For a Senior Engineer role in GPU Linux Kernel Driver development, you can expect a mix of low-level system programming, driver architecture, GPU pipeline, and performance optimization questions. Here are some key topics and questions:

1. Linux Kernel & Driver Development

* Explain the Linux device driver model.
* What is the difference between character drivers, block drivers, and PCI drivers?
* How does memory-mapped I/O (MMIO) work in GPU drivers?
* Explain the probe() and remove() functions in a PCI driver.
* How do interrupts work in Linux? How do you handle MSI/MSI-X interrupts in GPU drivers?
* Explain workqueues vs tasklets in the Linux kernel.

2. PCIe & GPU Hardware Interaction

* How does the PCIe enumeration process work in Linux?
* Explain the role of BAR (Base Address Register) in PCIe device drivers.
* How do you perform DMA operations in a PCIe-based GPU driver?
* What is an IOMMU, and how does it impact GPU memory management?
* How do you handle GPU power management in Linux?

3. GPU Memory Management

* Explain the Linux DMA API and its use in GPU drivers.
* What are TTM (Translation Table Manager) and GEM (Graphics Execution Manager)?
* How do you handle GPU buffer allocation and memory mapping?
* How do you implement shared memory between CPU and GPU?
* How does Heterogeneous Memory Management (HMM) work?

4. Synchronization & Concurrency

* Explain spinlocks, mutexes, and semaphores in kernel programming.
* How do you handle race conditions in a multi-threaded GPU driver?
* What is an RCU (Read-Copy-Update), and when would you use it?
* How do you prevent deadlocks in GPU driver code?

5. GPU Pipeline & Rendering

* Explain the GPU rendering pipeline in simple terms.
* How does command submission work in a GPU driver?
* What are ring buffers, and how are they used in GPU drivers?
* How does a GPU execute shader programs?
* What is a Fence Mechanism, and how does it ensure proper GPU-CPU synchronization?

6. Debugging & Performance Optimization

* How do you debug GPU driver crashes in Linux?
* What tools do you use to analyze PCIe performance bottlenecks?
* Explain how to use ftrace, perf, or kgdb for GPU driver debugging.
* How do you measure and reduce latency in GPU command execution?
* What optimizations can be done in PCIe bandwidth utilization for a GPU?

7. Virtualization & Multi-GPU Support

* How does SR-IOV (Single Root I/O Virtualization) work in GPUs?
* What is a VFIO (Virtual Function I/O) driver, and how does it help with GPU passthrough?
* Explain GPU context switching and how it affects performance.
* How does a Linux GPU driver handle multi-GPU synchronization?

8. Display Subsystem (Optional)

* What is the role of DRM (Direct Rendering Manager) in Linux?
* How do framebuffers work in GPU display drivers?
* Explain the difference between KMS (Kernel Mode Setting) and UMS (User Mode Setting).
* How does vsync and triple buffering work in GPU drivers?

This is a detailed guide to help you prepare for a Senior Engineer GPU Linux Kernel Driver interview. I will explain each topic thoroughly.

1. Linux Kernel & Driver Development

Q1: Explain the Linux device driver model.

The Linux device driver model provides a unified framework to handle different types of devices. It organizes drivers using:

* Bus (PCI, USB, etc.) – Detects and manages connected hardware.
* Device (GPU, storage, etc.) – Represents the actual hardware.
* Driver (software component) – Interacts with the hardware.

The sysfs virtual filesystem exposes driver details (/sys/class, /sys/bus). The udev system dynamically manages devices.

Q2: What is the difference between character drivers, block drivers, and PCI drivers?

* Character drivers: Read/write byte-by-byte (e.g., serial ports, GPUs in compute mode).
* Block drivers: Read/write in fixed-size blocks (e.g., hard drives).
* PCI drivers: Manage devices on the PCI/PCIe bus (e.g., GPUs).

Q3: How does memory-mapped I/O (MMIO) work in GPU drivers?

GPUs use MMIO to interact with registers.

* MMIO maps device memory to CPU address space.
* The driver uses ioremap() to access BAR regions of PCIe devices.
* Example:

void \_\_iomem \*gpu\_base;

gpu\_base = ioremap(pci\_resource\_start(pdev, 0), pci\_resource\_len(pdev, 0));

iowrite32(0x1, gpu\_base + REG\_CONTROL);

Q4: Explain probe() and remove() functions in a PCI driver.

These functions are called when a GPU is detected or removed.

static int gpu\_probe(struct pci\_dev \*pdev, const struct pci\_device\_id \*id) {

// Enable the device

pci\_enable\_device(pdev);

return 0;

}

static void gpu\_remove(struct pci\_dev \*pdev) {

pci\_disable\_device(pdev);

}

static struct pci\_driver gpu\_driver = {

.name = "gpu\_driver",

.id\_table = gpu\_ids,

.probe = gpu\_probe,

.remove = gpu\_remove,

};

Q5: How do interrupts work in Linux? How do you handle MSI/MSI-X interrupts?

* Interrupts signal asynchronous events from the GPU.
* MSI/MSI-X (Message Signaled Interrupts) avoid shared IRQ conflicts.
* Use request\_irq() in the driver:

static irqreturn\_t gpu\_irq\_handler(int irq, void \*dev\_id) {

printk(KERN\_INFO "GPU Interrupt received");

return IRQ\_HANDLED;

}

int irq = pci\_irq\_vector(pdev, 0);

request\_irq(irq, gpu\_irq\_handler, 0, "gpu\_driver", dev);

2. PCIe & GPU Hardware Interaction

Q6: How does the PCIe enumeration process work in Linux?

1. BIOS/UEFI detects and assigns resources to PCIe devices.
2. The Linux kernel reads PCI configuration space via pci\_bus\_read\_config\_\*.
3. Devices are added to /sys/bus/pci/devices/.
4. The driver’s probe() function gets called.

Q7: Explain the role of BAR (Base Address Register) in PCIe device drivers.

* BARs define memory-mapped regions of a PCIe device.
* The driver maps BARs using ioremap() to access GPU registers.
* Use lspci -v to see BAR assignments.

Q8: How do you perform DMA operations in a PCIe-based GPU driver?

Use DMA APIs to transfer large amounts of data efficiently.

* Allocate DMA buffers:

dma\_addr\_t dma\_handle;

void \*cpu\_buf = dma\_alloc\_coherent(dev, SIZE, &dma\_handle, GFP\_KERNEL);

* Transfer data using scatter-gather DMA.

Q9: What is an IOMMU, and how does it impact GPU memory management?

* IOMMU (Input-Output Memory Management Unit) provides virtual memory for GPUs.
* Enables secure GPU memory access and buffer sharing in virtualization.

3. GPU Memory Management

Q10: Explain the Linux DMA API and its use in GPU drivers.

* Coherent DMA (dma\_alloc\_coherent()) – Ensures cache consistency.
* Streaming DMA (dma\_map\_single()) – Needs explicit cache synchronization.

Q11: What are TTM and GEM in Linux GPU drivers?

* GEM (Graphics Execution Manager) – Simpler, used by modern drivers like i915, amdgpu.
* TTM (Translation Table Manager) – Older, supports advanced memory paging.

4. Synchronization & Concurrency

Q12: Explain spinlocks, mutexes, and semaphores in kernel programming.

* Spinlock: Fast, used for short critical sections.
* Mutex: Blocks when locked, avoids busy-waiting.
* Semaphore: Allows multiple readers, one writer.

5. GPU Pipeline & Rendering

Q13: Explain the GPU rendering pipeline.

1. Vertex Processing
2. Rasterization
3. Fragment Processing
4. Memory Writes

Q14: What are ring buffers in GPU drivers?

* Circular buffers used for command submission.
* Helps in asynchronous GPU execution.

6. Debugging & Performance Optimization

Q15: How do you debug GPU driver crashes in Linux?

* dmesg | grep -i gpu for kernel logs.
* Use ftrace to trace function calls.
* Use perf for performance profiling.

7. Virtualization & Multi-GPU Support

Q16: How does SR-IOV work in GPUs?

* Single Root I/O Virtualization (SR-IOV) creates Virtual Functions (VFs) for sharing GPU resources across VMs.

Q17: What is VFIO, and how does it help with GPU passthrough?

* VFIO (Virtual Function I/O) allows VMs to directly control GPUs via PCIe passthrough.

8. Display Subsystem (Optional)

Q18: What is the role of DRM (Direct Rendering Manager)?

* DRM (Direct Rendering Manager) handles GPU rendering in Linux.
* Interfaces with KMS (Kernel Mode Setting) for display output.