Protecting Real-Time GPU Kernels in Integrated CPU-GPU SoC Platforms

Waqar Ali, Heechul Yun
University of Kansas
GPU in Autonomous CPS

- Needed for real-time processing of high bandwidth sensor data (e.g., vision), deep neural networks, AI, etc.
- Must meet size, weight, and power (SWaP) and cost constraints
Discrete GPU

- GPU uses **dedicated** GPU memory
- Good for performance, but bad for cost & SWaP
Integrated CPU-GPU SoC

- CPU and GPU use the same **shared** DRAM
- Good for cost, SWaP, data movement, ... *BUT*

![Integrated CPU-GPU SoC Diagram](image)
Memory Bandwidth Contention

Co-scheduling memory intensive CPU task affects GPU performance on Integrated CPU-GPU SoC
CPU Memory Access Characteristic

• “Low Latency (LL) – the dominant characteristics of memory traffic coming from the CPUs are random, small size accesses (typically cache line fills) that are sporadic in nature. Key requirement for CPU accesses is low latency so as to provide maximum thread execution performance.”
  

• Prioritizing CPU traffic over GPU is usually good, but bad for real-time GPU kernels
Outline

• Motivation

• BWLOCK++
  – Memory bandwidth throttling
  – Binary instrumentation
  – Throttle fair scheduler (TFS)
  – Schedulability analysis

• Evaluation

• Conclusion
• **Goal**: automatically protect real-time GPU kernel while minimizing CPU **throughput** impact
Real-Time Core

• Dedicated core to schedule ALL real-time tasks
  – GPU kernels from diff tasks are serialized* anyway

(*) They are time multiplexed rather than being truly concurrent. N. Otterness et al., “An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads.” RTAS, 2017
Memory Bandwidth Throttling

- MemGuard*: Throttle CPU core’s memory bandwidth using its performance counters

(*) Yun et al., “MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms.” RTAS’13
Real-Time GPU Kernel Protection

• Idea: *Throttle* CPU memory bandwidth usage *while* running real-time GPU kernels to protect their performance

• Questions
  • How much do we need to throttle?
  • When and how to start/stop throttling?
  • How to minimize CPU throughput loss?
  • How to analyze schedulability?
Determining Throttling Budget

- Based on each GPU task’s bandwidth sensitivity

1) Inject CPU memory traffic
2) Vary CPU bandwidth budget
3) Measure GPU kernel’s timing
4) Find tolerable budget

Chosen budget
Dynamic Instrumentation

• Begin/stop throttling by instrumenting CUDA

No source code modification is needed
CPU Throttling and Scheduling

• Completely Fair Scheduler (CFS)
  – Linux’s default scheduler (for non-real-time tasks)
  – Virtual runtime: weighted execution time
  – Pick the task with smallest virtual runtime

• *Destructive* interplay of throttling and CFS
  – More throttling $\Rightarrow$ less virtual runtime increase
  – CFS prefers throttled tasks $\Rightarrow$ more throttling
Example Schedule under CFS

- CFS preferred memory intensive task $\tau_{mem}$

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c} 
  t_{CPU} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
  \tau_{RT} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \tau_{CPU} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \tau_{MEM} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \text{Throttle} & 0 & 0.33 & 0.67 & 1 & 2 & 2 & 2 & 3 & 4 & 4 & 4 \\
\end{array} \]

Throttled duration: 2 ms
Throttle Fair Scheduler (TFS)

• Account throttled time in virtual runtime

\[ V_{i}^{new} = V_{i}^{old} + \delta_{i}^{j} \times \rho \]

- Task’s virtual runtime
- Throttled duration
- Scale factor

• Effect
  - prefer more CPU intensive tasks
  - less CPU throttling
  - improved CPU throughput
Example Schedule under TFS

- TFS preferred CPU intensive task $\tau_{cpu}$

<table>
<thead>
<tr>
<th>$\nu_{CPU\ runtime}$</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\nu_{MEM\ runtime\ (TFS-3X)}$</td>
<td>0</td>
<td>2.34</td>
<td>2.34</td>
<td>2.34</td>
<td>3.34</td>
<td>3.34</td>
<td>4.34</td>
<td>5.34</td>
<td>6</td>
</tr>
<tr>
<td>$\nu_{MEM\ runtime\ (Actual)}$</td>
<td>0</td>
<td>0.33</td>
<td>0.33</td>
<td>0.33</td>
<td>1.33</td>
<td>1.33</td>
<td>2.33</td>
<td>3.33</td>
<td>4</td>
</tr>
</tbody>
</table>
Schedulability Analysis

- Classical RTA for preemptive fixed priority scheduling with blocking

\[ R_i^{n+1} = E_i + B_i + \sum_{\forall j \in hP(i)} \left[ \frac{R_j^n}{P_j} \right] E_j \]

- Treat GPU kernel execution as critical section
- Use priority ceiling protocol (PCP)

Outline

• Motivation

• BWLOCK++
  – Memory bandwidth throttling
  – Binary instrumentation
  – Throttle fair scheduler (TFS)
  – Schedulability analysis

• Evaluation

• Conclusion
Setup

• Hardware
  – Nvidia Jetson TX2
    • 4x Cortex-A57 (used) + 2x Denver (not used)
  – RT core: Core 0

• Software
  – Linux kernel 4.4.38 (+ TFS, BW regulator, …)
  – CUDA 8.0 + custom library (LD_PRELOAD)

• Benchmarks
  – Parboil benchmark suite (GPU tasks)
  – IsolBench benchmark suite (CPU tasks)
Real-Time Performance Impact

- Real-time GPU kernel performance is improved
• TFS improves CPU throughput (reduce throttling)
Conclusion

• Integrated CPU-GPU SoC platforms
  – Good: performance, cost, size, weight, power
  – Bad: memory bandwidth contention

• BWLOCK++
  – Automatically and efficiently protect real-time GPU kernels on integrated CPU-GPU SoC
  – Throttling + runtime instrumentation + scheduling
  – Practical solution

• Availability
  – https://github.com/wali-ku/BWLOCK-GPU
Thank You

Disclaimer:
This research is supported by the National Science Foundation (NSF) under the grant numbers CNS 1718880, CNS 1815959