THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

GPUSync: A Framework for Real-Time GPU Management

Glenn A. Elliott (gelliott@cs.unc.edu), Bryan C. Ward, and James H. Anderson

Traditional throughput-oriented GPGPU-based platforms are primarily designed to support a **single** GPGPU process at a time. This is problematic in deadline-oriented (real-time) systems when **multiple** processes compete for GPU resources. **System-level services** are necessary to schedule competing work according to **priority** to ensure that deadlines are met. **GPUSync** is a framework for implementing such schedulers in multi-GPU, multicore, real-time systems. GPUSync enables GPUs to be shared among processes in **safety-oriented applications**, such as advanced driver assistance systems (ADAS) and autonomous vehicles, since timing constraints can be guaranteed to be met.

Real-Time Versus Real-Fast

The term "real-time" is commonly interpreted as meaning "real-fast," but in the field of real-time systems, there are concerns for **guaranteeing precise timing constraints** and predictable system behavior. Real-time systems properly prioritize work to improve responsiveness, reduce jitter, minimize required hardware computing capacity, and ensure computing resources are given to the most critical applications while avoiding starvation of others. In contrast, "real-fast" systems are commonly designed for high throughput and are optimized for **average-case performance**, without guarantees of worst-case behavior.

GPUSync

Safety-critical applications often require computations to complete before a given physical-time deadline. Real-time scheduling algorithms are necessary to ensure that such computations meet their deadlines when system resources (such as CPUs) are **shared** by computations of varied priority. Such systems include advanced driver assistance systems and autonomous vehicles. Here, a vehicle may be outfitted with several types of input sensors (video, Lidar, etc.). Sensor data must be processes, synthesized, and acted upon within a given time-window to guarantee **vehicle reaction times**. It has been demonstrated that GPUs are well suited to handle computations in many stages of this processing [1, 2]. However, safety and certification requirements necessitate that real-time GPU management techniques are employed when various real-time GPGPU applications share GPU resources.

We present GPUSync, a framework for managing GPUs in multi-GPU, multicore, real-time systems, which...

- Can be used in conjunction with a variety of real-time CPU schedulers
- Can allocate CPUs & GPUs on a partitioned, clustered, or global basis
- Provide flexible mechanisms for allocating GPUs to computations
- Enables deterministic direct (P2P) state migrations between GPUs
- Provides migration cost predictors that determine when migrations can be effective
- Enables a single GPU's execution and DMA copy engines to be accessed in parallel by different real-time computations
- Properly integrates GPU-related interrupt and worker threads into real-time scheduling, even when GPU drivers are closed-source
- Provides budgeting policies to facilitate program isolation

A detailed description and evaluation of GPUSync may be found in [3].

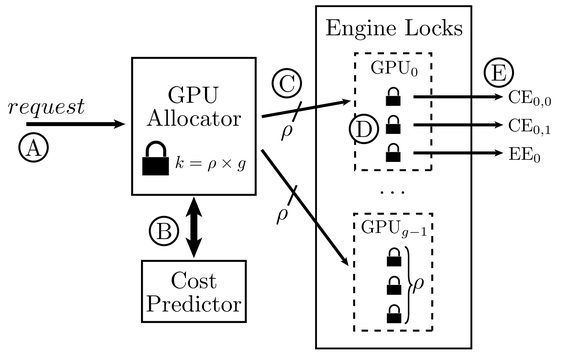
[1] F. Homm et al., "Efficient Occupancy Grid Computation on the GPU with Lidar and Radar for Road Boundary Detection." IVS, 2010.
[2] M. McNaughton et al., "Motion Planning for Autonomous Driving with a Conformal Spatio-temporal Lattice." ICRA, 2011.
[3] G. Elliott et al., "GPUSync: A Framework for Real-Time GPU Management," RTSS, 2013.
[4] www.litmus-rt.org
[5] J. S. Kim et al., "Realtime Affine-photometric KLT Feature Tracker on GPU in CUDA Framework." ICCV Workshop, 2009.

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

High-Level Design of GPUSync

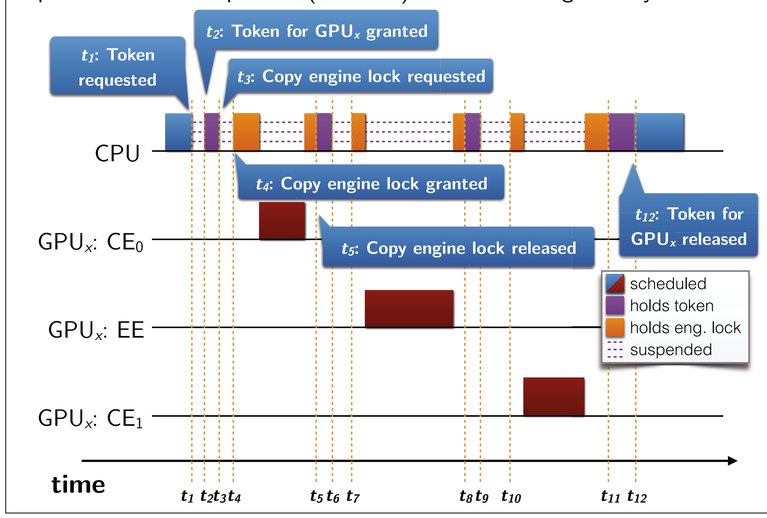
The high-level design of GPUSync is shown below. Each GPU is associated with ρ **tokens**. Each GPU execution and DMA copy engine is protected by a unique **lock**. Tasks (CPU-side threads) suspend if a token or lock is unavailable when requested; real-time **priority-inheritance** ensures no task suspends for an unbounded time duration. A task goes through the following sequence of operations to acquire GPU resources:

Step A: The task issues a request for a GPU from the GPU Allocator.
Step B: The GPU Allocator **predicts** which GPU will allow the requesting task to complete the earliest, given other GPU task assignments and the cost of migrating task state across GPUs.
Step C: The task is assigned a token (and awoken, if necessary).
Step D: The task requests the lock protecting the engine it requires.
Step E: The task utilizes the GPU's execution engine or copy engine to execute a GPGPU kernel or DMA memory copy. The task releases the engine lock when the operation completes. The task frees its GPU token once it no longer requires a particular GPU.



The Cost Predictor passively monitors each task's GPU assignments and GPU access patterns to estimate the cost of migrating task state between GPUs. Estimations are made by according to moving averages with outlier filtering. These estimations are **sensitive to the PCIe topology** of the system.

A possible execution pattern (schedule) for a task using GPUSync:



Implementation

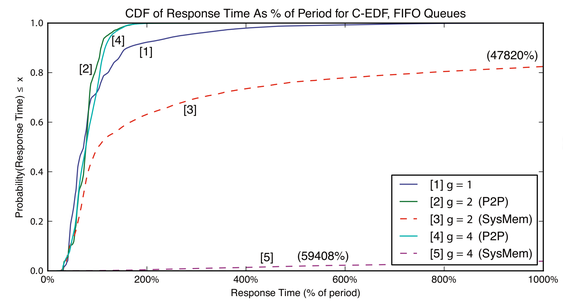
GPUSync is implemented as an extension to LITMUS^{RT}, a real-time patch to the Linux kernel (v3.10.5) [4]. GPUSync is comprised of **~20k lines of code**. Contributions to this total by category are: GPU Allocator and locking protocols, 35%; scheduler enhancements, budget enforcement, nested priority-inheritance, 35%; interrupt and thread management, 20%; and misc., 10%.

Evaluation

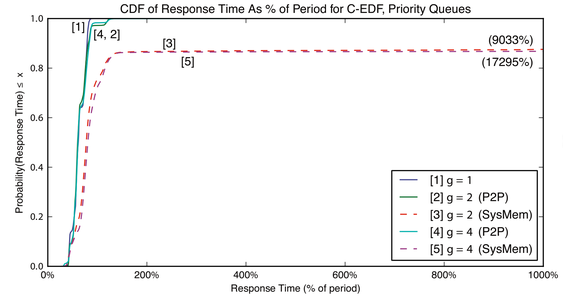
Test platform was a two-socket six-core Xeon X5060 system with eight NVIDIA Quadro K5000 GPUs. We performed KLT feature tracking on **30 simultaneous pre-recorded video streams** that ran at various frame rates, using code from [5], modified for GPUSync. This workload was scheduled under clustered earliest-deadline-first scheduling (C-EDF). GPU cluster sizes (g) were 1 (partitioned), 2 (small clusters) or 4 (large clusters). Engine lock requests were satisfied in FIFO- or priority-order. Task state migration between GPUs was either via pinned system memory or direct peer-to-peer (P2P) DMA.

Results

We examine the cumulative distribution functions of observed frame response times, normalized by frame period, over the duration of two minutes. Frames with normalized response times ≤ 1.0 met their deadlines. "Higher" lines indicate shorter response times of more frames (higher is better).



(a)



(b)

From the above, we observe:

1. Clustered GPU scheduling ($g \neq 1$) was only feasible when **P2P migrations** were used. Observe lines 3 and 5 in both figures above.
2. More frames completed before their deadline under **priority-ordered engine locks**. Observe that lines 1, 2, and 4 in (b) are higher than the corresponding lines in (a).
3. **Clustered GPU scheduling is advantageous** under FIFO engine locks. Observe this in (a), where lines 2 and 4 are higher than line 1.