GPU Management

1. Motivation
    a. Programmable accelerators becoming common
    b. Leverage existing demand in graphics for massive computation for non-graphics tasks
        i. Floating-point intensive, data parallel tasks
    c. Examples:
        i. Xeon Phi
        ii. GPU
            1. Many cores
            2. Each core has many threads – warps/wave fronts, like hyperthreads
            3. BUT: Each thread has lanes that execute the same instruction at the same time on different data. For different instructions, pause lanes that don't
            4. Languages:
                a. NVidia: CUDIA
                b. Intel/AMD: OpenCL
        iii. Programmable network interface card (smart NIC)
        iv. Programmable storage device (e.g. smart SSD, smart disk)
    d. Issues:
        i. How do you abstract devices to programmers?
            1. Network: sockets
            2. Disk: file system
            3. Keybord/display: tty
            4. GPU: no abstraction, just a device
        ii. Who should control these devices?
            1. Default: vendor device driver
            2. OS writer: want OS in control
        iii. What is needed from OS perspective?
            1. Scheduling mechanisms for policy goals
            2. Programming abstractions that compose with OS
                a. E.g. communication, synchronization
2. Proposed solutions
    a. Barrelfish Multikernel / Helios satellite kernels
        i. Run OS kernel on every device
        ii. OS does local scheduling, local functionality
        iii. Can have different architecture, as communicate via messages and RPC

    iv. Can change communication mechanism based on whether it is a CPU or a programmable device
        1. Shared memory
        2. DMA / I/O memory
  b. GPUnet, GPUfs:
    i. Write implementations of OS functionality to be called from accelerator (GPU), but not provide scheduling control
        1.
3. Pegasus
  a. Goals:
    i. Allow virtualized access to GPUs
    ii. Make GPUs first-class entity
        1. Allocate/schedule work on them by OS, rather than leaving it all up to the driver
    iii. QUESTION: WHY?
        1. Who do OS people want to control/schedule all hardware?
        2. ANSWER: allows sharing between applicatons
        3. ANSWER: allows efficiency; can overlap use of a device with other things
    iv. Coordinated scheduling of CPU and GPU
        1. May need to run at same time to pass work to GPU, use results
        2. Example: using GPU for gesture recognition with a camera
            a. Want real-time response
            b. Need both CPU and GPU
                i. Detect movement in GPU
                ii. Convert to mouse movements on CPU
  b. Assumptions:
    i. Static toolchain decides what to run on GPU (no dynamic decision making on CPU vs GPU)
    ii.
  c. Architecture:
    i. Use virtualization to share GPU
        1. Apps talk to virtual GPU that is scheduled by pegasus, rather than a real GPU
        2. Scheduling within a domain is not a Pegasus problem
    ii. All programmable entities are schedulable:
        1. VCPUs and aVCPUs can be scheduled independently
        2. May want coordination if need to run code on both at the same time
    iii. GViM GPU virtualization
        1. Key idea: virtualize at CUDA api interface

            a. Ship CUDA calls to backend driver
         2. Run GPU driver & runtime in Dom0 (management / device driver domain)
         3. Provide CUDA API (user-mode GPU API for compute) in guest client process
         4. Add GPU front end to guest, GPU backend to Dom0 for communication
         5. Use shared memory for data movement to avoid copying
            a. Guest allocates GPU data in memory shared with Dom0, or ideally, with GPU itself
         6. Ring buffer of requests for CUDA commands
            a. Pass data to backend over shared pages, ring buffer per VM for requests
               i. Like other drivers
            b. In backend: polling thread pulls requests off ring buffer and calls CUDA runtime and sends responses
            c. SO: burn a virtual CPU for communicating from frontend to backend GPU

         7. Management service in Dom0 handles scheduling of GPU
            a. Round robin: equal timeslice monitoring of different DomUs
            b. xenCredit: timeslice proportional to credit monitoring of DomUs; more credits means longer monitoring of queue

4. Accelerator Virtual CPU:
   a. Abstract representation/virtualization of running code on an accelerator (GPU)
      i. Contains CPU/GPU state needed to run on accelerator (e.g. shared data, queues, context information)
   b. Can be scheduled by management code in Dom0
5. Resource Management
   a. Phase 1: domain selection
      i. Decide which domains can use the GPU (exclusively)
         1. Place these domains in ready queue
   b. Phase 2: running requests
      i. When a domain issues request over ring buffer, runs and its requests are forwarded to GPU
      ii. Goal: restrict # of domains using GPU at a time due to limited memory available
   c. Deciding which GPU to use

        i. Have profile of GPU properties (memory, speed, bandwidth, etc.) + dynamic information (memory available)
        ii. Order GPUs in priority order of best to use (most available capacity) to worst to use (least capacity left)
    d. Doamin profile:
        i. How aggressively does it use GPU?
        ii. How much GPU memory does it need?
        iii. How much share has it been given of the PU?
        iv. Created manually for now.
    e. DomA scheduler:
        i. Pick which domains to assign to which GPUs when
        ii. Coordinates with Hypervisor scheduler for better behavior
6. Scheduling GPUs
    a. What is the right granularity?
        i. Per call: too fine grained, too small + too much switching overehad
        ii. Per app (1 app at a time): too coarse grained, too inefficient and too high latency
        iii. Really: want something in the middle that is fine grained for responsiveness but coarse grained for efficiency
    b. Possible policies:
        i. Hypervisor-independent (not consider CPU scheduling)
            1. FCFS (default GPU policy)
                a. Bad isolation, sharing as described before
            2. Accelerator Credit – proportional share
                a. Same as XenCredit but have separate credits for accelerator
        ii. Hypervisor-controlled policies: HV says who can use GPU
            1. CoScheduling: only allow a domain access to GPU when its domain is running on a VCPU
                a. Good for latency-sensitive code; VCPU is running to submit requests & receive results and use immediately
        iii. Hypervisor coordinated policies
            1. Problem: if scheduled domain does not use GPU, GPU is idle
                a. Domain may not have GPU credit left when it has CPU credit
            2. Augmented credit:
                a. Hypervisor tells DomA scheduler what upcoming schedule is & credits for each domain

        b. DomA scheduler adds GPU credits to domains that the CPU will be running soon
            i. Increases chances of the domain using a GPU soon, but does not guarantee it (not strict co-scheduling)
            ii. Effectively: get a priority boost when VCPU of a domain runs
    3. SLA feedback for QoS
        a. How handle real-time apps that need to complete task?
        b. Solution:
            i. Assign SLO (objective) for each app: how much time it should be getting on GPU per period
            ii. Periodically poll domains with SLOs to see if they are getting enough time
            iii. If not, give more credits to those domains
        c. Results: automatically adjust credit assignment to accommodate fluctuations in actual utilization

7. Implementation:
    a. GPU scheduling:
        i. Simple policies:
            1. Timer interrupt to DomA triggers scheduler to switch domains
            2. One timer interrupt per GPU (like CPU) to decide when to switch it
        ii. Complex policies:
            1. thread per GPU to be scheduler
            2. Thread per domain to poll for requests
        iii. Coordination with Hypervisor:
            1. Share VCPU->PCPU schedule with DomA (shared mem?)
            2. Quantum drift between CPU and GPU for co-scheduling/coordination
                a. Want to have same start/end
                b. Requests to GPU are in a queue; may not start running when domains VCPU starts running
                c. Current solution: run aVCPU for a bit longer (before/afterwards) to increase chance of overlap
                d.