A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling

Hyeonsu Lee, Jaehun Roh, Euiseong Seo
School of Software
Sungkyunkwan University
Contents

• Introduction
• Motivation
• Related Work
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Introduction

- GPU is specialized in parallel computing workload
  - Embedded systems can earn substantial benefit from GPU computing

NVIDIA's Machine Learning Benchmark

13.6x Faster

Health Analysis

Face Detection

Voice Recognition

A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling
• Introduction
• **Motivation**
• Related Work
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Motivation

- Example: Autonomous vehicle control systems using GPU

High Priority
Evasion Path Finder

W..wait!

Object Detection

Run

Crash

A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling
Motivation

- Priority inversion problem on GPU
  - No GPU Preemption HW and Interfaces

Priority: Kernel B > Kernel A
Motivation

- Priority inversion problem on GPU
  - No GPU Preemption HW and Interfaces

Embedded Systems Need Preemptive Priority Scheduling of GPU Kernels
• Introduction
• Motivation
• **Related Work**
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Related Work - Hardware

● HW extension for preemptive GPU scheduling
  • Experimental architecture was proposed and evaluated with simulation
  • Nvidia commercialized GPUs with HW preemption
    (But, they have not disclosed API for its, yet)

● Limitations of HW GPU preemption
  • Up to 100μs context switching time from massive amount of state saving/restoring
  • Increased complexity in HW design and thus cost to manufacture
Related Work - Software

- Kernel slicing approach

  - High Priority Sub Kernel
  - High Priority Sub Kernel
  - High Priority Sub Kernel

  Determined through static analysis

- Thread-block-level scheduling approach

  - Thread Block
  - Block Task
  - Delayed

  Repeat

  Preemption?

  Yes
  No

  Switching to the highest-priority kernel

We propose a software preemption solution that
- Immediately aborts currently running kernel
- Re-executes aborted kernels later
• Introduction
• Motivation
• Related Work
• **Background**
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Background – Shared System Memory

- GPU memory management in embedded systems

A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling

CPU Virtual Address Space

- Host-side Buffer
- Entry
- Shared

- MMU
- CPU

GPU Virtual Address Space

- Kernel Buffer 1
- Kernel Buffer 2
- Entry
- Shared

- IOMMU
- GPU

CPU can directly access GPU memory
Background – Shared Kernel Buffers

- Producer-consumer relationships through sharing kernel buffers
- Blind abortion and re-execution may corrupt data delivered from its producer
• Introduction
• Motivation
• Related Work
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Our design is structured in

**GPU kernel abortion:**
To immediately abort the low priority kernel
Our design is structured in

- GPU memory context saving/restoring:
  - For reexecution of a preempted kernel
Design - Preemption with Transactionized Kernels

- Our design is structured in

Transaction of GPU kernel execution:
- To prevent simultaneously access shared kernel buffer

[Diagram showing scheduling queue, dirty data snapshot, running queue, and GPU with an inconsistency of a kernel buffer highlighted]
Design - Transactionization Scheme

- GPU kernel transaction structure for re-execution of kernel
  - Procedures from snapshotting to kernel finish
Design – Example of Transactionization Scheme

- Pseudo-preemptive priority scheduling flow of Transactionized GPU Kernels

A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling
• Introduction
• Motivation
• Related Work
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Implementation – Snapshot Mechanism

- Implementing snapshot mechanism through GPGPU programming model
Implementation – Snapshot Memory Management

- Allocating snapshot memory when allocating a kernel buffer
- Tracking kernel buffer addresses thorough metadata of kernel when launching kernel
Implementation – Snapshot Process

- Snapshot process is implemented by kernel threads

Snapshots Thread Pseudo Code

```c
while(1){
    if(!is_next_kernel)
        interruptible_sleep();
    else
        roll_back();
    else
        snapshot();
    submit_to_GPU();
}
```

Snapshot Cancel Range

- High Priority Task
  - Kernel
  - Launch
- Low Priority Task
  - Snapshot Thread-1
  - 1. Launch (wake-up)
  - 2. Snapshot Cancel
  - 3. Switching
  - 4. Launch
- Snapshot Thread-2
Implementation – Preemption Mechanism

- Pre-preemption process
  - Minimizing process of aborting scheduled kernels

- Post-preemption process
  - Recovering aborted kernels to the scheduling queue
  - Running a kernel launch and a post-preemption process

![Diagram showing pre- and post-preemption processes]

A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling
• Introduction
• Motivation
• Related Work
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Evaluation Setup (1)

- Target device environment

<table>
<thead>
<tr>
<th>Board</th>
<th>Odroid-XU3 Rev. 2</th>
<th>SoC</th>
<th>Exynos 5422</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4×Cortex <a href="mailto:A15@2.0Ghz">A15@2.0Ghz</a></td>
<td>GPU</td>
<td>Mali-T628 MP6</td>
</tr>
<tr>
<td></td>
<td>4×Cortex <a href="mailto:A7@1.4Ghz">A7@1.4Ghz</a></td>
<td></td>
<td>@600Mhz</td>
</tr>
<tr>
<td></td>
<td>2MB Shared L2 Cache</td>
<td></td>
<td>256KB L2 Cache</td>
</tr>
<tr>
<td>Memory</td>
<td>2GB LPDDR3 RAM @ 933MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage</td>
<td>32GB eMMC 5.0 HS400 Flash Storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Linux 3.10.72 with Mali r5p0-06rel0 driver</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Rodinia benchmark suite 3.1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k-Nearest Neighbors</td>
<td>NN</td>
<td>Dense Linear Algebra</td>
<td>1</td>
<td>60 MB</td>
<td>60.35 MB</td>
<td>10.38 ms</td>
<td>0.68 s</td>
</tr>
<tr>
<td>Back Propagation</td>
<td>BP</td>
<td>Unstructured Grid Algebra</td>
<td>2</td>
<td>6.79 MB</td>
<td>9.42 MB</td>
<td>15.74 ms</td>
<td>0.39 s</td>
</tr>
<tr>
<td>PathFinder</td>
<td>PF</td>
<td>Dynamic Programming</td>
<td>5</td>
<td>38.84 MB</td>
<td>39.25 MB</td>
<td>609 ms</td>
<td>3.50 s</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>BF</td>
<td>Graph Traversal</td>
<td>24</td>
<td>21.94 MB</td>
<td>37.55 MB</td>
<td>4.57 ms</td>
<td>0.91 s</td>
</tr>
<tr>
<td>Kmeans</td>
<td>KM</td>
<td>Dense Linear Algebra</td>
<td>38</td>
<td>67.6 MB</td>
<td>130.38 MB</td>
<td>115.01 ms</td>
<td>8.40 s</td>
</tr>
<tr>
<td>Hotspot3D</td>
<td>3D</td>
<td>Structured Grid</td>
<td>100</td>
<td>24 MB</td>
<td>24.3 MB</td>
<td>10.78 ms</td>
<td>11.82 s</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>LUD</td>
<td>Dense Linear Algebra</td>
<td>190</td>
<td>4.15 MB</td>
<td>26.92 MB</td>
<td>2.35 ms</td>
<td>10.18 s</td>
</tr>
<tr>
<td>Needleman-Wunsch</td>
<td>NW</td>
<td>Dynamic Programming</td>
<td>255</td>
<td>32.47 MB</td>
<td>104.45 MB</td>
<td>0.29 ms</td>
<td>0.67 s</td>
</tr>
<tr>
<td>Gaussian Elimination</td>
<td>GE</td>
<td>Data Mining</td>
<td>510</td>
<td>0.50 MB</td>
<td>1.18 MB</td>
<td>0.26 ms</td>
<td>0.40 s</td>
</tr>
<tr>
<td>Myocyte</td>
<td>MC</td>
<td>Biological Simulation</td>
<td>3,913</td>
<td>1.76 MB</td>
<td>2.12 MB</td>
<td>0.13 ms</td>
<td>8.63 s</td>
</tr>
<tr>
<td>CFD Solver</td>
<td>CF</td>
<td>Fluid Dynamics</td>
<td>14,004</td>
<td>7.24 MB</td>
<td>19.3 MB</td>
<td>5.53 ms</td>
<td>99.99 s</td>
</tr>
</tbody>
</table>
Evaluation Setup (2)

● Micro benchmark (M-Bench)
  ○ We made simple GPGPU workload to be used for lower priority workload than Rodinia benchmark’s workloads
  ○ M-Bench running in background disturb the high priority task

● Priority configuration
  ○ [E] : Emergency priority
    ▪ This priority is the highest priority and does not require snapshotting
  ○ [C] : Common priority
    ▪ This priority is a higher priority than M-Bench
  ○ [N] : Normal environment
    ▪ This is result of evaluation in vanilla kernel
Performance of High Priority Task

- Performance of a high priority task
  - Degrading performance up to $186.9 \times$ depending on low priority tasks on normal environment
  - Guaranteeing consistent performance regardless of background load on Transactionization scheme
Scheduling Delay on Original Environment

- Histogram of scheduling delay of high priority kernels
  - Scheduling delay depending on the running kernel length in normal environment
- Our approach suppressed the launch delay of high-priority kernels within 18\(\mu\)s

![Histogram](image)

**Normal environment**

- Increased delay due to kernel length

**Transactionization scheme**

- 99.9% percentile within 18\(\mu\)s
Analysis of Preemption Delay

- There is no delay caused by eviction in the preemption process since it exceeds the majority of $18\mu s$.
Our scheme has significant overhead due to snapshotting

- Advantages of performance improvement due to launch without snapshotting at emergency priority

Using invalidated kernel buffer
Discussion

- Reducing snapshot overhead through high bandwidth memory (HBM)
  - Exynos 8890 provides up to 51 GB/s memory bandwidth while the device used in our evaluation presents 28.7 GB/s
  - Snapshot process is more efficient when using HBM because it implements memory copying using ARM's vector instruction set

- Selective snapshot process using compile-time hints
  - GPU kernel is idempotent sometimes
  - If enabling to detect idempotent kernels, then selective snapshot processes can be used
• Introduction
• Motivation
• Related Work
• Background
• Our Approach
  • Design
  • Implementation
• Evaluation
• Conclusion
Conclusion

- A GPU kernel transactionization for preemptive scheduling
  - We proposed a GPU kernel transactionization scheme that enables immediate abortion and re-execution of a GPU kernel
  - Based on the transactionization scheme, we developed a pseudo-preemptive GPU kernel scheduler

- Evaluation results showed that the scheduling delay for the urgent task was reduced to approximately 18μs

- Source code is available to public at
  - https://github.com/Hyunsu-Lee/psched_gpu
Thank you

Q&A
- Multiple priority evaluation