

IIT KANPUR
Indian Institute of Technology Kanpur

CS623: GPU Architecture and Programming

GPU Device Interface

PCI Subsystem

- PCI can be viewed as tree-like organization of I/O devices
- Each device mapped to PCI bus can be examined based on the IDs (device, vendor etc.)

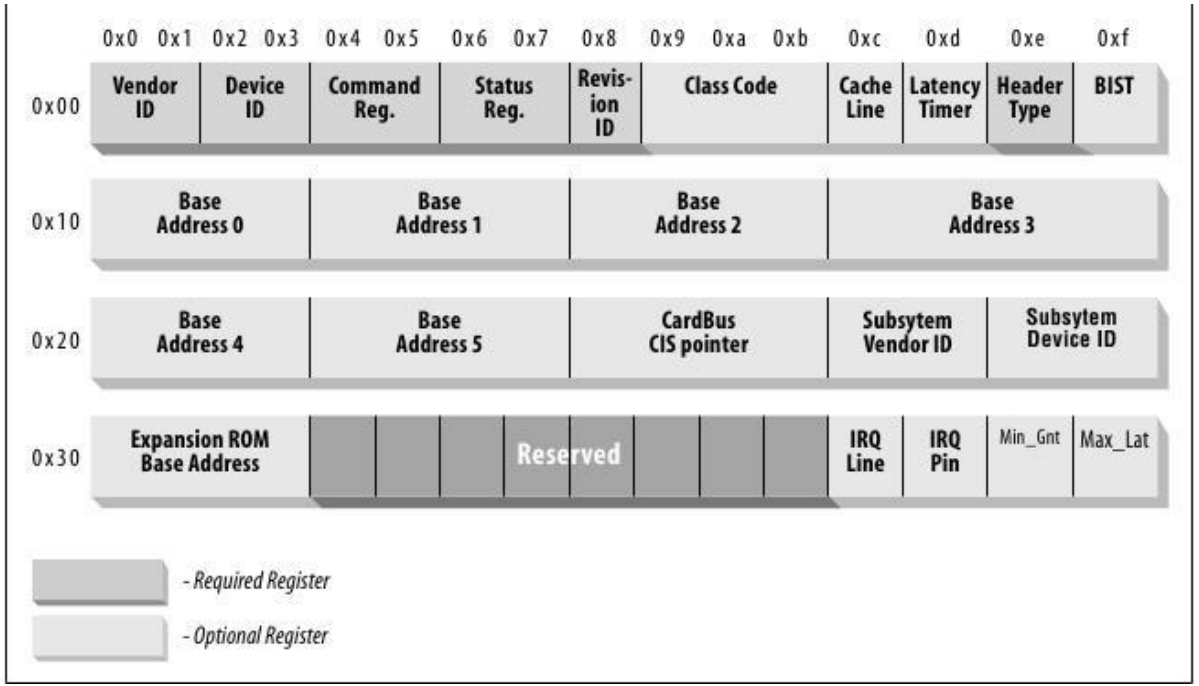


Image source: Linux Device Drivers, Ch12

PCI Layout

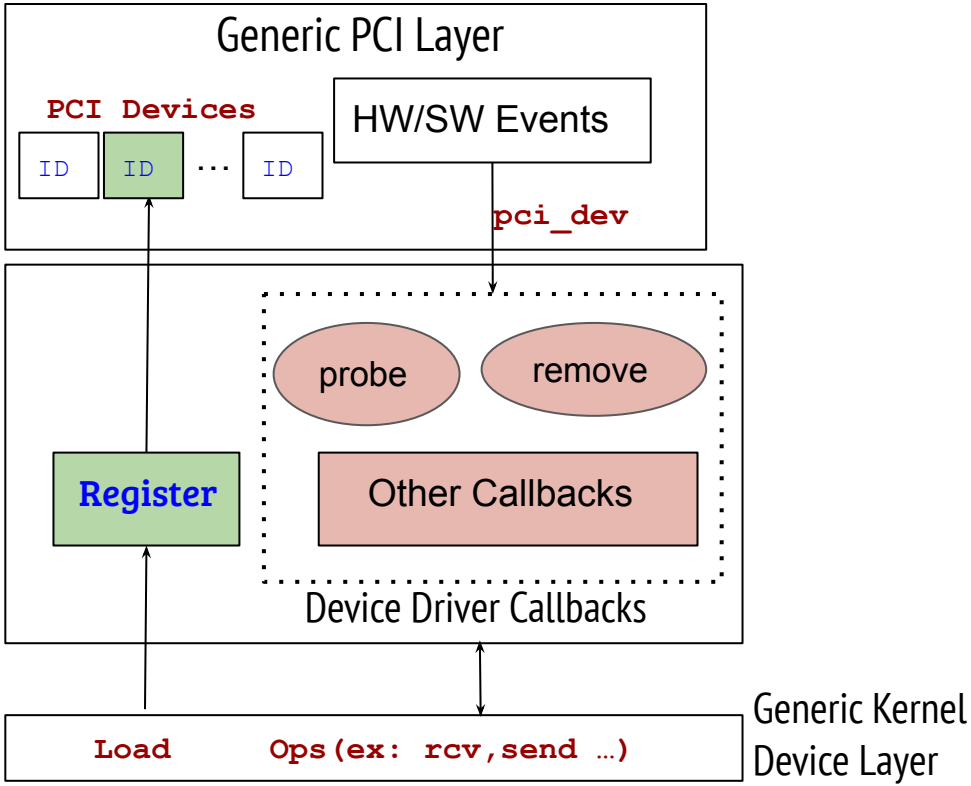
- PCI can be viewed as tree-like organization of I/O devices
- Each device mapped to PCI bus can be examined based on the IDs (device, vendor etc.)
- Devices can be found by querying the PCI controller and scanning the mapped devices though the nested laying of
  - Domain
  - Bus
  - Device
  - Function

PCI Layout

- PCI can be viewed as tree-like organization of I/O devices
- Each device mapped to PCI bus can be examined based on the IDs (device, vendor etc.)
- Devices can be found by querying the PCI controller and scanning the mapped devices though the nested laying of
  - Domain
  - Bus
  - Device
  - Function
- Linux kernel pre-creates this list and invokes the probe method of the matching driver when a driver is registered
- The “lspci” user space utility (and the /sys/bus/... interface) can be used to examine

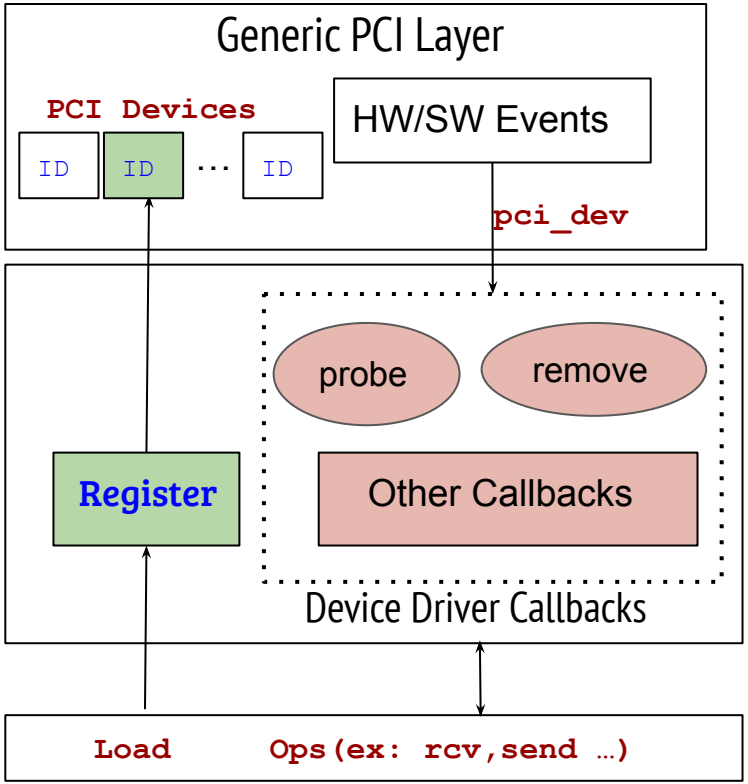
Linux PCI device driver

- A PCI device driver must register itself using an object of type “struct pci_driver”



Linux PCI device driver

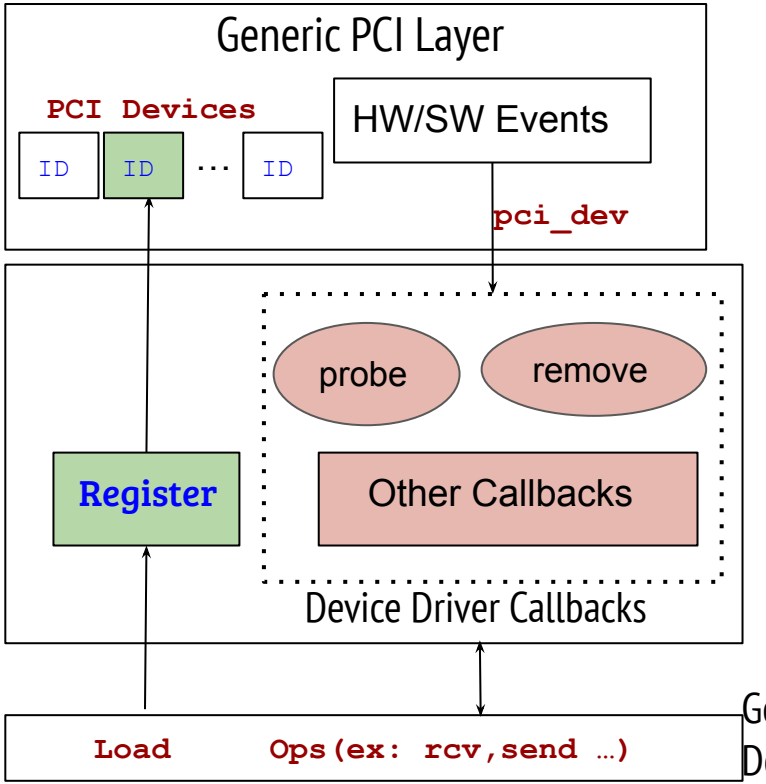
- A PCI device driver must register itself using an object of type “struct pci_driver”



- While registering a driver for a PCI device, an ID table containing a list of ID entries (*vendor, device, subvendor, subdevice*) are passed to the PCI layer to match a device for this driver
- A probe method (part of pci_driver structure) is registered as a call back

Linux PCI device driver

- A PCI device driver must register itself using an object of type “struct pci_driver”

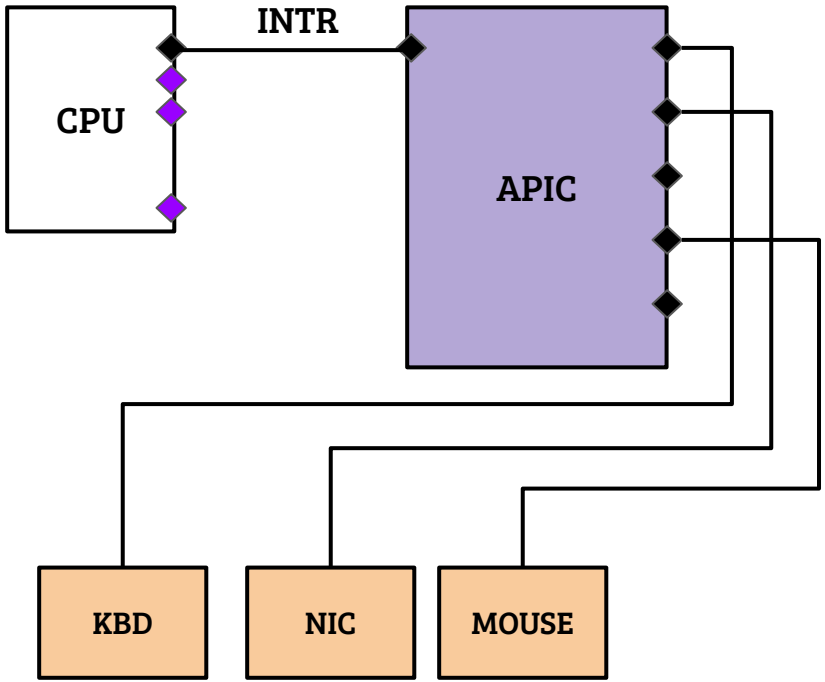


- While registering a driver for a PCI device, an ID table containing a list of ID entries (*vendor, device, subvendor, subdevice*) are passed to the PCI layer to match a device for this driver
- A probe method (part of pci_driver structure) is registered as a call back
- The generic PCI layer invokes the probe method to allow the device driver to perform device and software initializations (device API for the generic device layer)

Useful Kernel PCI helpers

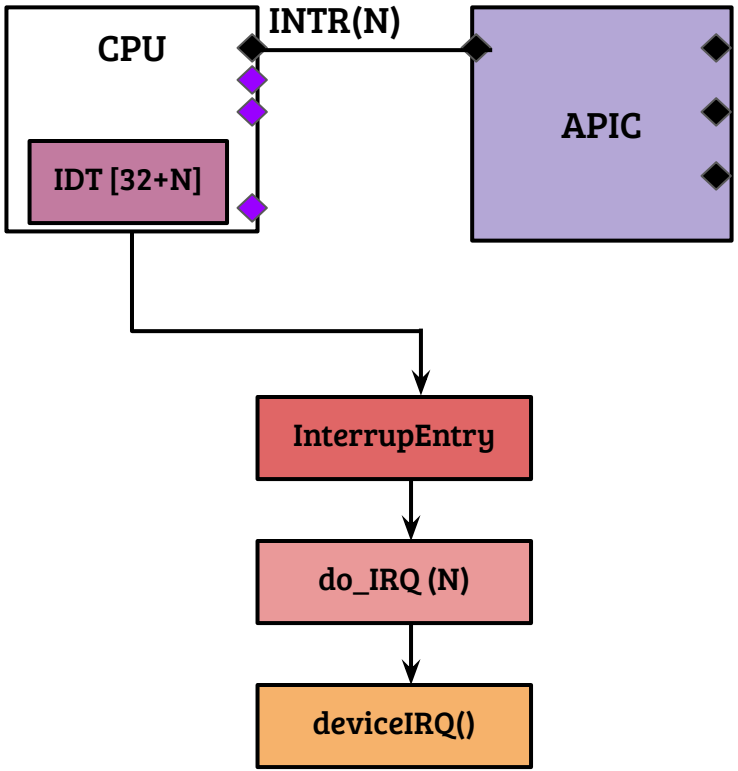
- Most PCI device drivers read and examine the BAR registers
- Reading the PCI configuration for any device (@PCI controller)
  - pci_read_config_byte/word/dword(pci_dev, offset, into)
  - pci_write_config_byte/word/dword(pci_dev, offset, from)
- Most PCI device drivers read and examine the BAR registers (BAR0, BAR1… Bar5)
  - pci_resource_flags(pci_dev, bar)
  - Type of resource (IO or MEM) can be examined, accordingly used for PIO or MMIO
  - pci_request_regions to check the I/O "address" resource availability
  - pci_resource_start(pci_dev, bar) returns handle to start of an I/O resource
- For MMIO resources
  - pci_ioremap_bar(pci_dev, barnum)
  - Returns a VA handle to operate on the device

Interrupt architecture - PIC and APIC



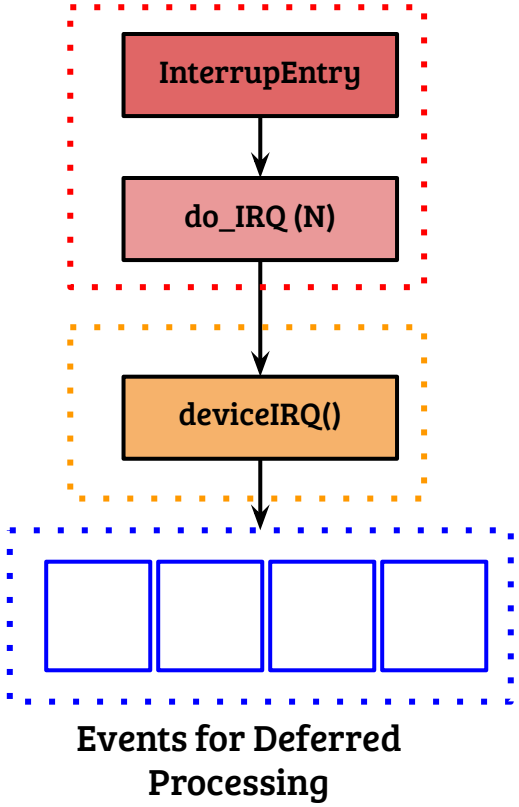
- Every device attached to the APIC is configured with a unique IRQ number
- APIC saves the IRQ in a control port register and raise CPU interrupt line on receipt of device interrupt
- CPU reads the IRQ number and invokes the interrupt handler
- Waits for acknowledgement before clearing the INTR line
- Selective disabling of IRQs possible
  - != cli (CPU interrupt disable)
  - New interrupts not lost

Interrupt handling



- IDT configured to load the interrupt execution context (CPL and stack)
- Interrupt entry: save regs, switch CR3 if needed
- do_IRQ checks the descriptor flags and invokes the real handler
- The device driver handler implements the device specific functionalities
- When is the interrupt acknowledged (i.e., INTR is cleared)?
- How long is the device interrupt masked?
- Not all interrupts can be handled quickly, e.g., transferring data from CPU to GPU memory

Interrupt handling in three stages



- Critical tasks: Interrupt context setup, APIC acknowledgement
- Semicritical: Accessing/updating device state, e.g., update fault buffer queue
- Deferrable: Actions that are device independent e.g., GPU fault handling

Interrupts in PCI devices

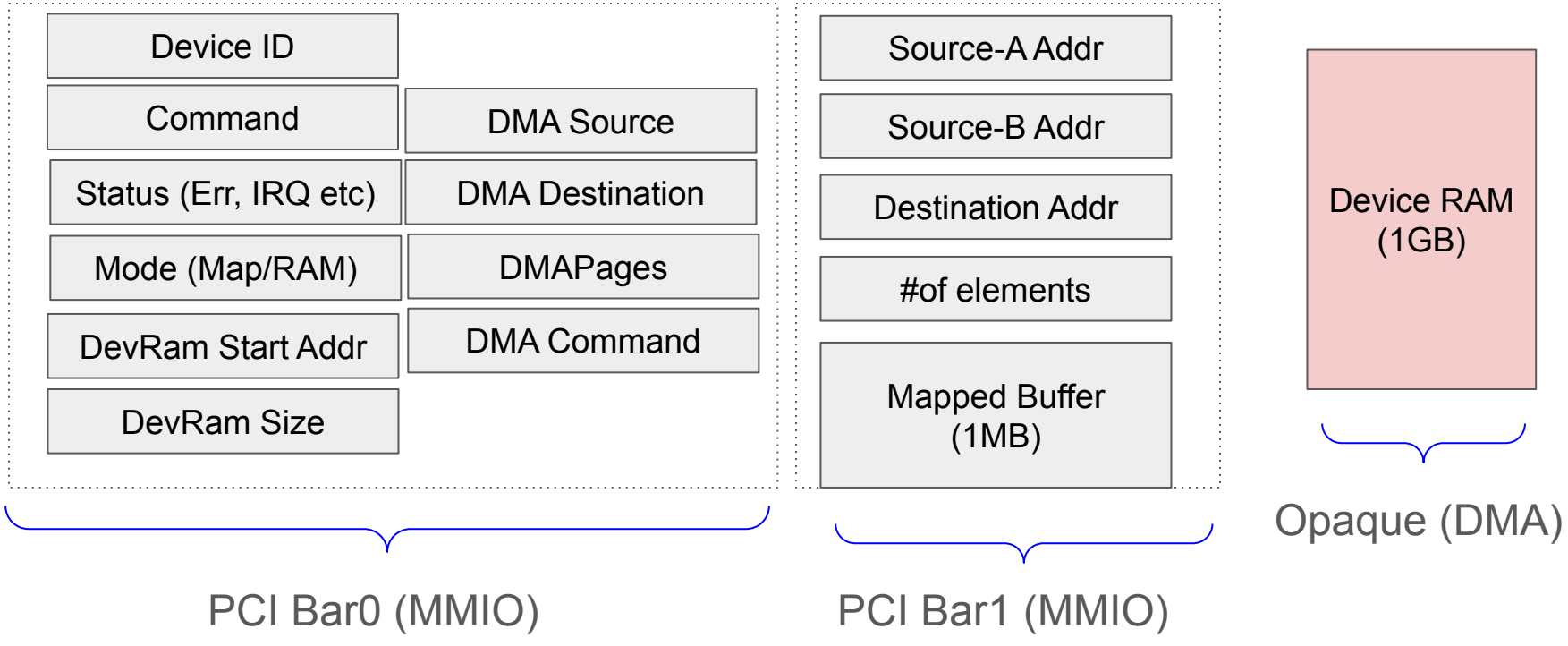
- Examining interrupt capability
  - Reading the PCI config using pci_read_config_byte (IRQ pin and IRQ line) directly
  - Using the PCI helper APIs such as
    - pci_alloc_irq_vectors
    - pci_irq_vector
    - request_irq
- Interrupt handler
  - The device level callback for interrupt handling is registered during request_irq
  - The handler code must determine if the IRQ belongs to the device, why? And How?
    - IRQ may be shared across many devices
    - By checking the interrupt status register of the device

PCI Mapping of NVIDIA GPUs

GP104GL [Quadro P5000] IRQ 22
Memory at fa000000 (32-bit, non-prefetchable) [size=16M]
Memory at 383800000000 (64-bit, prefetchable) [size=256M]
Memory at 383810000000 (64-bit, prefetchable) [size=32M]
I/O ports at c000 [size=128]

GA102GL [A40] IRQ 22
Memory at fc000000 (32-bit, non-prefetchable) [size=16M]
Memory at 1000000000 (64-bit, prefetchable) [size=64G]
Memory at 2000000000 (64-bit, prefetchable) [size=32M]

Devil in the details: An example device (Acceldev)



- Intended usage: Perform vector operations after moving vectors into the device
- Commands: Copy (from/to) device, vector add/sub etc.

User program schematic for Acceldev

```
int *vec_s1 = alloc_and_init(4096 *sizeof(int);
int *vec_s2 = alloc_and_init(4096 *sizeof(int);
int *vec_dst = alloc(4096*sizeof(int);
```

Host Memory Allocation

```
acc_handle_t s1 = dev_alloc(4096*sizeof(int));
acc_handle_t s2 = dev_alloc(4096*sizeof(int));
acc_handle_t dst = dev_alloc(4096*sizeof(int));
```

Device Memory Allocation

```
acc_memcopy(s1, vec_s1, H2D);
acc_memcopy(s2, vec_s2, H2D);
```

Data Copy from Host to Device

```
acc_add(s1, s2, dst, 4096);
```

Launch Vector Add in the device

```
acc_memcopy(vec_dst. dst, D2H);
```

Data Copy from Device to Host

User program schematic for Acceldev

```
int *vec_s1 = alloc_and_init(4096 *sizeof(int);
int *vec_s2 = alloc_and_init(4096 *sizeof(int);
int *vec_dst = alloc(4096*sizeof(int);
```

Host Memory Allocation

```
acc_handle_t s1 = dev_alloc(4096*sizeof(int));
acc_handle_t s2 = dev_alloc(4096*sizeof(int));
acc_handle_t dst = dev_alloc(4096*sizeof(int));
```

Device Memory Allocation

Where and How to implement?

```
acc_memcopy(s1, vec_s1, H2D);
acc_memcopy(s2, vec_s2, H2D);
```

Data Copy from Host to Device

How to coordinate data transfer?

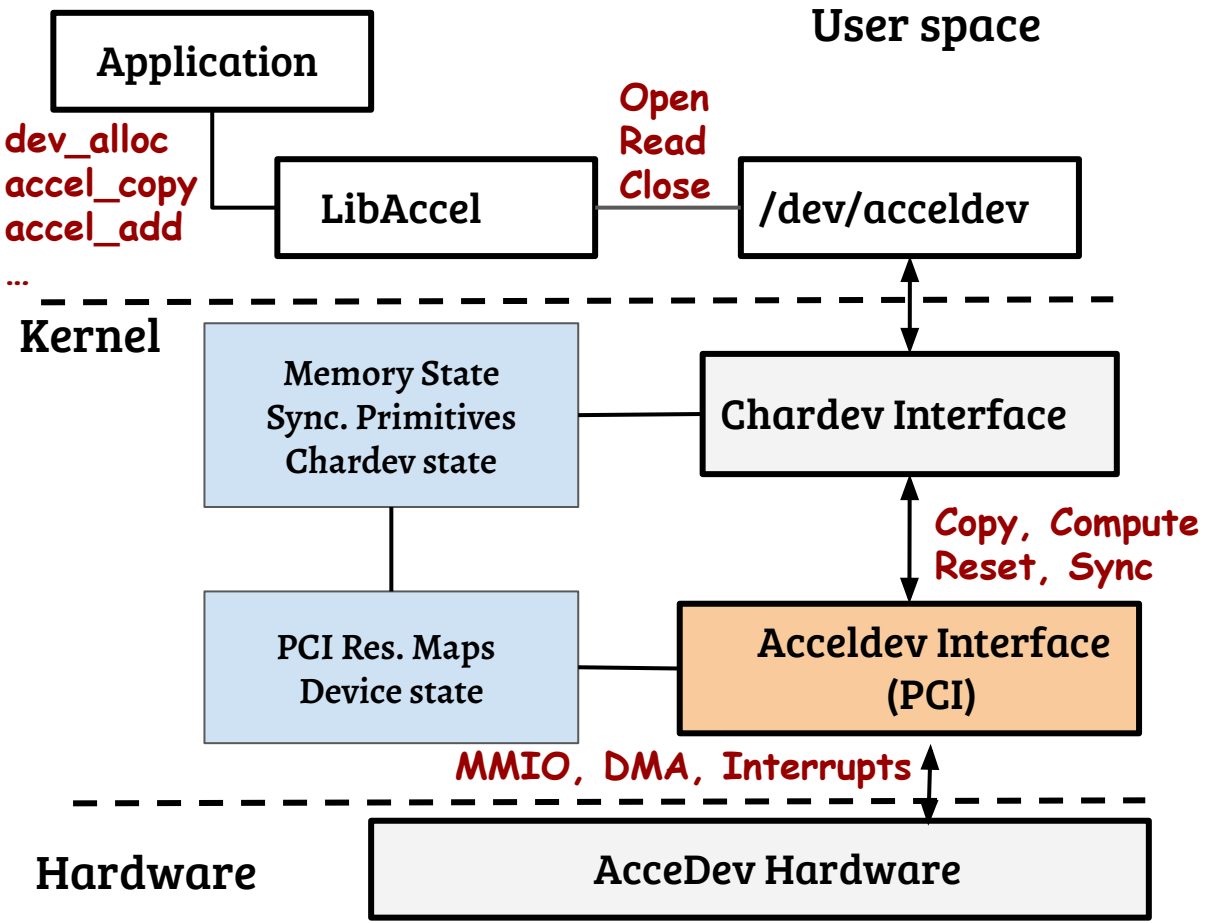
```
acc_add(s1, s2, dst, 4096);
```

Launch Vector Add in the device

```
acc_memcopy(vec_dst. dst, D2H);
```

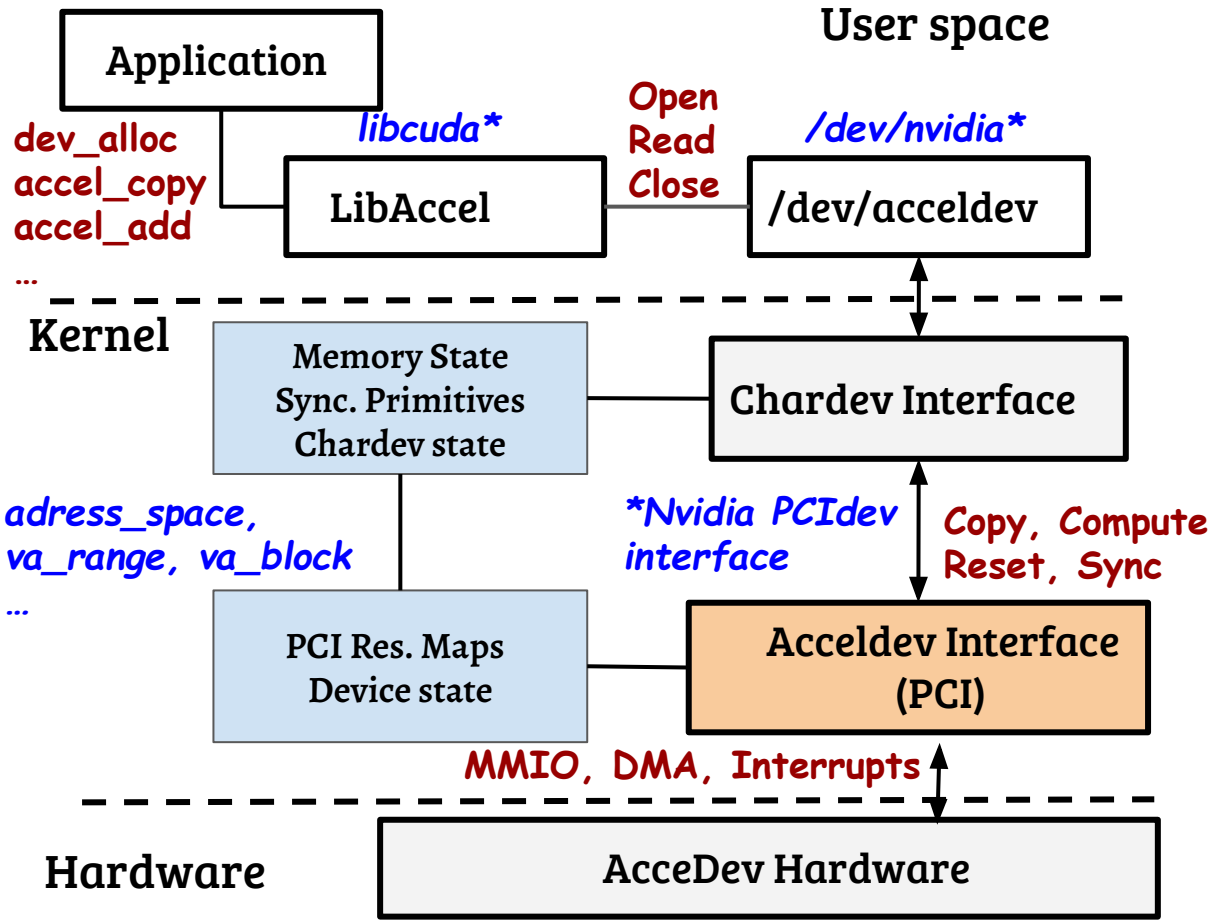
Data Copy from Device to Host

Design Overview



- Device memory management is performed by the chdev interface (by maintaining allocation meta-data)
- Data copy is performed by invoking DMA/MMIO operations through the PCI device interface
- Currently only one operation at the device layer is allowed

Design Overview - NVIDIA parallels



- NVIDIA exposes multiple chardev interfaces to the library layer for different operations (memory, compute, synchronization control)
- Device memory management is handled by the driver
  - va_range: Typically one contig. allocation
  - va_block: Typically a 2MB contig. region