Disengaged Scheduling For Fair, Protected Access to Fast Computational Accelerators

Amir Taherin

Prof. Dwarkadas

CSC 456: Operating Systems Course

Fast Computational Accelerators

Introduction

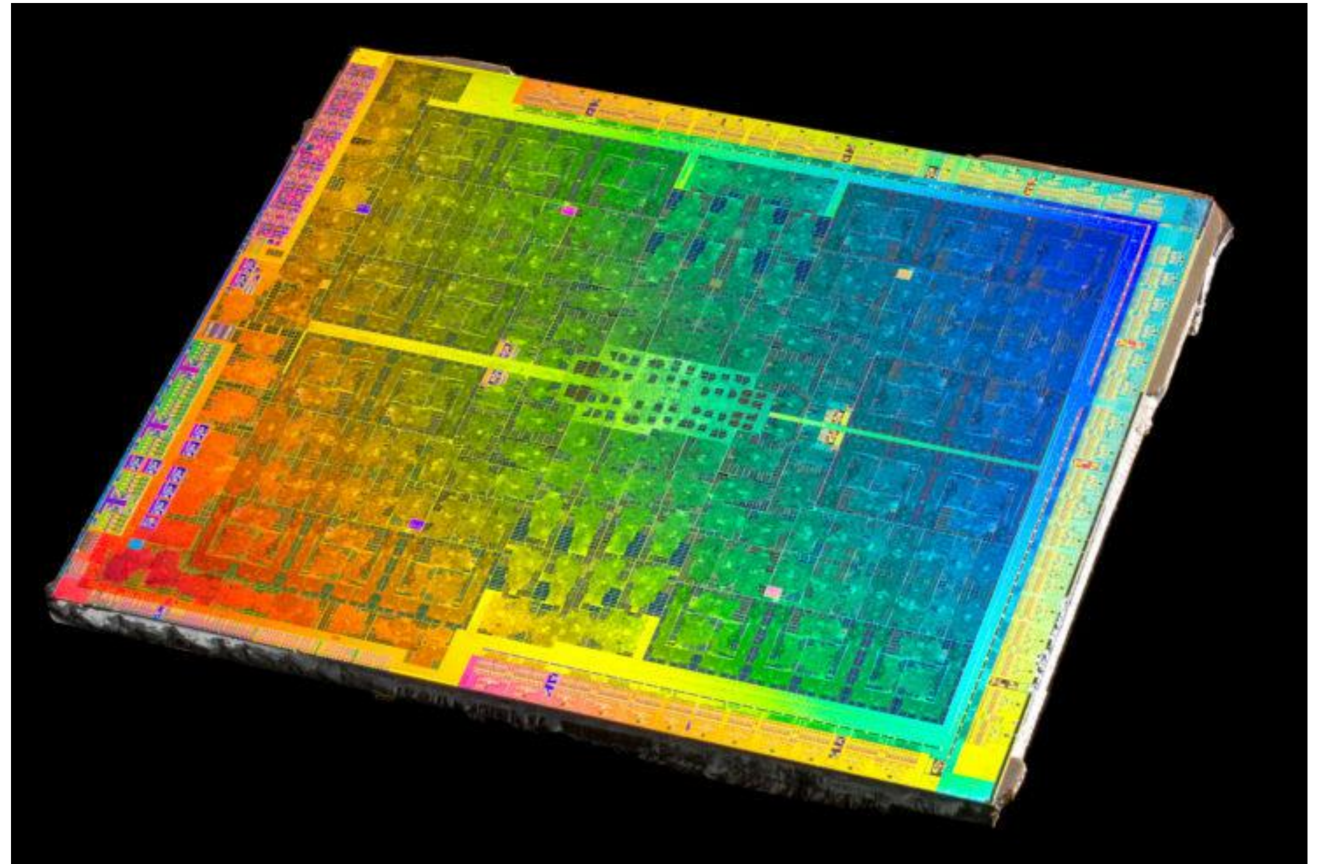
Example: GPU

Nvidia GTX 1070

First: 1970 in arcade games

In the home market, the Atari 2600 in 1977 used a video shifter called the Television Interface Adaptor

Nvidia GeForce 3 : 2001



https://wccftech.com/nvidia-gtx-1080-gp104-die-shot/

https://Wikipedia.com

Today

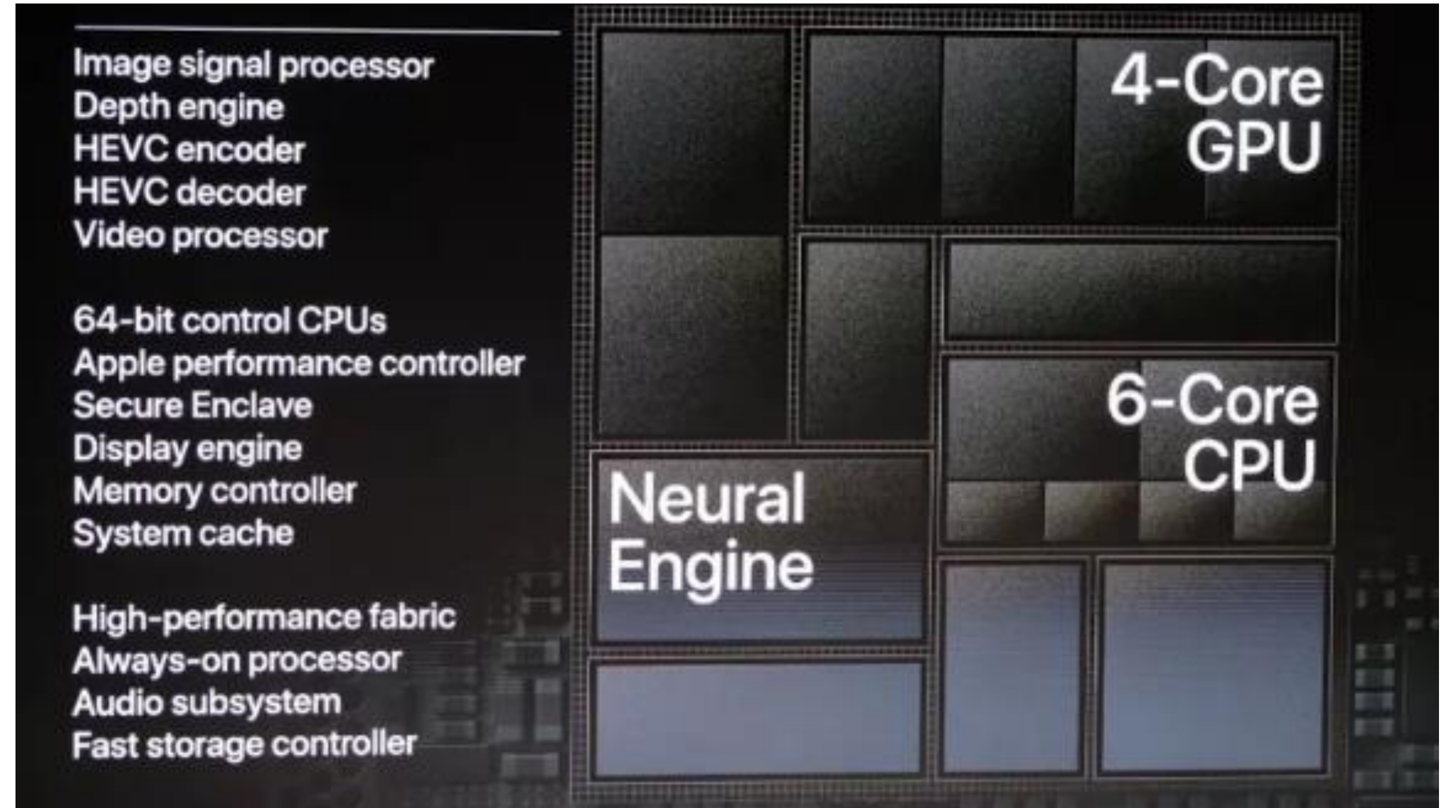
Mobile SoCs

Iphone Xs Max

A12 Bionic chip

Apple-designed 64-bit ARMv8.3-A 6-Core CPU

Apple-designed 4-Core GPU



Shortcoming

- Operating System has <u>no control</u> over hardware accelerators.

- Why?
  - HIGH Overhead
  - e.g., Direct GPU accesses from user
    - Management in OS requires Interrupts
    - Interrupts are too costly
- GPUs are **<u>Black-Boxes</u>** for the OS
  - No sufficient information from vendors

GPUs: Black Boxes [USENIX 2013]

- Today's typical GPU architecture
  - Device
  - Driver
  - User-level libraries
- Open Documents:
  - High level programing model
  - Library interface
  - Architectural characteristics
    - Performance tuning and high-level programing

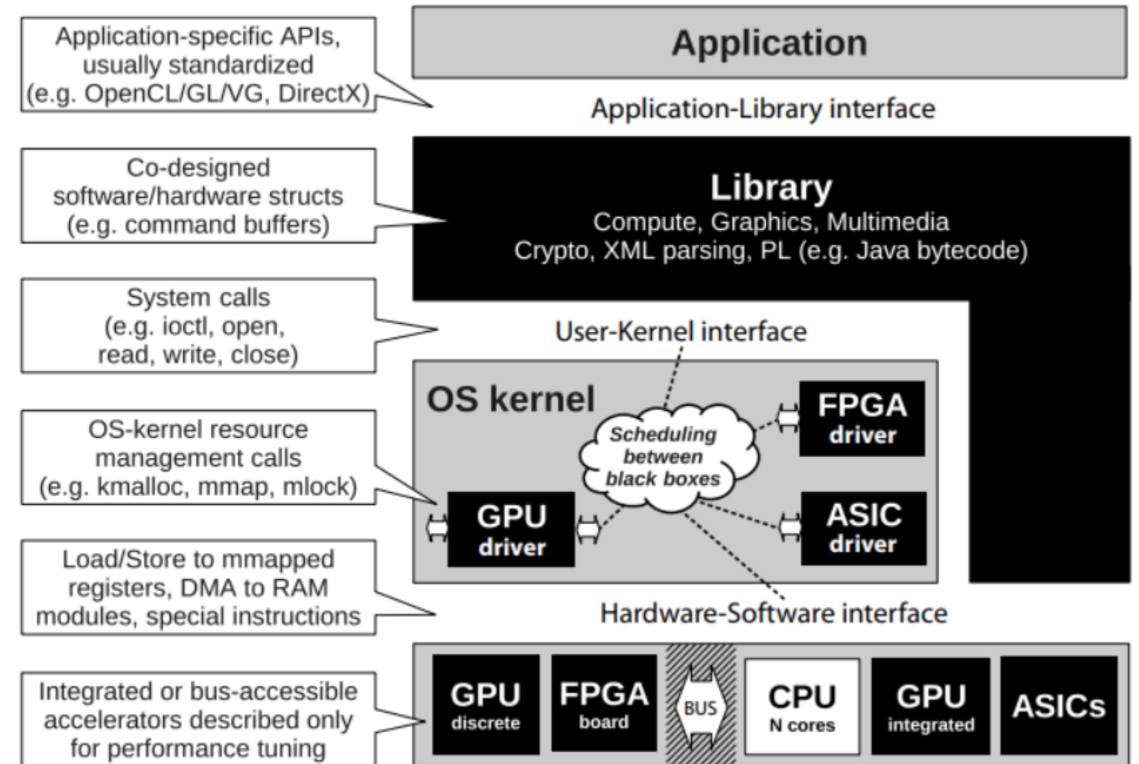


Figure 1: The GPU software / hardware architecture, with notes on interfaces and components. Gray areas indicate open system / application components while black areas indicate black-box components without published specifications or behaviors.

How are GPUs used?

- For minimal overhead on very low latency GPU requests
  - User-level library
  - Frequently communicates directly with the device (in both directions)
  - Memory-mapped buffers and registers are used
  - Bypassing the OS kernel
- Malicious application:
  - Obtain an unfair share of GPU resources (cycles, memory, and bandwidth)
  - Kernel has no way to coordinate GPU activity with other aspects of resource management in pursuit of system-wide objectives.

We Need More Information on GPUs

- Protected OS-level resource management
  - A pressing need
- To satisfy this need:
  - The kernel must be able to:
    - Identify and delay GPU request submissions
    - Tell when requests complete
  - Need for clean interface to expose this information
    - Hopefully be provided by vendors in the near future

Open The *Black-Box*

Previous Work [USENIX 2013]

“Model the black-box stack as a state machine that captures only as much as we need to know to manage interactions with the rest of the system.”

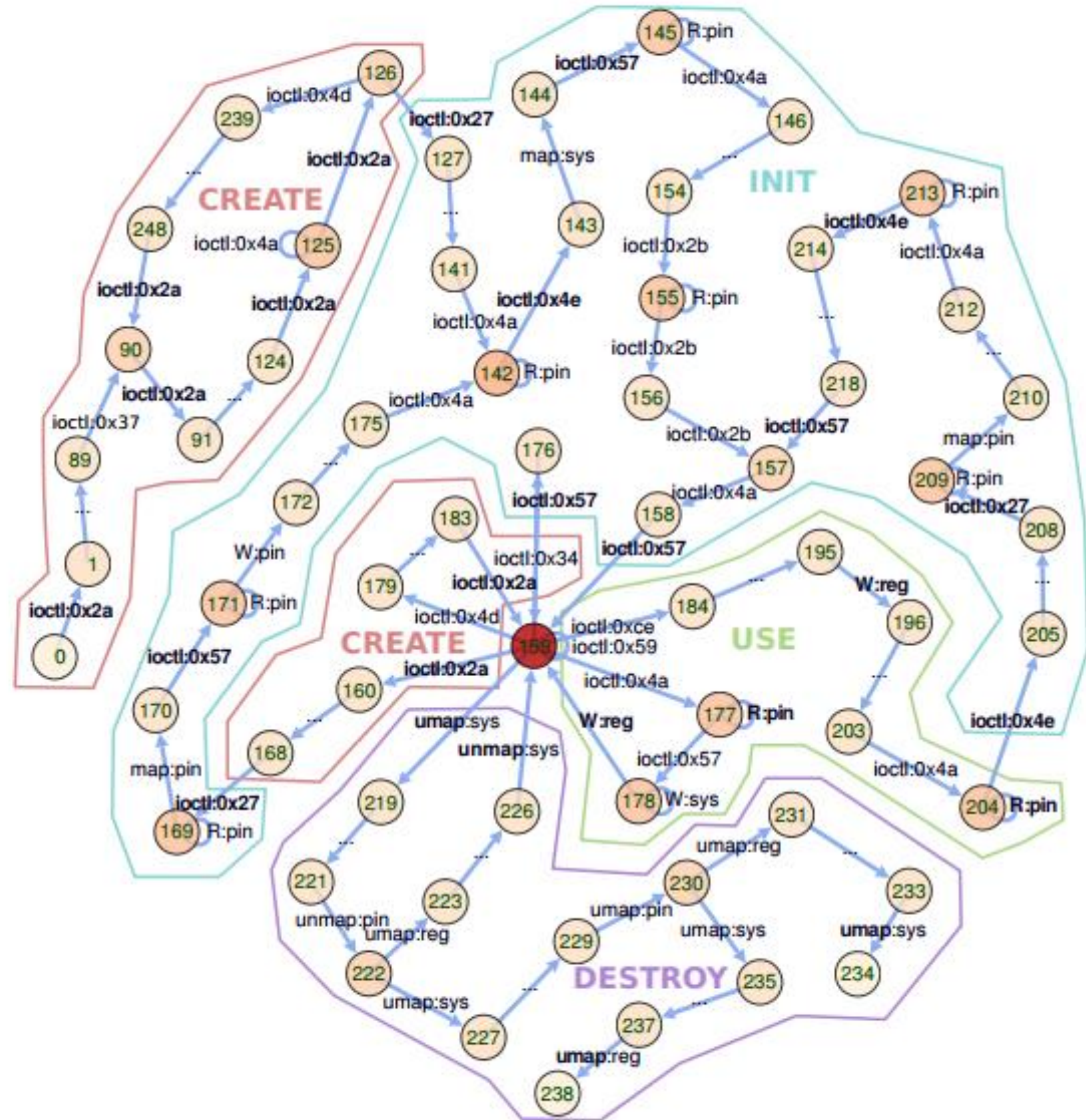
High-Level Solution

Language Words

<table>
<tr><th>Event type</th><th>Meaning</th></tr>
<tr><td>ioctl:0x??</td><td>ioctl request : unique hex id</td></tr>
<tr><td>map:[pin|reg|fb|sys]</td><td>mmap : address space</td></tr>
<tr><td>R:[pin|reg|fb|sys]</td><td>read : address space</td></tr>
<tr><td>W:[pin|reg|fb|sys]</td><td>write : address space</td></tr>
<tr><td>pin</td><td>locked (pinned) pages</td></tr>
<tr><td>reg</td><td>GPU register area</td></tr>
<tr><td>fb</td><td>GPU frame buffer</td></tr>
<tr><td>sys</td><td>kernel (system) pages</td></tr>
</table>

Table 1: Event types and (for map, R, and W) associated address spaces constitute the alphabet of the regular language / GPU state machine we are trying to infer.

State Machine Case Study



Lesson Learned

- State Machine:
  - Request handling by the system
  - Interface for more direct involvement of the operating system
- NOTE: We can intercept and intercede on request making and request-completion events
  - Make a scheduling decision that reflects its own priorities and policy.

Scheduling for Fair, Protected Access to Fast Computational Accelerators

Problem Statement

ASPLOS 2014

Prerequisites

- Future Microprocessors: Highly Heterogeneous
  - **Hardware accelerators**
- Current OS: No control over accelerators
  - Application with larger requests: receives more time
  - Greedy application hog the GPU
  - Malicious application may launch a denial-of-service (DoS) attack

Modern Accelerator Systems

- Two major challenges:
- First
  - Low Latency
    - Bypassing the OS
      - No resource management
- Second
  - NO information on accelerators
    - Hardware interface for accelerators
      - Hidden from programmers

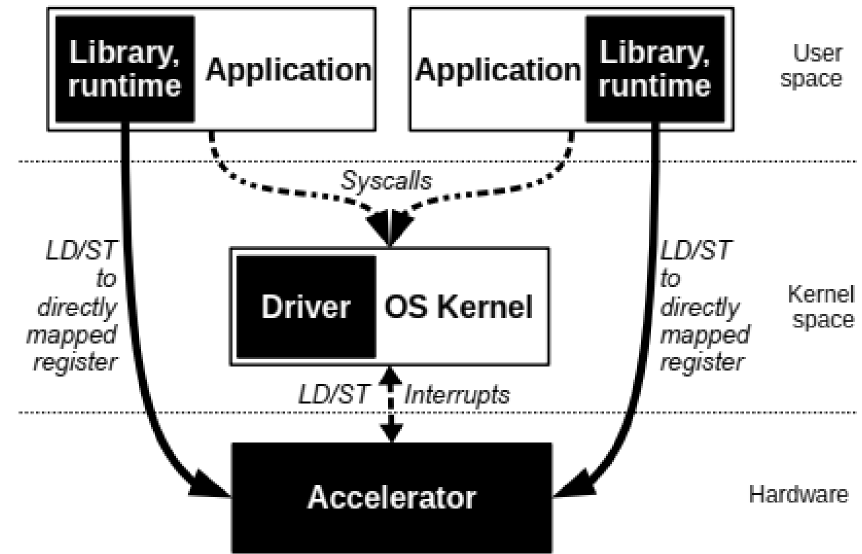
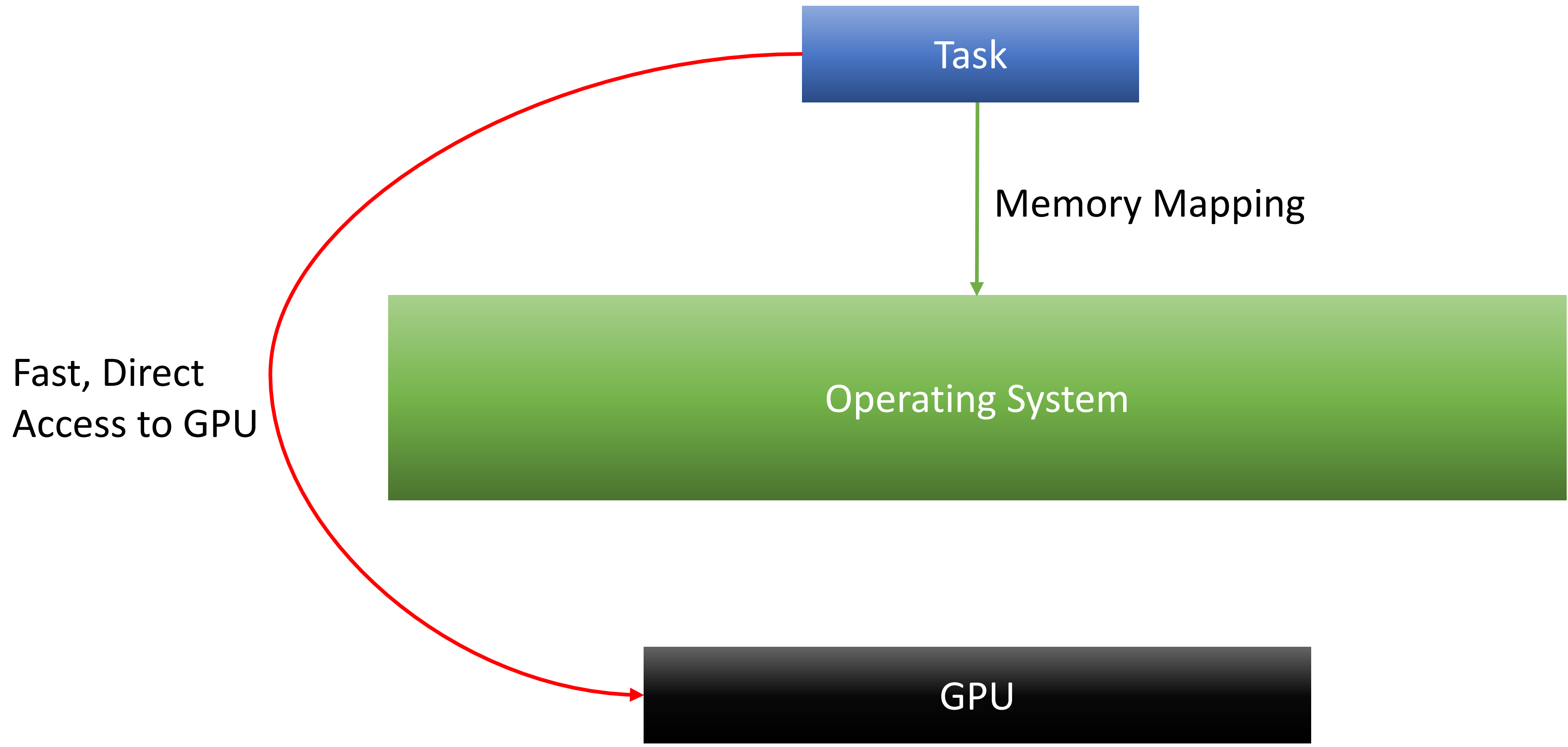
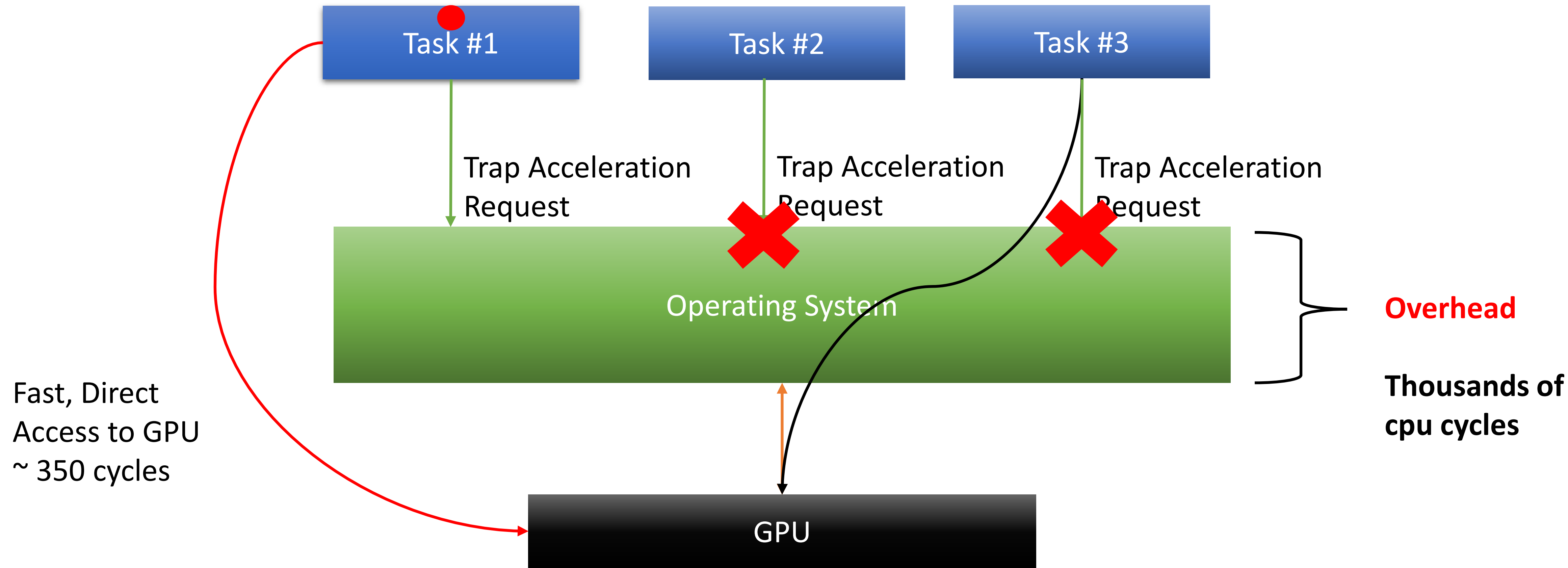


Figure 1. For efficiency, accelerators (like Nvidia GPUs) receive requests directly from user space, through a memory-mapped interface. The syscall-based OS path is only used for occasional maintenance such as the initialization setup. Commonly, much of the involved software (libraries, drivers) and hardware is unpublished (black boxes).



How are GPUs Used?

Timeslice with Overuse Control



Timeslice Scheduler

Timeslice with Overuse Control

- Fairness:
  - Upon the completion of requests that overrun the end of a timeslice
  - Deduct their excess execution time from future timeslices of the submitting task
  - Charge the token holder
    - Sometimes: skip the task's next turn(s)

Timeslice with Overuse Control: *prototype*

- Protection against over-long requests
  - Killing the offending task
  - Leave the accelerator in a clean state
    - Let the existing accelerator stack follow its normal exit protocol
  - Returning occupied resources back to the available pool
  - Cleanup depends on (**undocumented**) accelerator features
    - It appears to work correctly on modern GPUs

Timeslice with Overuse Control: *Limitations*

- Two efficiency drawbacks
  1. Interrupt-based requests
     - Significant overhead on fast accelerators
  2. Not *work-conserving*:
     - The accelerator may be idled during the timeslice of a task that temporarily has no requests
     - e.g., Other tasks are waiting for accelerator access

Disengagement

Solution

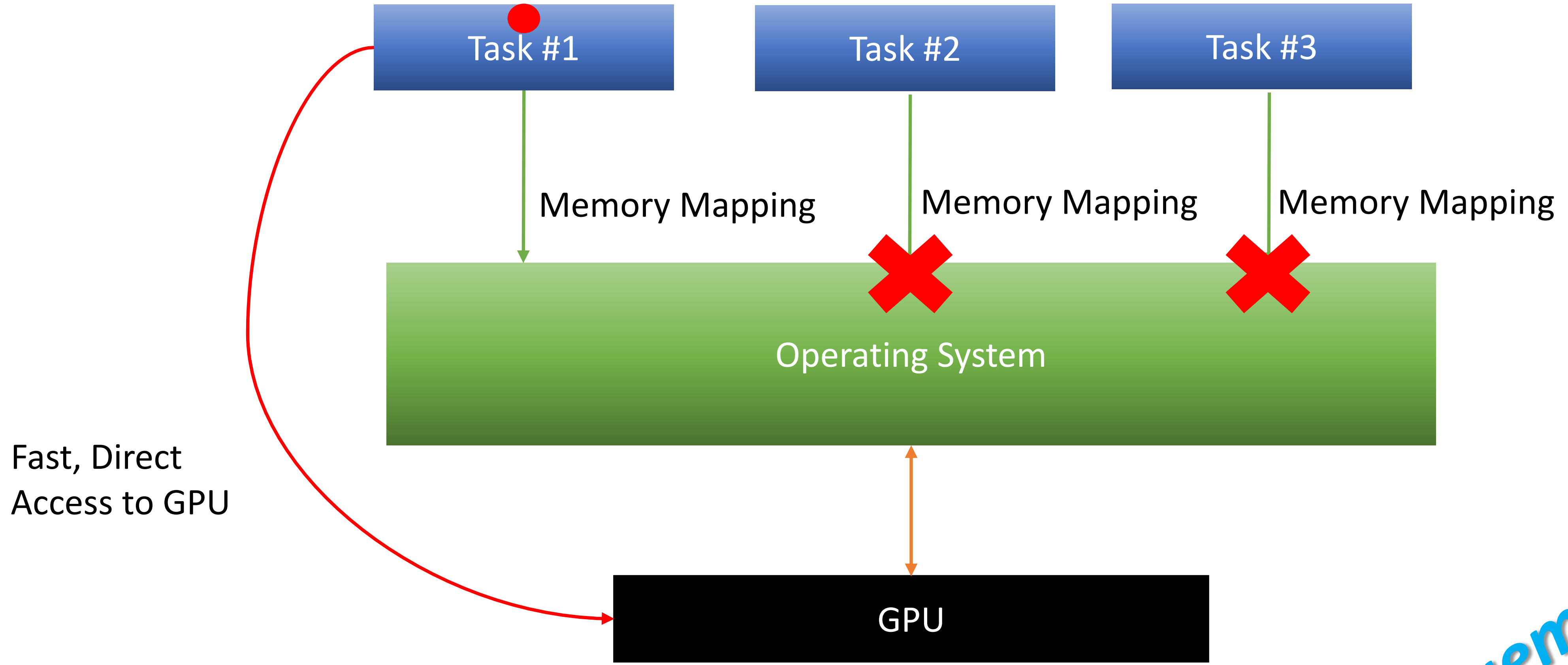
Disengaged Timeslice scheduler: tackles the overhead problem

Disengaged Fair Queueing scheduler: tackles the problem of resource idleness

High-Level Overview

- key idea, called **Disengagement**
  - OS resource scheduler
  - Maintain fairness
    - Only: Interceding on a small number of acceleration requests
    - How? => Disabling the direct mapped interface and intercepting resulting faults
  - **Majority of requests**
    - Unhindered direct access

Disengaged Timeslice Scheduler



Disengagement

Disengaged Timeslice Scheduler

Disengaged Timeslice Scheduler

- Difference:
  - Direct, unmonitored access
  - For token holder, In its Timeslice
- NOTE: Interception is not required for all requests
- When passing the token between tasks
  - OS updates page tables to enable or disable direct access
  - Largely ***disengaged***
  - Scheduler re-engages at the start of a new timeslice
  - Challenge: pending requests from the token holder of the last timeslice
  - To account for overuse: wait for such requests to finish

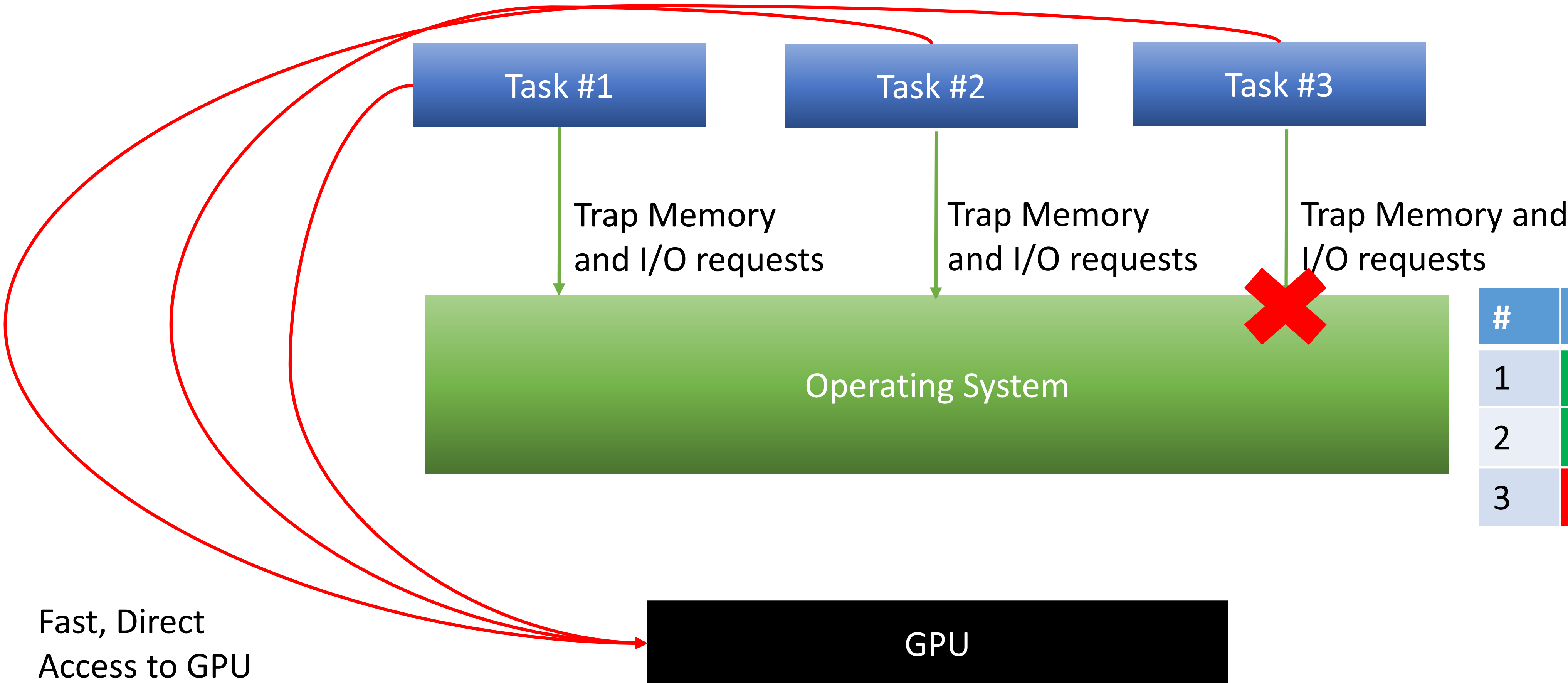
Disengaged Timeslice Scheduler: *prototype*

- Discover the semantics of data structures shared between the user and the GPU
  - Reference counter
    - Written by the hardware upon the completion of each request.
- Upon re-engaging
  - Traverse in-memory structures to find the reference counter of the last submitted request
  - Poll the reference counter for an indication of its completion

Disengaged Timeslice Scheduler: *Limitations*

- Disengaged Timeslice scheduler
  - Not suffering from high per-request management costs
- BUT
  - May lead to poor utilization
  - When the token holder is unable to keep the accelerator busy
  - This problem is addressed by **Disengaged Fair Queueing Scheduler**

Disengaged Fair Queueing Scheduler



#	Usage
1	OK
2	OK
3	Exceed

Disengagement

Disengaged Fair Queueing Scheduler

Disengaged Fair Queueing Scheduler

- Achieves fairness while maintaining work-conserving properties
  - i.e., it avoids idling the resource when there is pending work to do
- A standard fair queueing scheduler
  - Assigns a *start tag* and a *finish tag* to each resource request
- Tags serve as
  - Task's cumulative resource usage before and after the request's execution
  - **Start tag**
    - **MAX[**Current system virtual time (as of request submission), Finish tag (prev. request)**]**
  - **Finish tag**
    - The start tag plus the expected resource usage of the request

Disengaged Fair Queueing Scheduler (cont.,)

- Multiple requests (potentially from different tasks) -> Active Tasks
  - Dispatched to the device at the same time
  - GPUs have internal schedulers (e.g., Round Robin)
- A fair queueing scheduler on Inactive Tasks
  - Might build up its resource credit without bound
    - Sudden reclamation with burst
    - Causing prolonged unresponsiveness to other tasks
  - Solution: Reflect only the progress of active tasks
    - Since the start tag of each submitted request is brought forward to at least the system virtual time, any claim to resources from a task's idle period is forfeited

Disengaged Fair Queueing Scheduler: *Prototype*

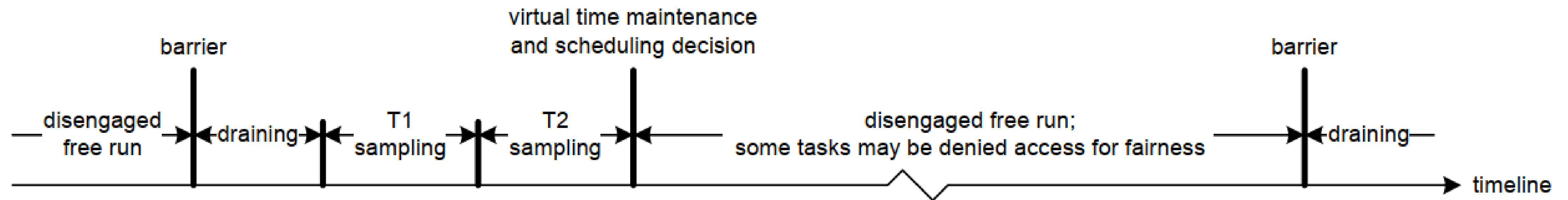


Figure 3. An illustration of periodic activities in Disengaged Fair Queueing. An engagement episode starts with a barrier and the draining of outstanding requests on the GPU, which is followed by the short sampling runs of active tasks (two tasks T_1 and T_2 in the case), virtual time maintenance and scheduling decision, and finally a longer disengaged free-run period.

Disengaged Fair Queueing Scheduler: *Limitations*

- In principle, better estimates might be based on more detailed reverse engineering of the GPU's internal scheduling algorithm
- Estimation can be imprecise, resulting in imperfect fairness
- Random vs Periodic behavior

Implementation

- Linux 3.4.7 kernel
- About 8000 lines of code
  - ioctl
  - mmap
  - munmap
  - copy_task
  - exit_task
  - do_page_fault
- Few hooks to the Nvidia driver's binary interface
  - initialization
  - ioctl
  - mmap requests

Implementation (cont.,)

1. Initialization phase
   - Identify: virtual memory areas
     - All memory-mapped device registers
     - Buffers associated with a given *channel*
       - Channel: a GPU request queue and its associated software infrastructure
2. Page-fault–handling mechanism
   - Catch: device register writes -> request submission
3. Polling-thread service
   - In kernel
     - Detect reference-counter updates
     - Identify completion of previously submitted requests

- 2 and 3: kernel internal interface for event-based scheduler

Results

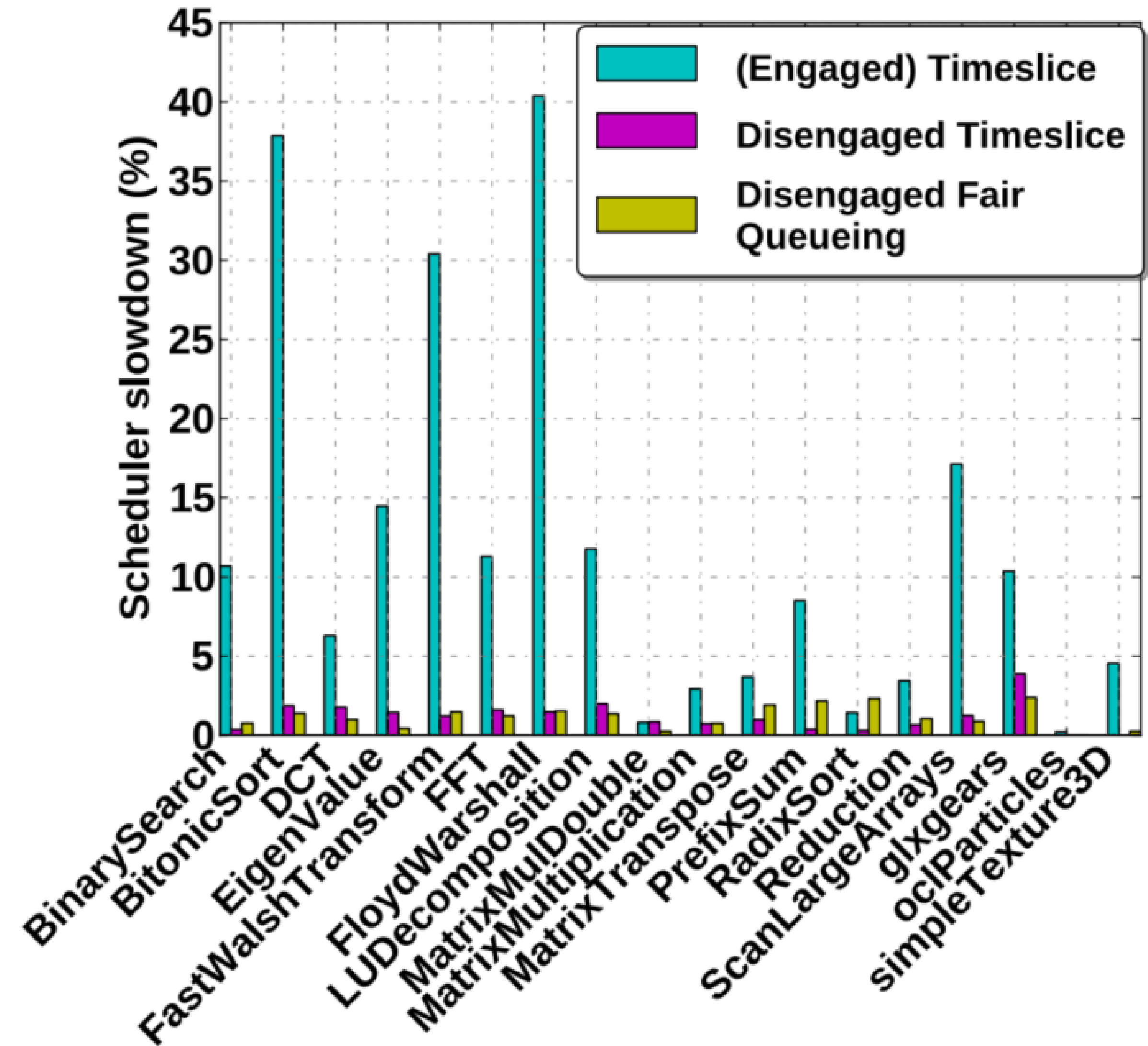


Figure 4. Standalone application execution slowdown under our scheduling policies compared to direct device access.

Results

Developed a “Throttle” microbenchmark

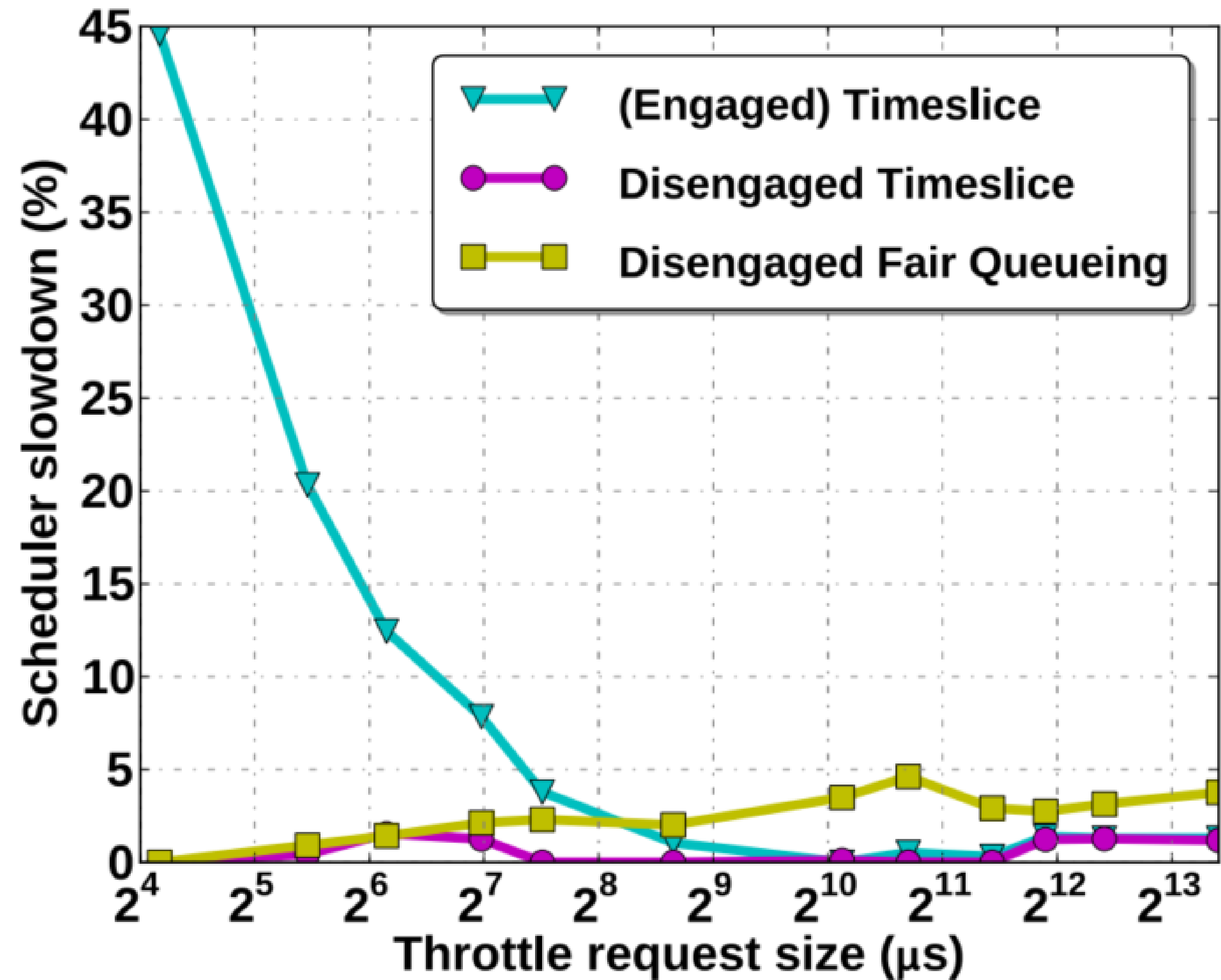


Figure 5. Standalone Throttle execution (at a range of request sizes) slowdown under our scheduling policies compared to direct device access.

Real-Time vs Non Real-Time

Area	Disengaged [ASPLOS 14]	Real-Time [RTSS 13]
Real-Time Properties	None	Soft Real-Time
Scheduler Structure	Token + Queue	Priority Queue + FIFO
Number of GPUs	One	Multiple GPU (migration between GPUs)
Target Application	Not Application Specific	Real-Time Applications in Self-Driving Cars
Approach	Scheduling Problem	Synchronization Problem
Goal	Reduce OS Management Overhead	Improve Average Case Performance and Maintaining the Soft Real-Time Constraints

?

1. Konstantinos Menychtas, Kai Shen, and Michael L. Scott. "Disengaged scheduling for fair, protected access to fast computational accelerators". In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (ASPLOS '14). ACM, New York, NY, USA, 301-316.
2. Menychtas, Konstantinos, Kai Shen, and Michael L. Scott. "Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack." *USENIX Annual Technical Conference*. 2013.

Thanks!

Amir

Protected Scheduling with Efficiency

- Kernel Perspective
  - Accelerators are Event-Based Interfaces
    - Start time
    - Complete time
- **Cost of OS management**
  - Trapping to the OS
    - Via syscalls or faults
      - Carries nontrivial costs
  - User/kernel mode switch => thousands of CPU cycles
  - Direct request submission:
    - 365 cycle on Nvidia GTX670

Overview of Schedulers

- *Timeslice* scheduler
  - Retains the overhead of per-request kernel intervention
  - Shares the GPU fairly
    - Unmodified applications
- *Disengaged Timeslice* scheduler
  - Extends Timeslice scheduler
  - Eliminates kernel intervention in most cases
- Timeslice Limitation: Only one application at a time uses the accelerator
- *Disengaged Fair Queueing* scheduler
  - Facilitates multiple application usage
  - BUT, Provides only statistical guarantees of fairness

Disengaged Fair Queueing Scheduler: *Design*

- Efficiency
  - Avoiding intercepting and manipulating most requests
- Track cumulative per-task resource usage and system-wide virtual time
- No per-request control
- **Start/finish tags**
  - A probabilistically-updated per-task virtual time
  - Closely approximates the task's cumulative resource usage
  - Time values are updated using statistics obtained through periodic engagement
- Control coarse grain:
  - Disable tasks that are running ahead in resource usage
    - Interval between consecutive engagements
  - Allow their peers to catch up
- NOTE: Requests from all other tasks are allowed to run freely in the disengaged time interval

Experimental Evaluation

Application	Area	μs per round	μs per request
BinarySearch	Searching	161	57
BitonicSort	Sorting	1292	202
DCT	Compression	197	66
EigenValue	Algebra	163	56
FastWalshTransform	Encryption	310	119
FFT	Signal Processing	268	48
FloydWarshall	Graph Analysis	5631	141
LUDecomposition	Algebra	1490	308
MatrixMulDouble	Algebra	12628	637
MatrixMultiplication	Algebra	3788	436
MatrixTranspose	Algebra	1153	284
PrefixSum	Data Processing	157	55
RadixSort	Sorting	8082	210
Reduction	Data Processing	1147	282
ScanLargeArrays	Data Processing	197	72
glxgears	Graphics	72	37
oclParticles	Physics/Graphics	2006	12/302
simpleTexture3D	Texturing/Graphics	2472	108/171

Table 1. Benchmarks used in our evaluation. A "round" is the performance unit of interest to the user: the run time of compute "kernel(s)" for OpenCL applications, or that of a frame rendering for graphics or combined applications. A round can require multiple GPU acceleration requests.