO Engine Locks

Partitioned GPUs: ~90% of jobs had response times < 160% of period
Observed Response Times, FIFO Engine Locks

CDF of Response Time As % of Period for C-EDF, FIFO Queues

System Memory Migration results in unscheduled task set
Observed Response Times, FIFO Engine Locks

CDF of Response Time As % of Period for C-EDF, FIFO Queues

Peer-to-Peer Clusters: ~95% of jobs had response times < ~125% of period

- versus -

Partitioned: ~82% of response times < ~125%
Observed Response Times, Priority-Ordered Engine Locks

CDF of Response Time As % of Period for C-EDF, Priority Queues
Observed Response Times, Priority-Ordered Engine Locks

Partitioned GPUs: 100% of response times < ~80% of period (no deadlines missed)
Observed Response Times, Priority-Ordered Engine Locks

Observed performance significantly affected by engine lock prioritization and migration methods.
Observed Response Times, Priority-Ordered Engine Locks

Observed performance significantly affected by engine lock prioritization and migration methods.
Conclusion

- Designed and implemented a framework for real-time GPU management
  - Flexible configuration to support different schedulers and analytical techniques
  - Budgeting policies
  - Closed-source driver support
  - First to support deterministic peer-to-peer migrations

- GPUSync source code: www.litmus-rt.org
Future Work

- Further explore the GPUSync configuration space (>20,000 “reasonable” configurations) to optimize:
  - Overhead-aware schedulability analysis
  - Observed performance
- Support graph-based workloads (different affinity characteristics)
Thank you!

Questions?
Backup Slides
Integration Issues

- GPUSync supports use of **closed-source** drivers and runtimes

- Two main issues:

  1. **“Non-real-time”** allocation methods
     - Locking protocols **effectively remove** closed-source software from allocation decisions

  2. **Interrupt** and user-space runtime **“worker” threads**
     - Example…
Interrupts & Worker Threads

(CUDA 4.2 and later)
Interrupts & Worker Threads

$W_{i,j}$ worker thread wakes $J_i$ when $J_i$ blocks for $GPU_j$ to complete.

$J_i$

$W_{i,j}$

$I_j$

$J_{i, GPU_j}$

(notification)

(notification)

(interrupt)

(CUDA 4.2 and later)
Interrupts & Worker Threads

$W_{i,j}$ worker thread wakes $J_i$ when $J_i$ blocks for $GPU_j$ to complete.

$I_j$ process all interrupts from $GPU_j$. (CUDA 4.2 and later)
Interrupts & Worker Threads

Must be scheduled with priority no less than $J_i$

(CUDA 4.2 and later)
Interrupts & Worker Threads

Must be scheduled with priority no less than $J_i$

$I_j$ scheduled with highest priority of **blocked** GPU$_j$ engine lock holders

(CUDA 4.2 and later)
Interrupts & Worker Threads

- \( J_i \): GPU_\text{j} engine lock holders
- \( W_{i,j} \): scheduled with priority of \( J_i \) when \( J_i \) is blocked
- \( I_j \): scheduled with highest priority of blocked \( \text{GPU}_j \) engine lock holders
- \( J_{i,\text{GPU}_j} \)

(CUDA 4.2 and later)

Must be scheduled with priority no less than \( J_i \)