Real-Time Support for GPU

GPU Management

Heechul Yun

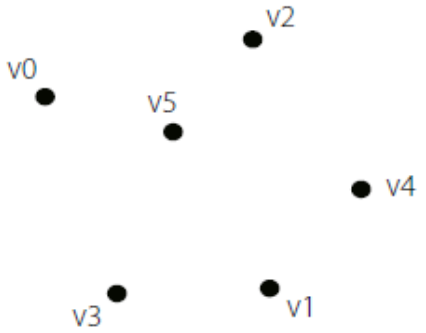
This Week

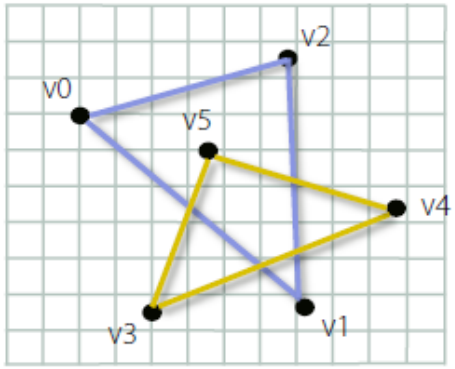
- Topic: Real-Time Support for General Purpose Graphic Processing Unit (GPGPU)

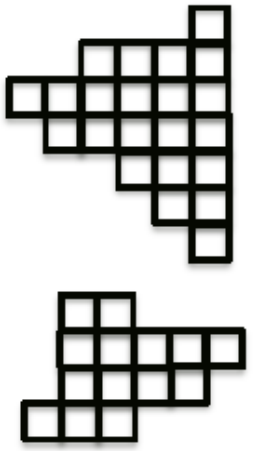
- Today
  - Background
  - Challenges
  - Real-Time GPU Management Frameworks

- No office hour today. I will have on Wed instead.

Pipeline entities





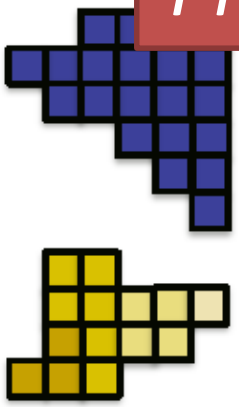


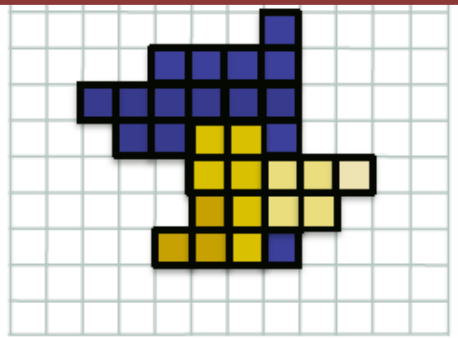
Vertices

Primitives

Fragments

Processed Independently





Fragments (shaded)

Pixels

Slide from Prof. Kayvon Fatahalian. CMU

History

- GPU
  - Graphic is **embarrassingly parallel** by nature
  - GeForce 6800 (2003): 53GFLOPs (MUL)
  - Some **PhDs** tried to use GPU to do some general purpose computing, but difficult to program

- GPGPU
  - **Ian Buck** (Stanford PhD, 2004) joined Nvidia and created CUDA language and runtime.
  - General purpose: (relatively) easy to program, many scientific applications

Nvidia Tesla K80 GPGPU

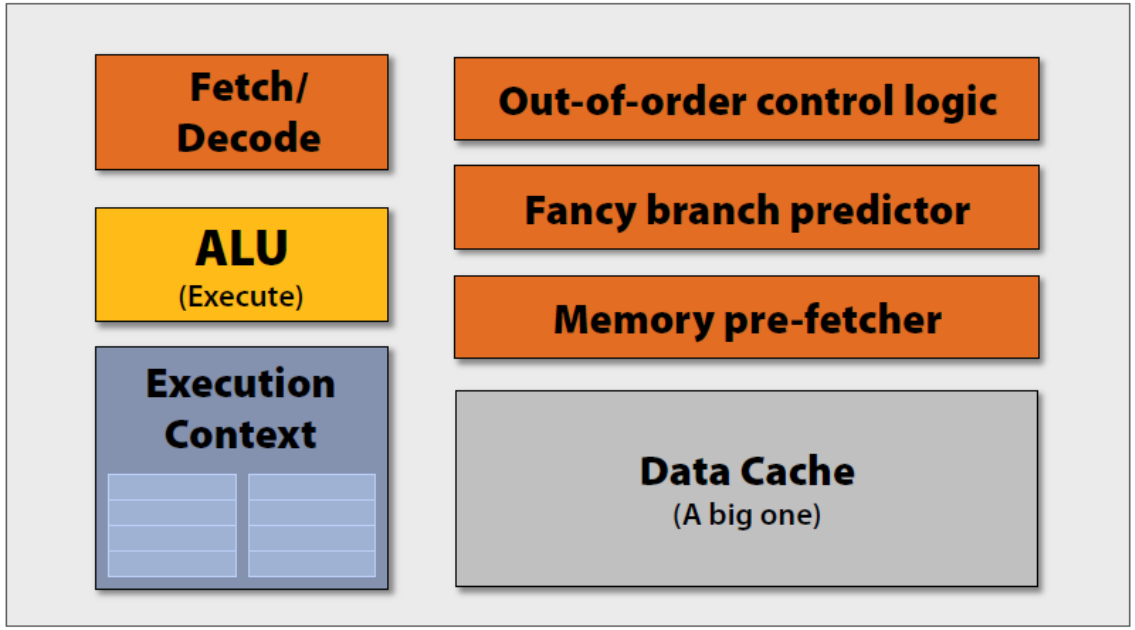
- CUDA cores: 4992
- Peak performance: **8.74 TFLOPs** (SP floating)

Image source: [Nvidia official website]

CPU vs. GPGPU

- CPU
  - Designed to run sequential programs faster
  - High ILP: pipeline, superscalar, out-of-order, multi-level cache hierarchy
  - Powerful, but complex and **big**
- GPGPU
  - Designed to compute math faster for embarrassingly **parallel data** (e.g., pixels)
  - No need for complex logics (no superscalar, out-of-order, cache)
  - Simple, less powerful, but **small**---can put **many** of them

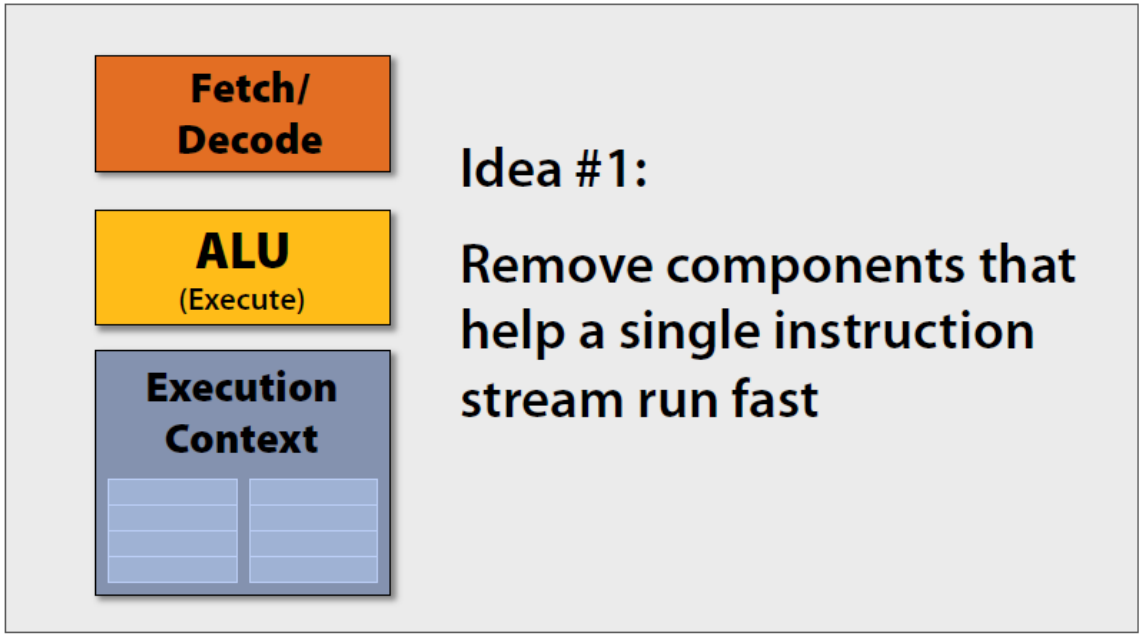
CPU-"style" cores



"From Shader Code to a Teraflop: How GPU Shader Cores Work", Kayvon Fatahalian, Stanford University

Slimming down

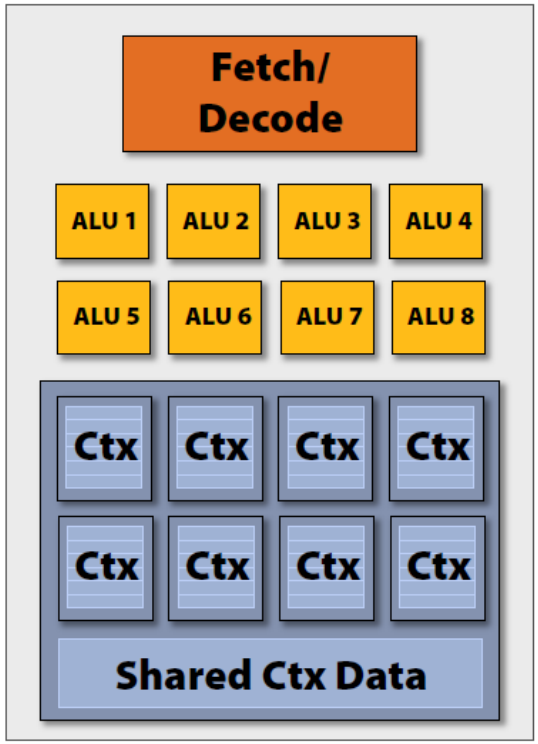
SIGGRAPH2008



"From Shader Code to a Teraflop: How GPU Shader Cores Work", Kayvon Fatahalian, Stanford University

Add ALUs

SIGGRAPH2008



Idea #2:

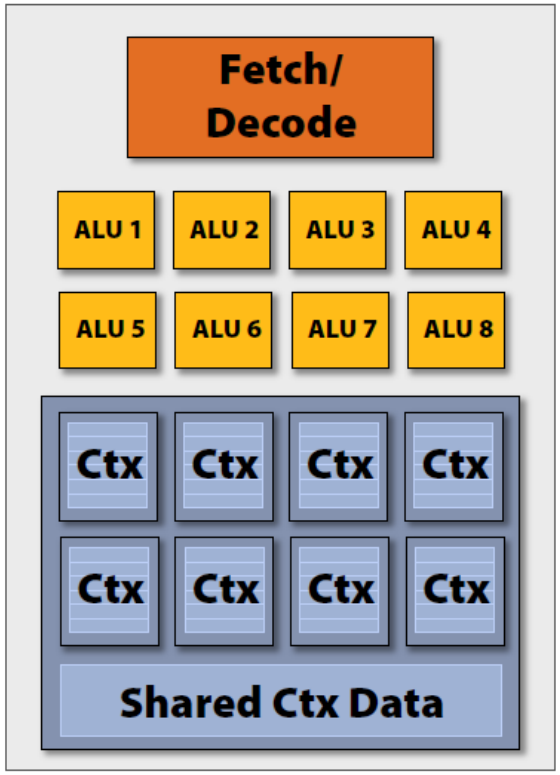
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

"From Shader Code to a Teraflop: How GPU Shader Cores Work", Kayvon Fatahalian, Stanford University

Modifying the shader

SIGGRAPH2008




```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```

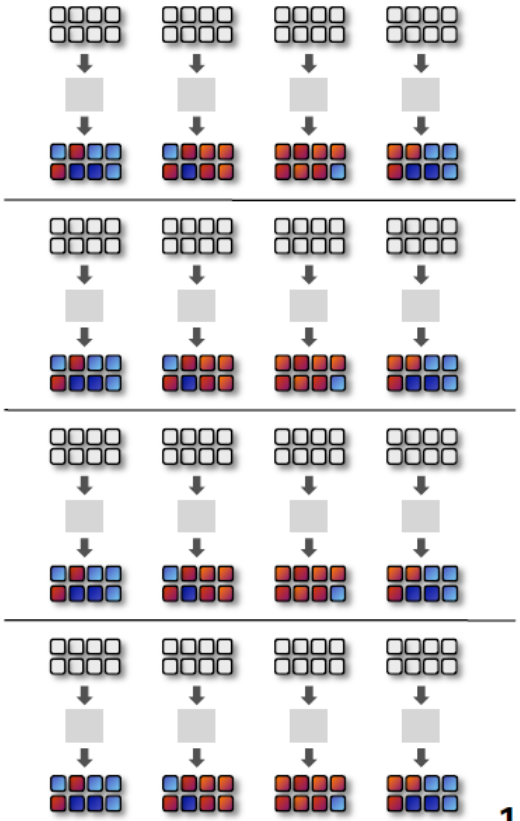
New compiled shader:

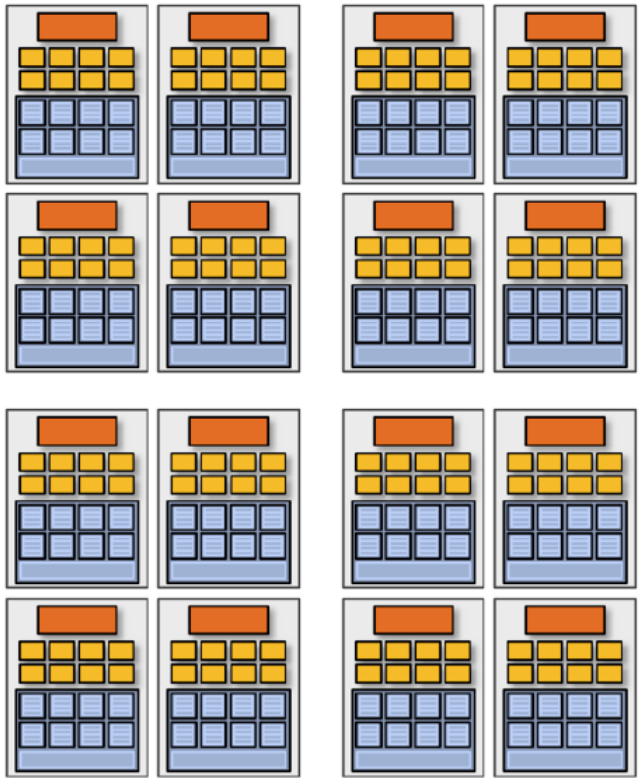
Processes 8 fragments using vector ops on vector registers

"From Shader Code to a Teraflop: How GPU Shader Cores Work", Kayvon Fatahalian, Stanford University

128 fragments in parallel

SIGGRAPH2008





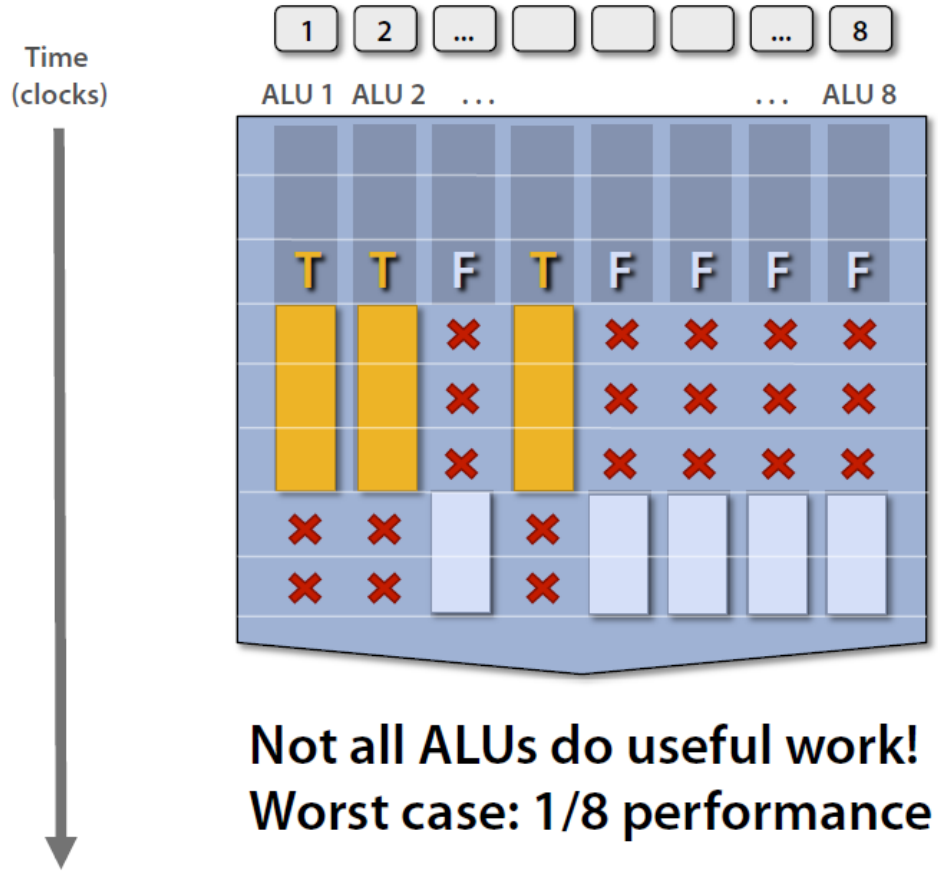
16 cores = 128 ALUs
= 16 simultaneous instruction streams

Beyond Programmable Shading: Fundamentals

23

"From Shader Code to a Teraflop: How GPU Shader Cores Work", Kayvon Fatahalian, Stanford University

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```


"From Shader Code to a Teraflop: How GPU Shader Cores Work", Kayvon Fatahalian, Stanford University

GPU Programming Model

- Host = CPU
- Device = GPU
- Kernel
  - Function that executes on the device
  - Multiple threads execute each kernel

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


void main()
{
    .....
        increment_cpu(a, b, N);
}
```

CUDA program

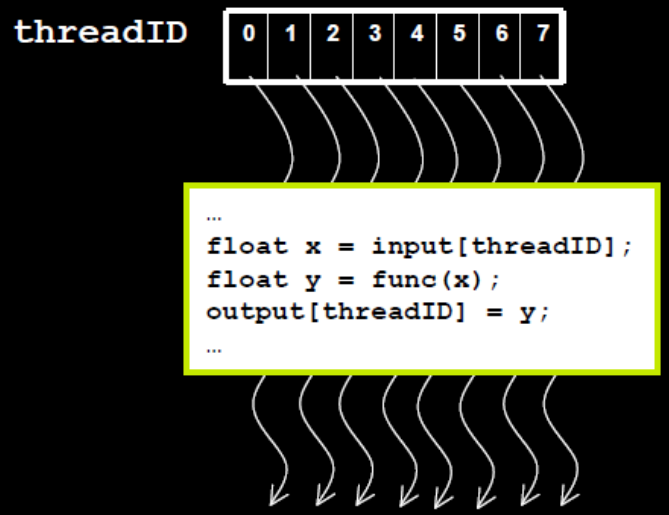
```
__global__ void increment_gpu(float *a, float b, int N)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < N)
            a[idx] = a[idx] + b;
}


void main()
{
    .....
        dim3 dimBlock (blocksize);
        dim3 dimGrid( ceil( N / (float)blocksize)  );
        increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```


Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

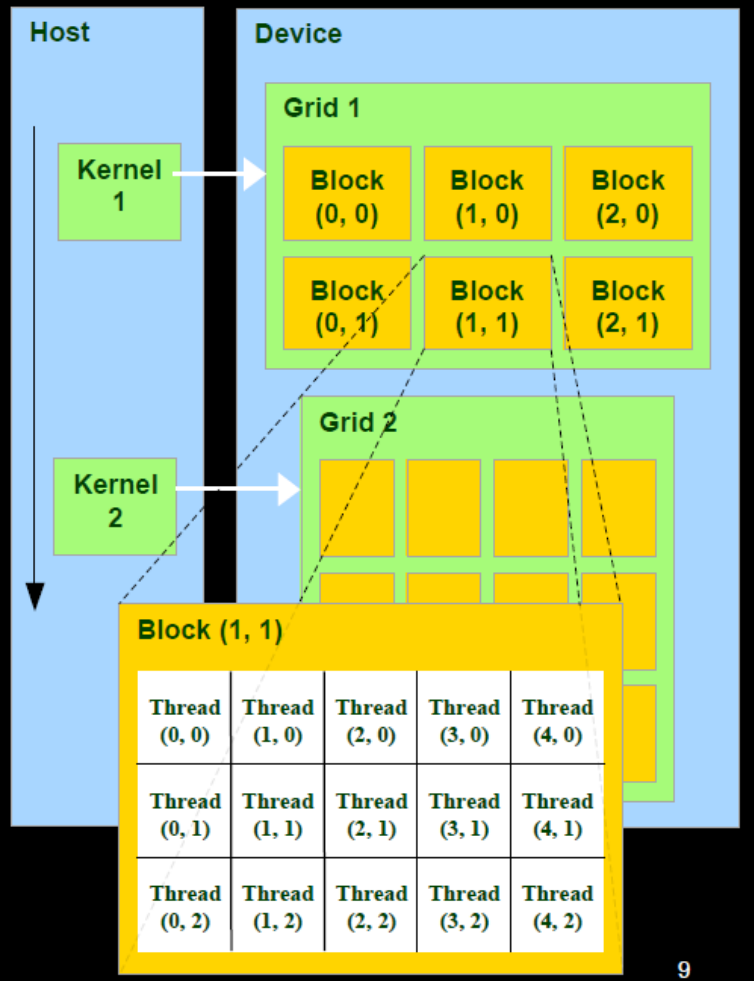


Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf

CUDA Programming Model

A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf

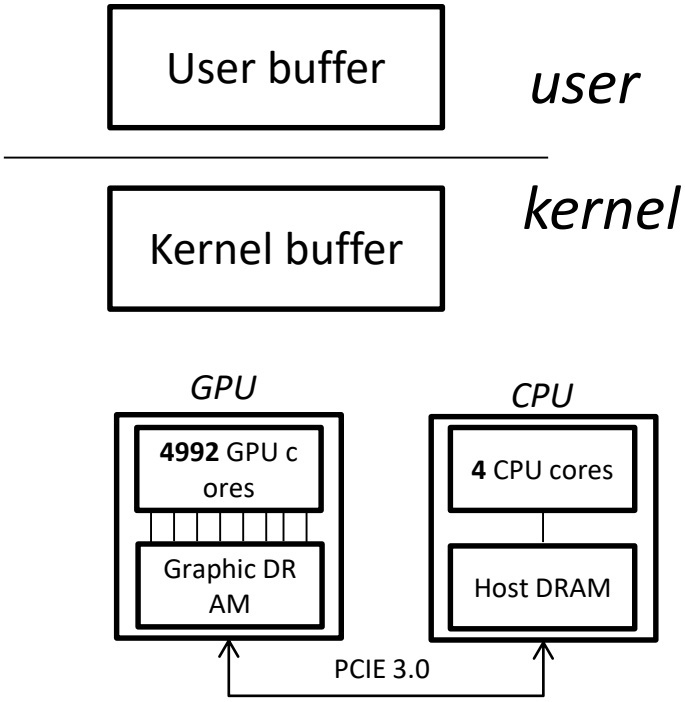
Memory Model

- **Registers**
  - Per thread
  - Data lifetime = thread lifetime
- **Local memory**
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime
- **Shared memory**
  - Per thread block on-chip memory
  - Data lifetime = block lifetime
- **Global (device) memory**
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation
- **Host (CPU) memory**
  - Not directly accessible by CUDA threads

Source: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf

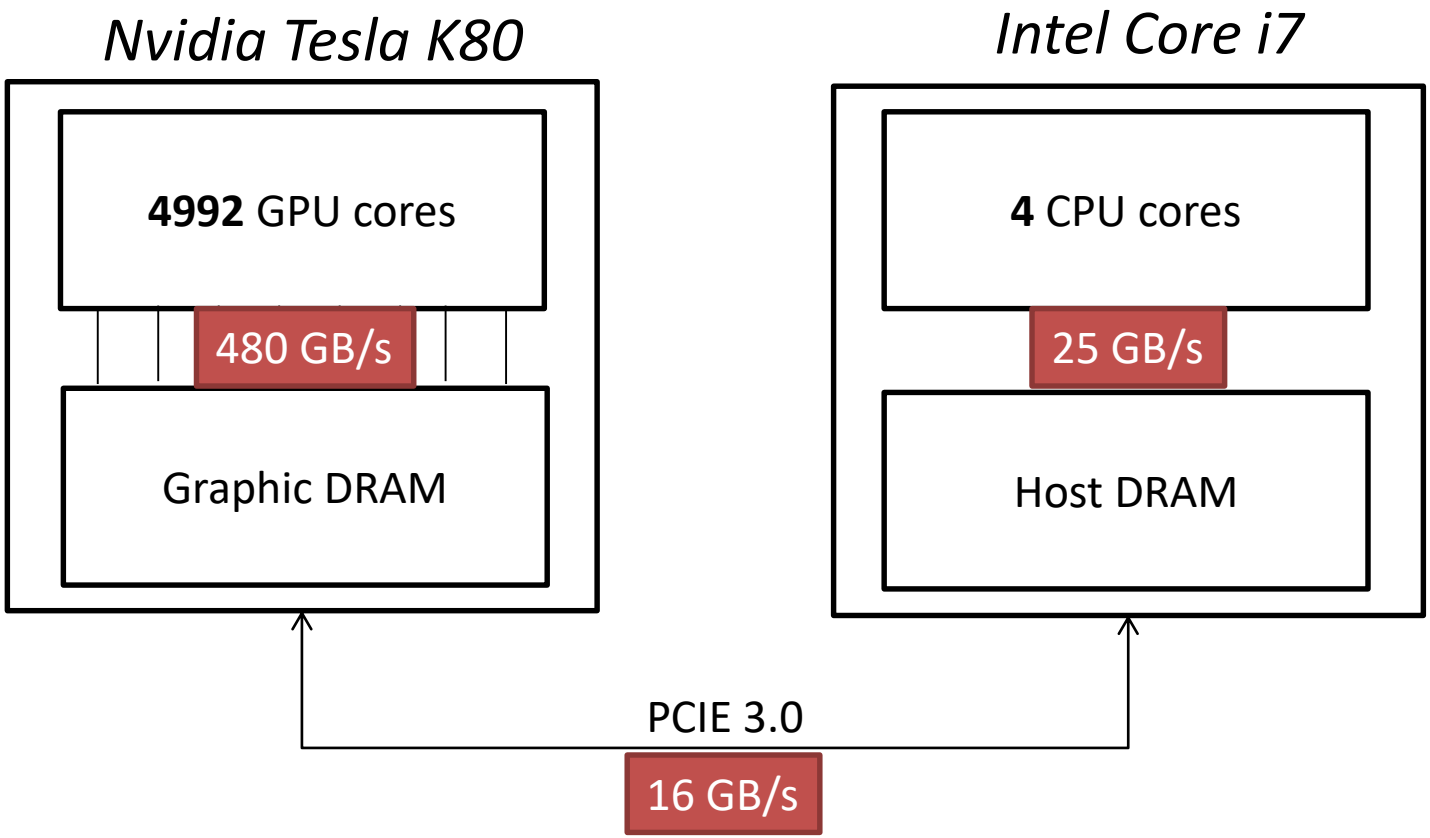
Challenges

- Data movement problem
  - User <-> kernel
  - Host mem <-> gpu mem
  - Other device <-> gpu



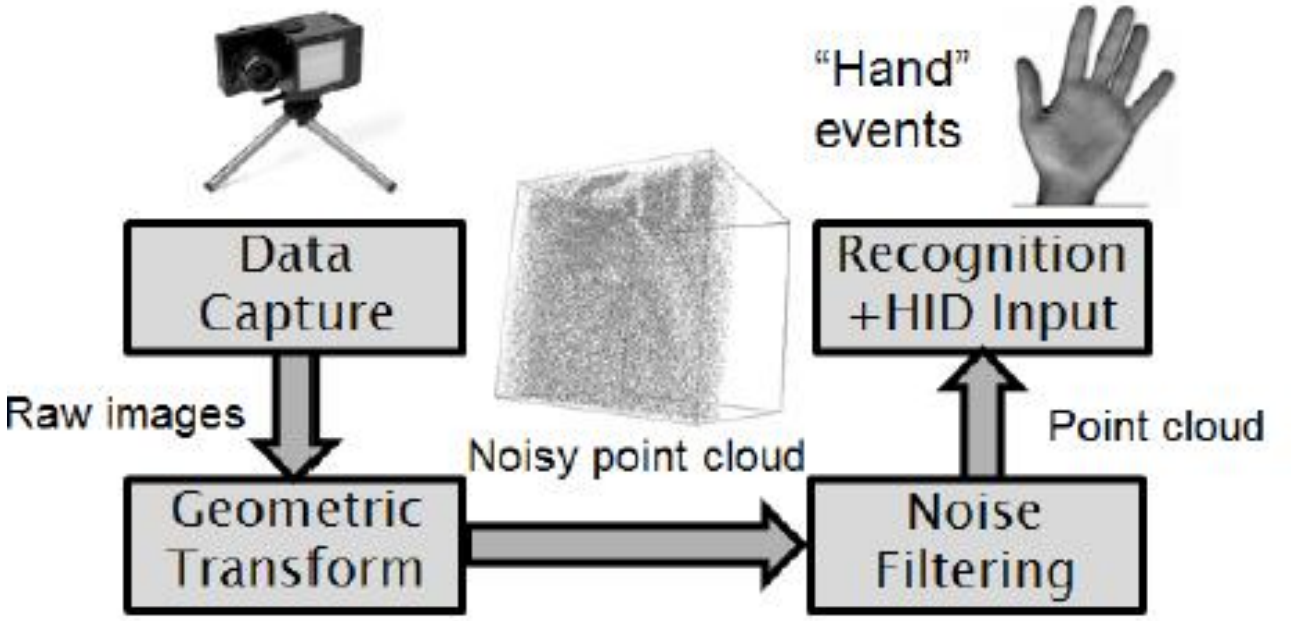
- Scheduling problem
  - No way to prioritize important GPU kernels
  - Unsynchronized CPU and GPU scheduling
  - No way to **preempt** once the kernel is launched.

CPU + Discrete GPU



Data transfer is the bottleneck

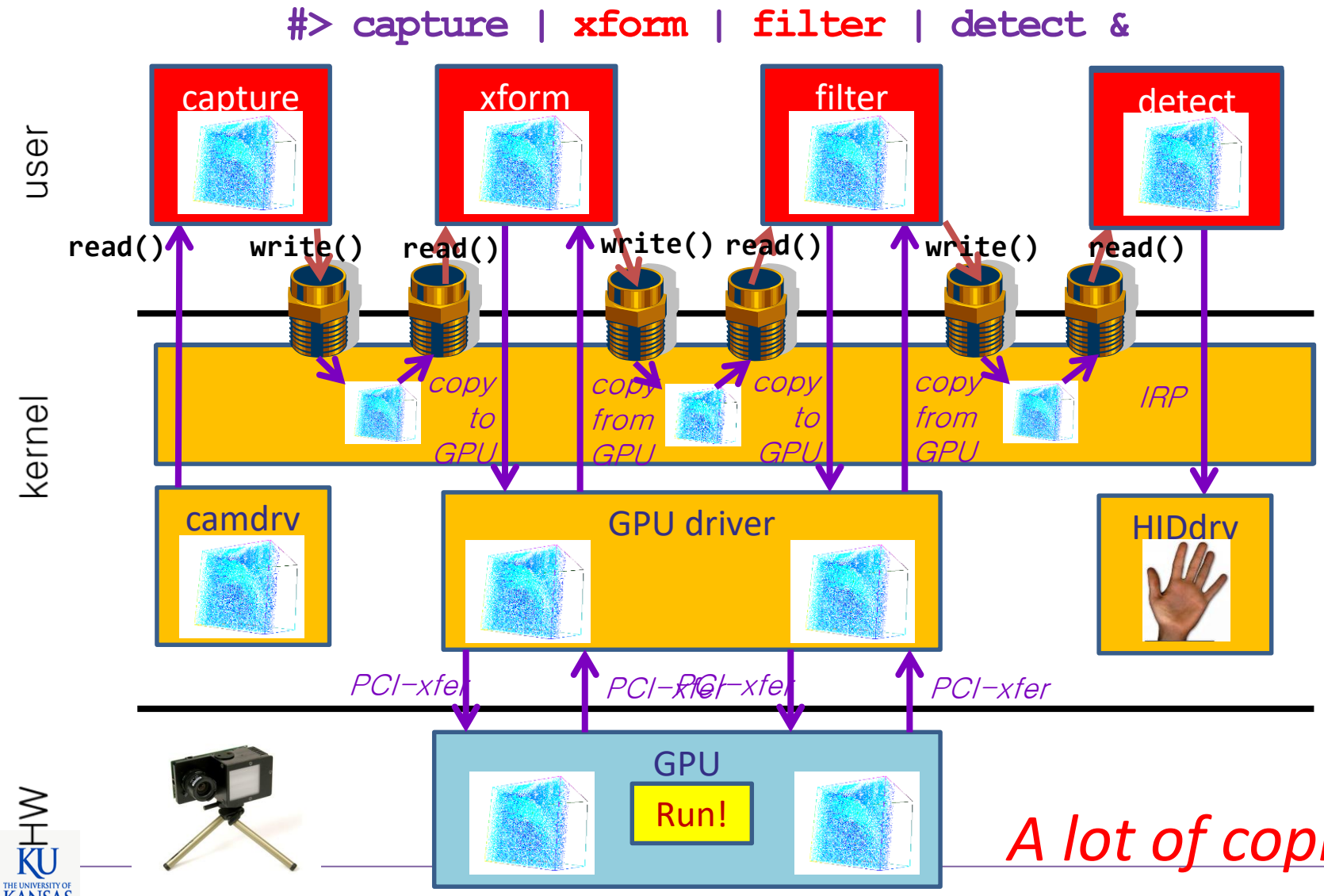
An Example




```
catusb | xform | filter | hidinput &
```

CPU *GPU* *GPU* *CPU*

Inefficient Data migration

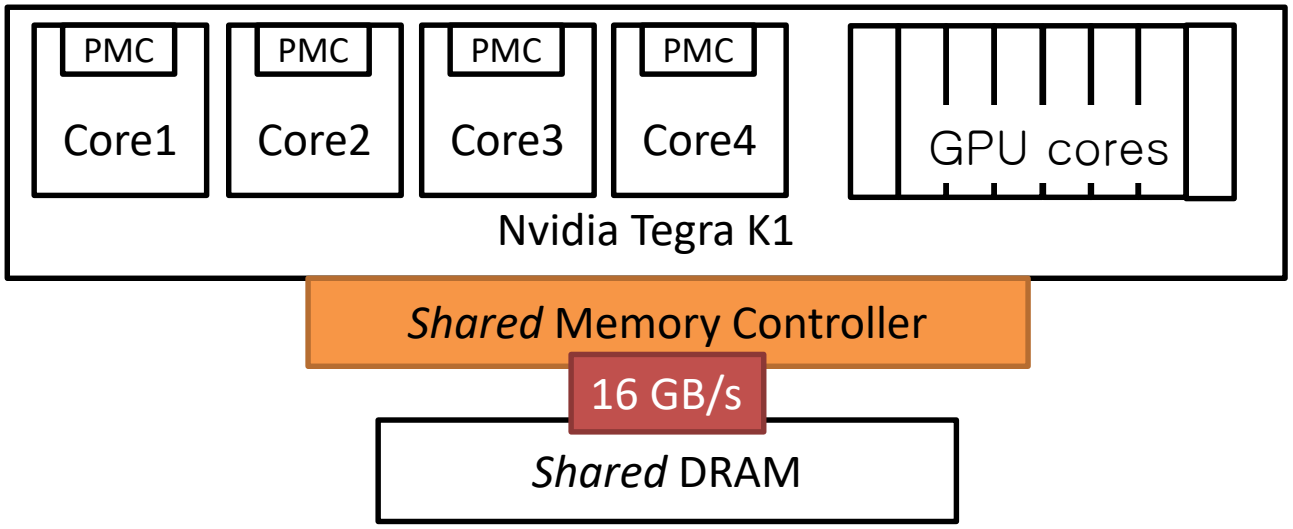


A lot of copies

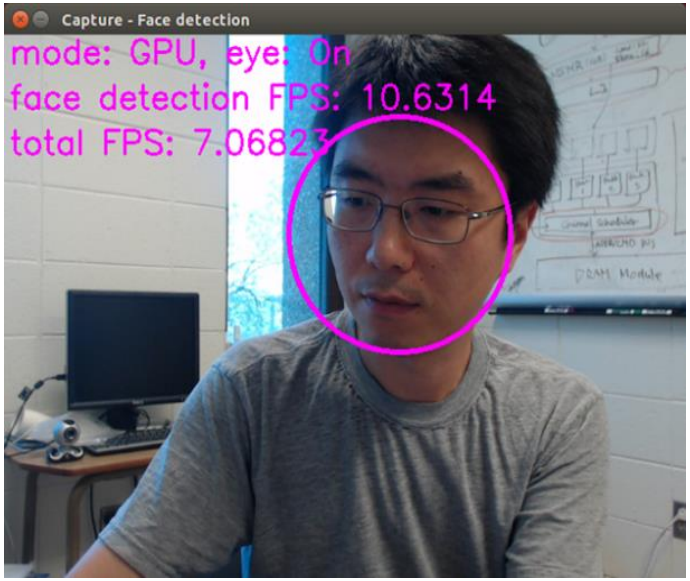
Acknowledgement: This slide is from the [paper's author's slide]

Heterogeneous Processor

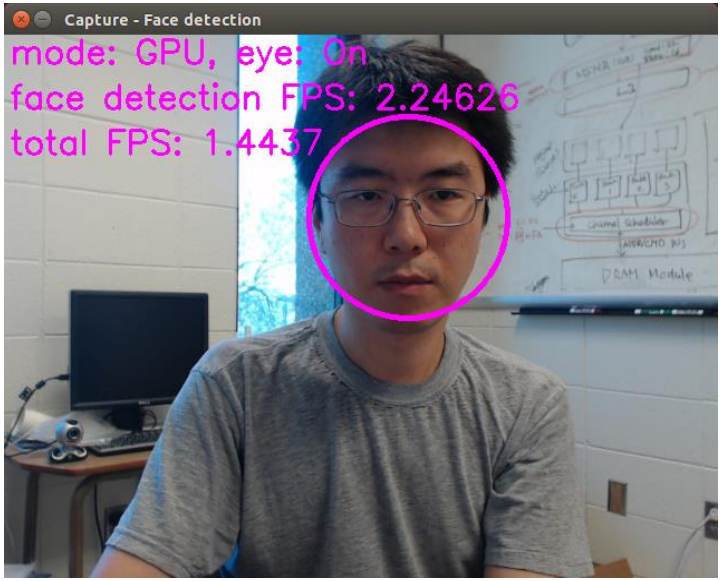
- Tighter integration of CPU and GPU
  - Memory is shared by both CPU and GPU



Memory Bandwidth Contention Between CPU and GPU



Run alone

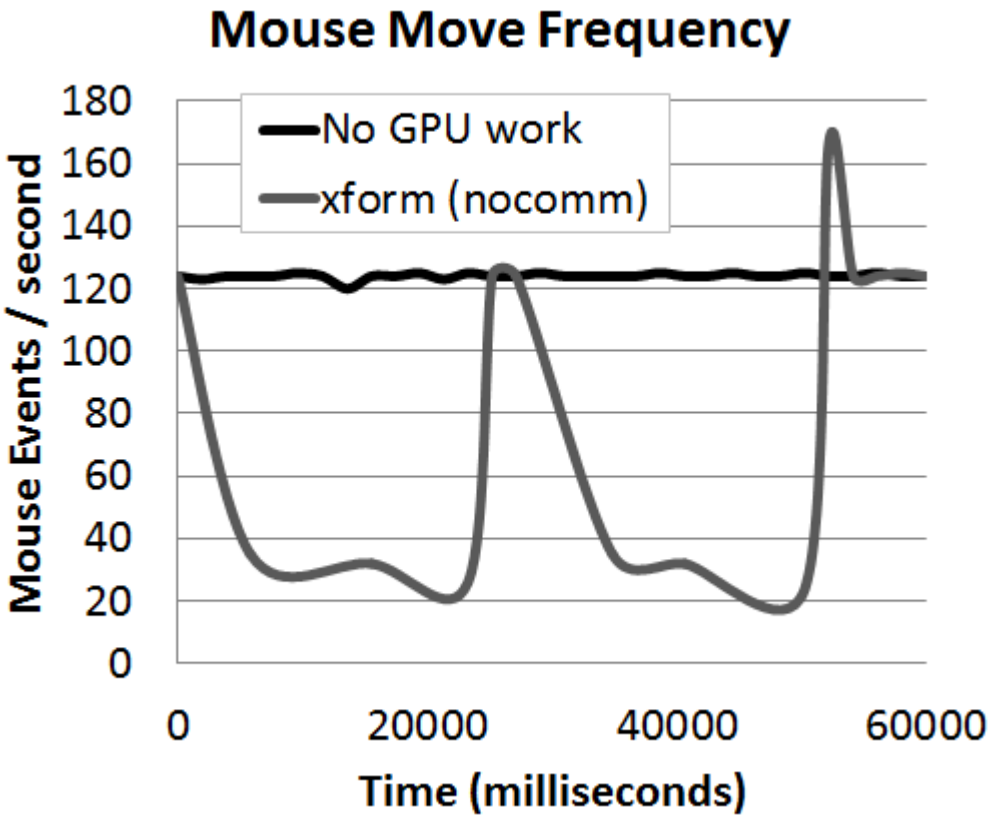


w/ CPU co-runners

Co-scheduling memory intensive CPU task affects GPU performance (5X slowdown)

KU THE UNIVERSITY OF KANSAS

Uncoordinated CPU and GPU Scheduling



CPU priorities do not apply to GPU

PTask: Operating System Abstractions To Manage GPUs as Compute Devices, SOSP'11

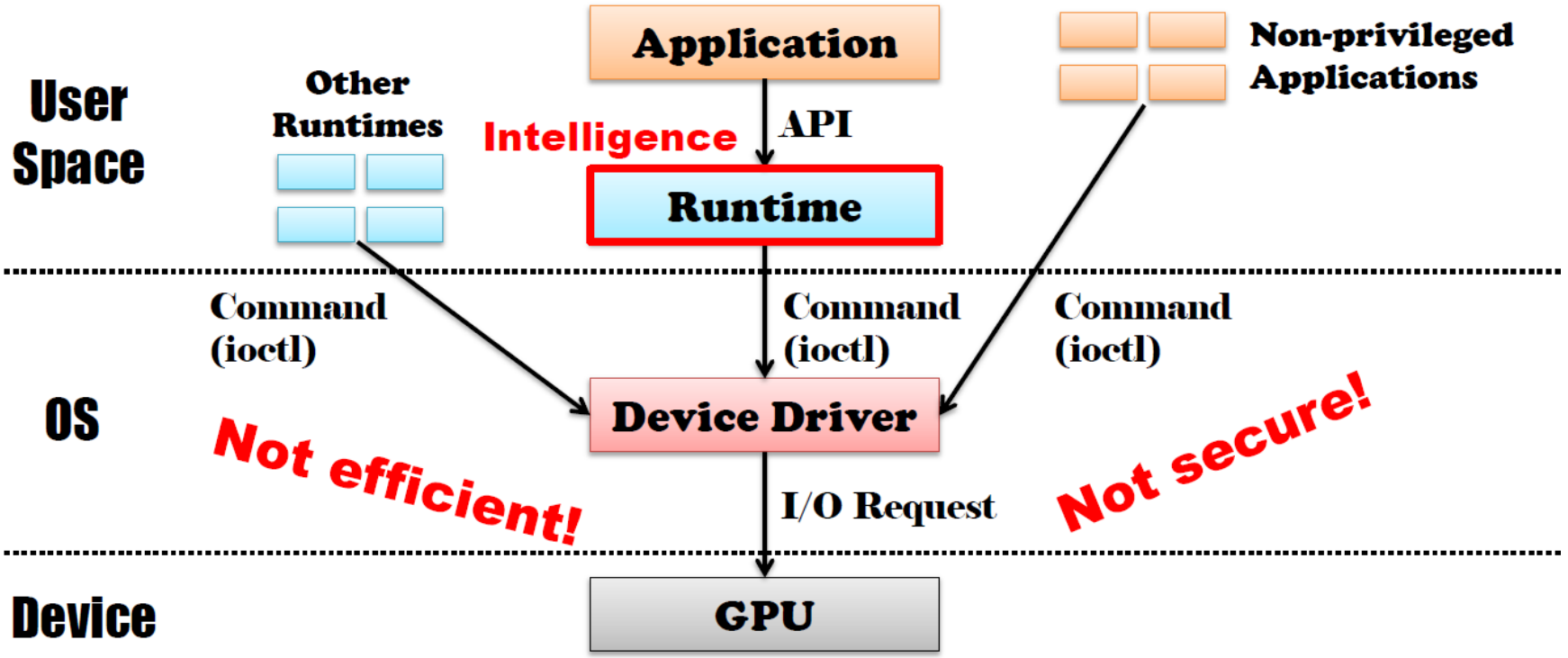
Real-Time GPU Management

- Goals
  - Priority scheduling among GPU tasks
  - GPU bandwidth (time) guarantee

- Frameworks
  - Timegraph
  - Gdev
  - GPUSync

KU THE UNIVERSITY OF KANSAS

Software Architecture



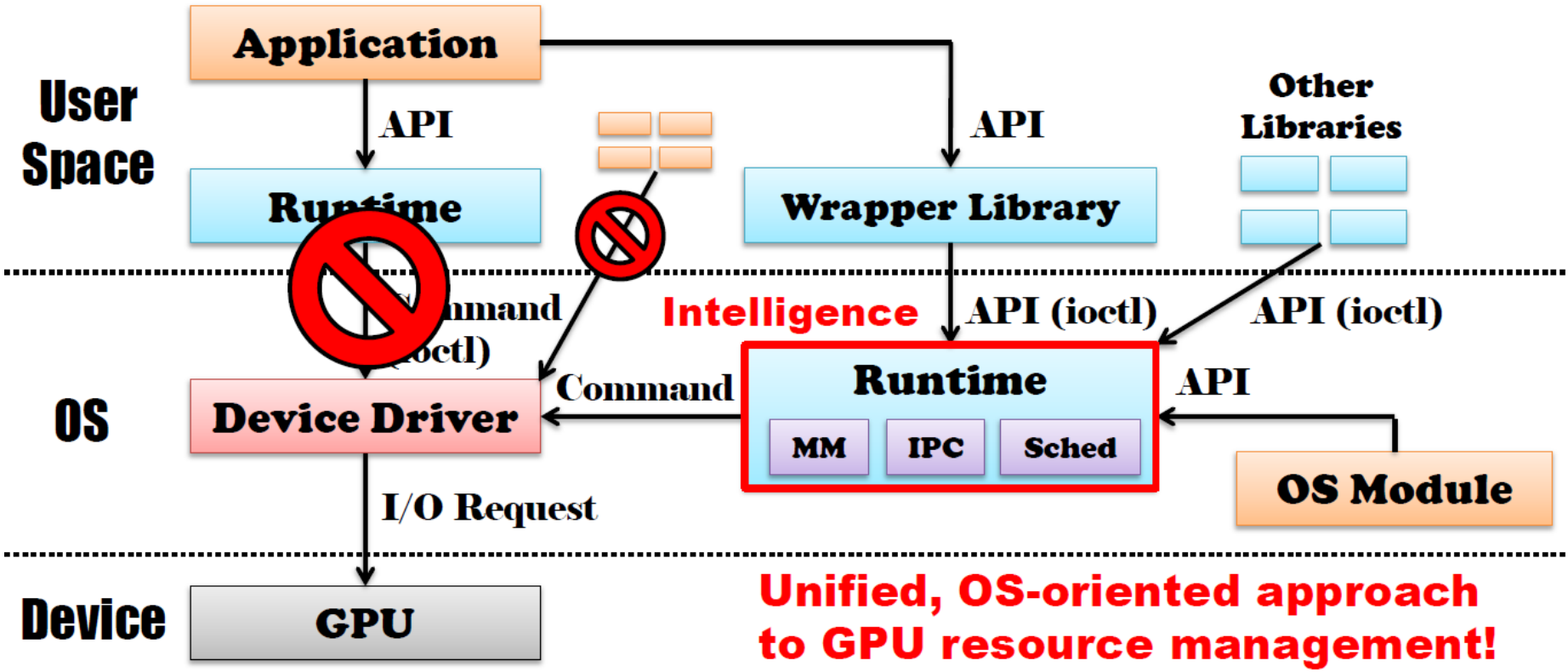
KU THE UNIVERSITY OF KANSAS

Acknowledgement: This slide is from the [paper author's slide]

GDev

- Runtime at the OS level
  - Better protection
  - Enable kernel code (e.g., filesystem) to use GPU
- Device memory management
  - Enable overcommit
  - Support shared device memory
- GPU virtualization
  - Expose multiple virtual GPUs to users
  - Support scheduling among the vGPUs

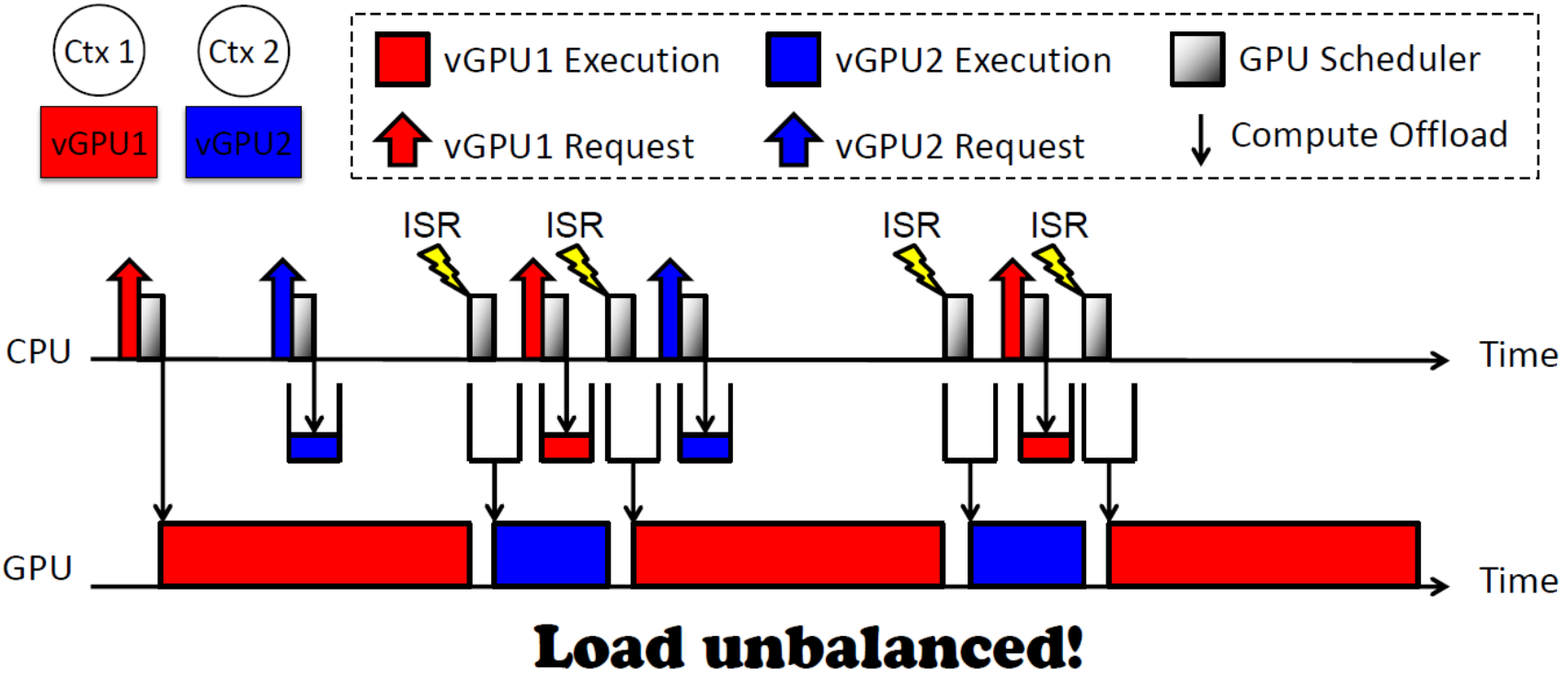
GDev Software Architecture



- Better protection
- GPU accelerated kernel (e.g., fs)

Acknowledgement: This slide is from the paper author's slide

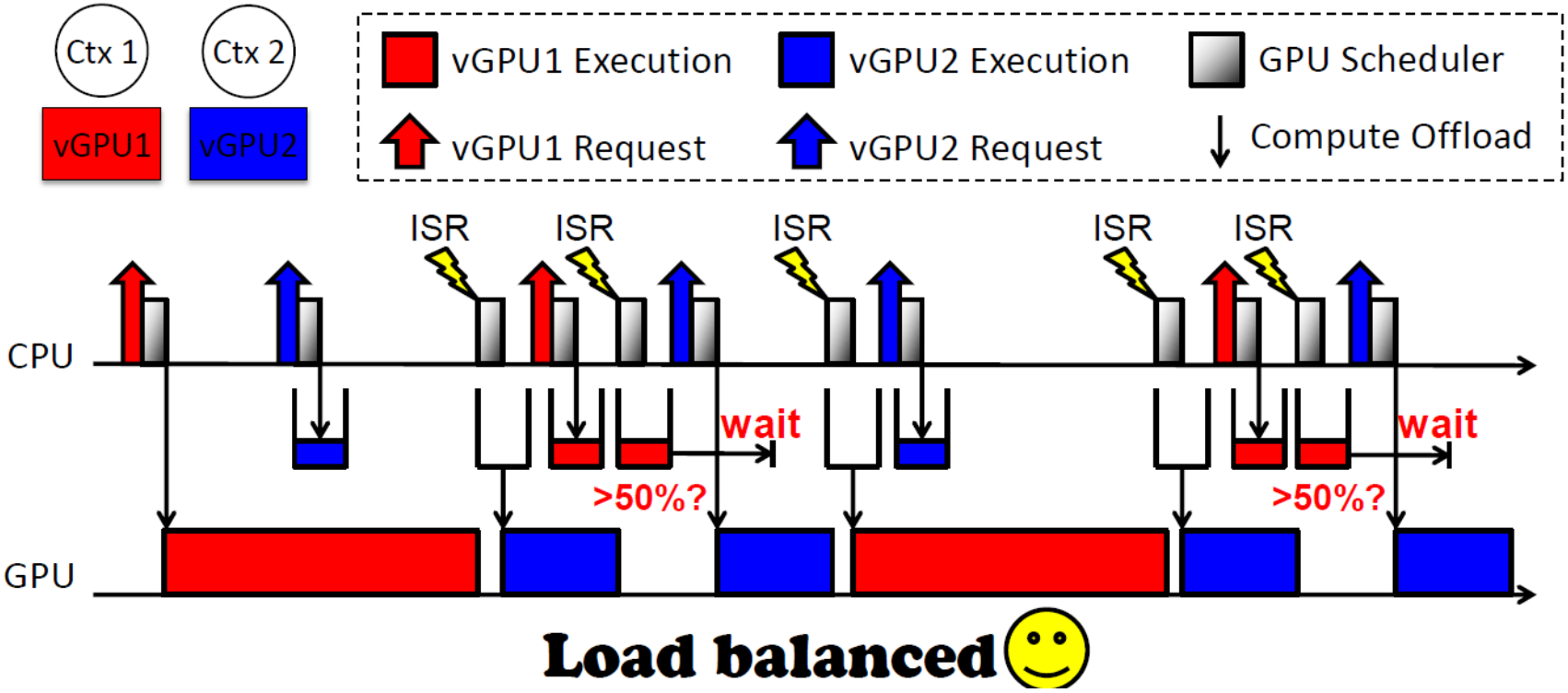
Previous GPU Scheduling



- TimeGraph [ATC11]
  - GPU commands are non-preemptive
  - Cause long, potentially unbounded, delay to high priority tasks

Figure source: paper author's slide

GDev's BAND Scheduler



- Non-work conserving scheduler
  - Monitor consumed b/w, add some delay
- Recent related work: GPES [RTAS14]
  - Divide kernel into smaller ones, insert preemption points

(*) GPES: A Preemptive Execution System for GPGPU Computing, RTAS'14

Figure source: paper author's slide

Summary

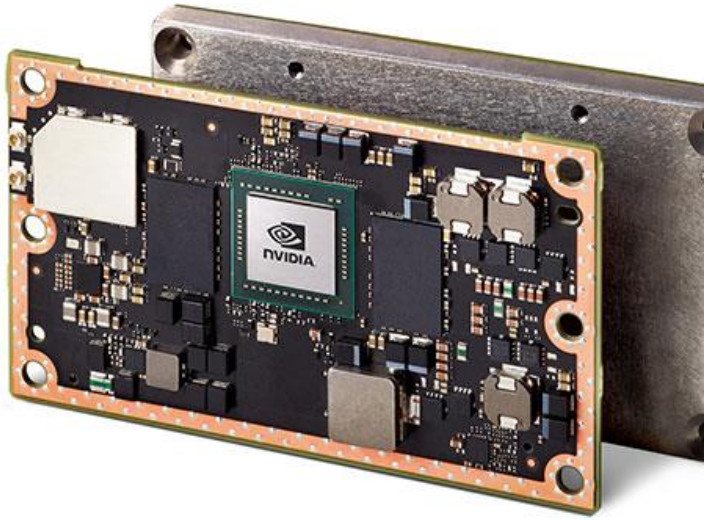
- GPU Architecture
  - Many simple in-order cores
- GPU Programming Model
  - SIMD
- Challenges
  - Data movement cost
  - Non-preemptive scheduling
  - Bandwidth bottleneck
- Real-Time Support
  - Priority and/or bandwidth based scheduling

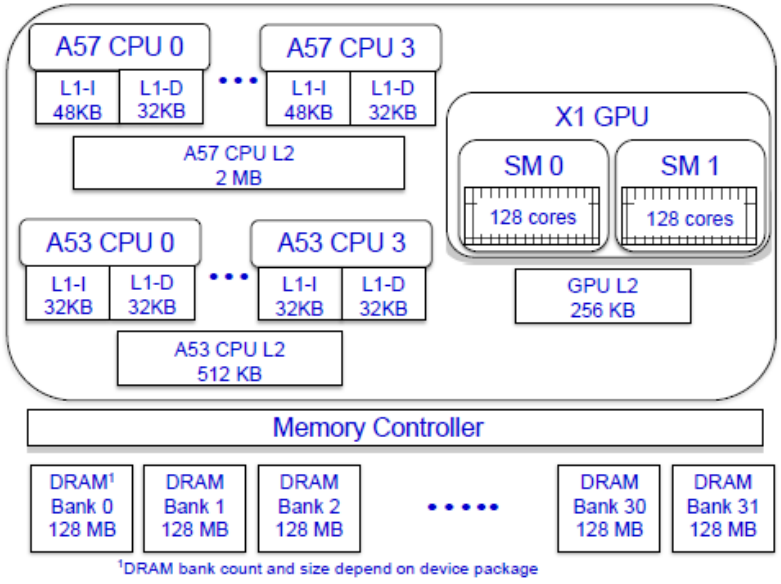
Real-Time Support for GPU (2/2)

GPU Management

Heechul Yun

NVIDIA Jetson Platform



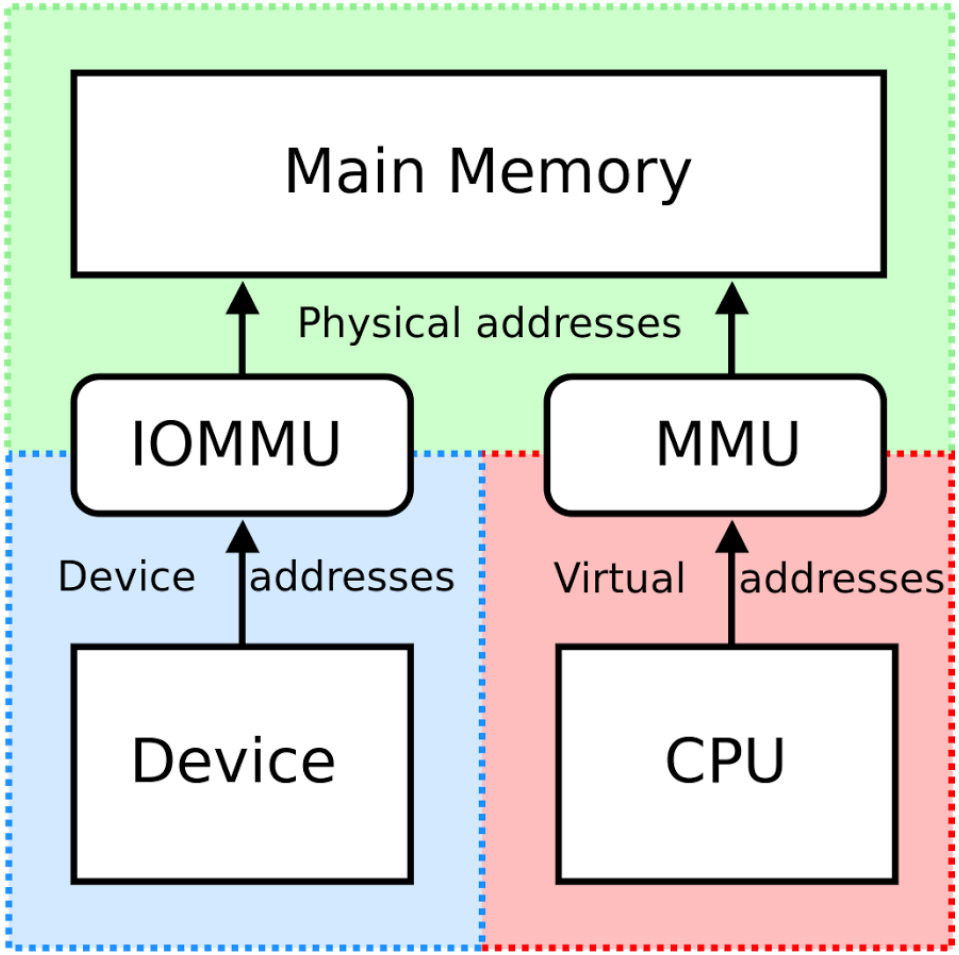


- Integrated CPU-GPU architecture
  - CPU and GPU share memory
  - cf. discrete GPUs

NVIDIA Jetson Platform

	Jetson TX2	Jetson TX1
GPU	NVIDIA Pascal™, 256 CUDA cores	NVIDIA Maxwell ™, 256 CUDA cores
CPU	HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2	Quad ARM® A57/2 MB L2
Video	4K x 2K 60 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (12-Bit Support)	4K x 2K 30 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (10-Bit Support)
Memory	8 GB 128 bit LPDDR4 58.3 GB/s	4 GB 64 bit LPDDR4 25.6 GB/s
Display	2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4	2x DSI, 1x eDP 1.4 / DP 1.2 / HDMI
CSI	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.2 (2.5 Gbps/Lane)	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.1 (1.5 Gbps/Lane)
PCIE	Gen 2 | 1x4 + 1x1 OR 2x1 + 1x2	Gen 2 | 1x4 + 1x1
Data Storage	32 GB eMMC, SDIO, SATA	16 GB eMMC, SDIO, SATA
Other	CAN, UART, SPI, I2C, I2S, GPIOs	UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0	
Connectivity	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth	
Mechanical	50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)	

Heterogeneous System Architecture (HAS)



Problems of Sharing Memory

- Bandwidth
  - GPU is a heavy bandwidth consumer
  - CPU task can suffer.

Benchmark	Solo	Co-run (unregulated)	Co-run (regulated)	Gain (%)
Face	22.5	14.8	17.9	39.7
Hog	19.2	12.2	16.4	59.8
Flow	11.1	8.4	10.0	58.3

KU THE UNIVERSITY OF KANSAS

Problems of Sharing Memory

- Cache coherency
  - CPU has caches, GPU has caches
    = Multiple copies of the same memory block
  - Updated cacheline in one cache must be visible in other caches