A Survey on GPU drivers

r11922213
r11922213@csie.ntu.edu.tw
National Taiwan University

r12922015
r12922015@csie.ntu.edu.tw
National Taiwan University

r12922072
r12922072@csie.ntu.edu.tw
National Taiwan University

ABSTRACT

Graphic Processing Units(GPUs) are originally designed for graphic computation. The recent evolution of GPUs has led to their widespread adaptation in both scientific computing and machine learning tasks. Central to these functionalities of GPUs are their drivers, which act as intermediaries between the hardware and the OS, conducting communications between software and the GPU. This survey provides an overview of Linux GPU driver architecture. Additionally, the survey explores case studies in the memory management of GPUs, bus communication between GPU and GPU/CPU, and paravirtualization for GPU computation.

1 INTRODUCTION

In recent years, GPUs have evolved from traditional roles of rendering graphics to important components in high-performance computing and artificial intelligence. As the functionalities of GPUs grow increasingly complex, so do the drivers that support them. The rising reliance on GPUs in various computing industries highlights the significance of GPUs, as well as their driver developments. Modern GPU drivers have to handle a variety of tasks, including GPU memory management, and memory synchronization, job scheduling.

Major GPU manufacturers — NVIDIA [19], AMD [3], and Intel [13] have developed their own driver ecosystems, each with unique features tailored to their specific hardware. For example, NVIDIA's drivers enable powerful parallel computing capability built on its CUDA architecture [18]. AMD's drivers provide interfaces for open-source parallel computing software stacks with its ROCm platform [4]. On the other hand, Intel's drivers provide efficient utility for their integrated GPU solution.

With multifarious drivers on the market, our survey will not dig deep into each of them and their unique features. Instead, this survey aims to provide a general overview of the architectural design of GPU driver software, i.e. how they interact with existing Operating System components, specifically under the Linux environment.

In addition, the survey dives into the following case studies:

- Data transferring between GPU and CPU, or GPU and GPU.
- Memory Management of GPU for AMD GPU drivers.
- Paravirtualization for GPU computation.

We explored GPU memory management on AMD's driver because it is already integrated with the Linux kernel.

2 BACKGROUND

2.1 Device Driver

A device driver (driver) is a software component that allows the operating system and a device to communicate with each other. In Linux, a driver works as a kernel module that helps the userspace to interact with the device. It usually provides the following functionalities:

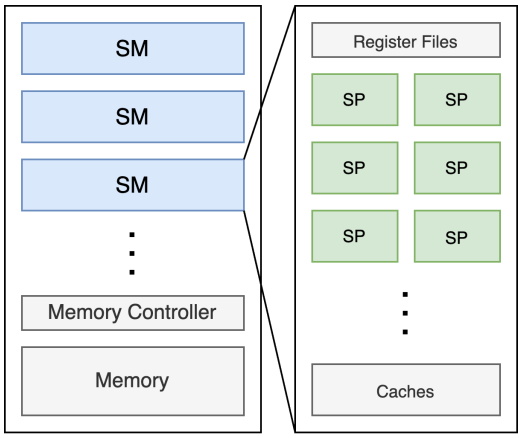

Figure 1: The layout of GPU hardware architecture

- Control the hardware.
- Register the device to the kernel, including setting up MMIO regions, ports, and DMA.
- Request interrupt request (IRQ) numbers and setup IRQ handlers.
- Provide virtual file system (VFS) interfaces, involving `open`, `read`, `write`, and `ioctl` operations.

2.2 Graphic Processing Unit

A Graphic Processing Unit (GPU) was originally used to accelerate graphics workloads. Recently, it has been used in computing tasks with high parallelism. Here, we introduce the basic overview of the GPU.

2.2.1 GPU Hardware Architecture. As shown in Figure 1, a GPU typically has the following components [17, 21]:

- Streaming Multiprocessors: A streaming multiprocessor (SM) is responsible for instruction executions. The SM consists of multiple streaming processors (SPs), register files, and caches.
- Memory System: A GPU has its own memory system, including the caches in SMs and dedicated memory, which is different from the memory for the CPU. Besides, there are memory controllers helping the SMs access the GPU's memory.

2.2.2 Data Flow between CPU and GPU. We illustrate the basic data flow between GPU and CPU in Figure 2. From the CPU's perspective, a GPU works as a device. Whenever the CPU assigns tasks to the GPU. It first transmits data and commands to the GPU, which can be achieved by writing the GPU's memory through Memory Mapped Input and Output (MMIO). After the GPU finishes its tasks, it sends

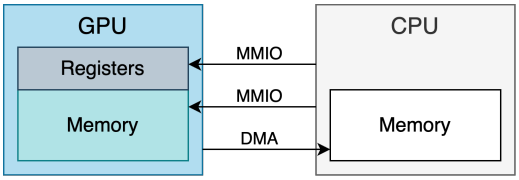

Figure 2: The data flow between GPU and CPU

the results back to the CPU. The GPU can perform writes to the CPU's memory through Direct Memory Access (DMA).

2.2.3 CPU-GPU Software Stack. Now, we introduce the software stack of GPU-related applications. Table 1 showcases the software stacks for GPU.

We suppose a user using GPU to do some computing tasks. The user usually begins with a userspace application (e.g., Pytorch [2]). The application may use userspace libraries like CUDA to access kernel drivers. These kernel drivers are designed based on the GPU hardware architecture, and they are responsible for interacting with GPU hardware. For instance, the kernel driver `nvidia.ko` helps userspace using an NVIDIA GPU.

In addition to computing tasks, users may use GPUs for graphics tasks. Compared to the computing tasks, the software stack for graphics tasks may involve different software components.

DRM. DRM [7] is a subsystem in the Linux kernel. It provides interfaces for the userspace to interact with GPUs. DRM has multiple components, enabling various functionalities regarding GPU management. For example, Graphics Execution Manager (GEM) is responsible for the memory management of GPUs. In addition, DRM contains PRIME, a cross-device buffer-sharing framework. It allows multiple GPUs to share a DMA buffer, which helps the display switch between different GPUs.

2.2.4 CPU-GPU Programming Model. In the GPU programming model, a *kernel* is the code to be executed by different threads in GPU parallelly. The GPU runs multiple threads that run the same kernel. The SMs execute these threads in parallel, speeding up the computation through parallelism.

In general, the system follows the below steps to run programs on GPU. Initially, it allocates memory on the GPU and copies the data from the host (CPU) to the GPU memory so that the GPU can execute the kernels. After the computation is done, the system copies data from GPU memory back to the host and frees up the allocated memory on the GPU. Figure 3 shows a sample code.

3 DESIGN

We present the scheme that we employed during our research in this section. The resources related to general computing GPU drivers is insufficient compared to those available for graphics computation GPU drivers. Therefore, We studied the graphics computation GPU driver as the start point. Specifically, we surveyed the structure of the codes under `linux/drivers/gpu/drm` and the corresponding user-space library. By studying the graphics computation GPU driver, we had some basic concepts about the software component of the GPU driver. For example, we know the functionality of DRM, KMS, TTM, and PRIME. Also, we were able to distinguish the

```c
#define SIZE 16
/* calculate vector addition c = a + b */
int main() {
    int size = SIZE * sizeof(float);
    float h_a[SIZE] = {...},
          h_b[SIZE] = {...},
          h_c[SIZE];
    float *d_a, *d_b, *d_c;

  /* 1. Allocate memory on GPU*/
    d_a = cudaMalloc(d_a, size);
    d_b = cudaMalloc(d_a, size);
    d_c = cudaMalloc(d_a, size);

    /* 2. Copy data from Host to GPU*/
    cudaMemcopy(d_a, h_a, size, HostToDevice);
    cudaMemcopy(d_b, h_b, size, HostToDevice);

    /* 3. Execute GPU kernel */
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c);
    cudaDeviceSynchronize();

    /* 4. Copy data from GPU back to Host */
    cudaMemcpy(h_c, d_c, DeviceToHost);

    /* 5. Free GPU memory */
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Figure 3: Sample Code for GPU Computation

component relating to general computing from that relating to graphics computing.

Next, we looked into the example code of general computing and used `strace` and `ftrace` to analyze the behavior of the code. We understood the basic programming model of GPU, which is presented at section 2. We also realized that the driver APIs are mostly exposed to the userspace through `ioctl()`. In addition, we found that the software components used in graphics computation drivers are used in general computing. The DRM subsystem is a general framework for utilizing GPU nowadays, instead of a subsystem specific for graphics computation.

With this knowledge, we finally delved into the real-world driver code and conducted research on several GPU-computing-related topics.

4 IMPLEMENTATION

We explored the source code of 1) Linux kernel AMD GPU driver, which includes AMD GPU driver (for general framework & graphics computation) and AMD GPU kernel fusion driver (for heterogeneous computing) 2) AMD ROCm library 3) NVIDIA GPU

Components	Computing	Graphics
Userspace Apps	TensorFlow [8], PyTorch [2]	Wayland [12], Mesa3D [14], X11 [5]
Userspace Libs	CUDA [18], OpenCL [9], Vulkan [11], OneAPI [6], ROCm, libnvidia-opencl	OpenGL [10], Vulkan, libdrm
Kernel Drivers	nvidia.ko, amdgpu.ko	nvidia_drm.ko, amdgpu.ko, drm.ko
Hardwares	NVIDIA, AMD, Intel GPU	NVIDIA, AMD, Intel GPU

Table 1: Software Stack for GPU. The items listed under Userspace Libs that do not start with "lib" are specifications rather than libraries. Developers need to implement libraries based on specifications.

kernel module [19] and surveyed on several topics. Our case studies mainly focused on topics related to general computing.

4.1 Data Bandwidth Bottleneck of GPU

Due to the difference of the bandwidth between PCIe buses and memory buses, the data links between GPUs and CPUs may become a performance bottleneck of GPU computing. For example, reading data from the disk and sending it to the GPU is likely to induce a performance bottleneck. Furthermore, cutting-edge AI systems usually include multiple GPUs and the training process often leverages data parallelism or model parallelism, which utilizes inter-GPU communication. We have the concern that the training may suffer from the overhead of data exchange between GPUs. We surveyed on this topic and listed some existing approaches which mitigate this problem.

4.1.1 Avoid Data Copying. The developers of the computing framework should have this issue in mind. They should ensure that the applications spend time on computing rather than data copying. But overall, it doesn't solve the core problem.

4.1.2 Improve the Bus. A straightforward solution is improving the bus bandwidth. The manufacturers have invented different technologies to boost the bus bandwidth. For example, NVIDIA and AMD have introduced NVLink and Infinity Fabric to solve this problem respectively. Multiple tech companies are proposing UALink standard to compete with NVLink.

We expected that using these technologies would require corresponding software effort. However, we only found out that the NVIDIA GPU kernel module contains code relating to NVLink, while there is no code relating to Infinity Fabric. We conclude that Infinity Fabric may be transparent to driver developers.

4.1.3 Enable device-to-device Communication. When two peripherals need to exchange data, which we refer to as *device-to-device communication*, the legacy scheme requires the peripherals to issue DMA requests separately and exchange data via the system memory, causing an additional data copy. While using GPU for general computing, several scenarios involve device-to-device communication, e.g. reading data from the disk or the internet and sending it to the GPU, transferring data between GPUs under data parallelism or model parallelism [1]. If the devices could directly communicate with each other without the assistance of the CPU and the system memory, the performance could be optimized.

There are several mechanisms in the Linux kernel to support device-to-device communication, such as *P2PDMA* [16] and *DMA-BUF* [15]. In brief, P2PDMA is for DMA transfers between devices and DMA-BUF provides the framework for sharing buffers for DMA access across multiple device drivers. Nonetheless, one could also leverage DMA-BUF to share device memory across devices and achieve device-to-device communication [20].

We studied the implementation of the AMD KFD driver. The nVME driver and the AMD GPU driver may export DMA-BUF buffer; The AMD GPU driver exports DMA-BUF buffer when a user issues an `ioctl()` call. If another GPU desires to initiate DMA requests to the exported DMA-BUF buffer, it has to call `dma_buf_attach()` first. The P2PDMA subsystem is inquired to check whether it could access the exported DMA-BUF buffer without the intervention of the CPU, that is, whether the physical peripherals support device-to-device communication. Assume the exported DMA-BUF buffer is accessible to the GPU, when the GPU driver sends DMA requests, the DMA-BUF subsystem can correctly launch device-to-device communication. The AMD KFD driver also utilizes the PRIME subsystem from DRM to manage the DMA-BUF buffer among different GPUs.

NVLink, Infinity Fabric, and UALink technologies can be used as the link between GPU cores, which further improve the data bandwidth.

4.2 Memory Management of GPU

With knowledge about kernel memory management, we would like to shed light on the memory management of the GPU driver. Specifically, we explored memory allocation and memory mapping of the AMD KFD driver. We presented some interesting details of the AMD KFD driver.

4.2.1 Memory Allocation. The flow of memory management could be roughly described in figure 4. Precisely, the AMD KFD driver initializes several data structures that record the information about the virtual memory. The functionalities of the data structures are similar to VMA in the Linux kernel. The recorded information includes 1) The type of the memory, since the corresponding physical memory could be VRAM or DMA memory 2) The reference count of the memory 3) The associated device. Note that during memory allocation, GPU is not involved. That is, memory allocation mostly relates to maintaining data structures that reside in the system memory. There are some similar designs to the Linux kernel in the DRM subsystem. For example, the data structures are linked with red-black tree and linked list, which is similar to VMA.

4.2.2 Memory Mapping. The flow of memory mapping and memory allocation are alike, except that GPU page faults could also trigger memory mapping. If the memory type is VRAM, The driver requests physical memory from the subsystem in charge of GPU physical memory. The subsystem is a part of GEM called

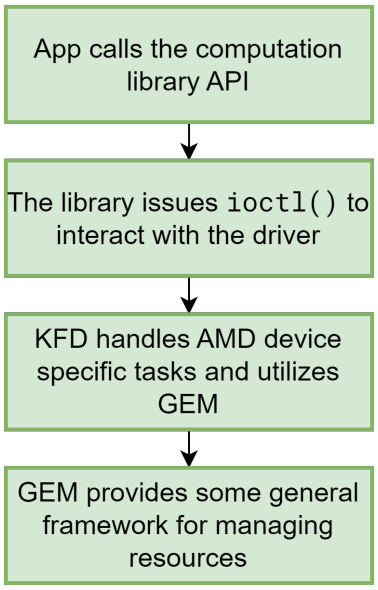

Figure 4: The Flow of Memory Allocation

`drm_buddy`, which obviously corresponds to the buddy system of the Linux kernel.

After getting the resource, the driver needs to program the GPU to map the physical memory to the GPU page tables. The driver programs the GPU by writing commands into the command buffer. The command buffer is a ring buffer consumed by the GPU, which is a common manner adopts by various devices and drivers. The AMD GPU driver supports a special scheme for issuing commands to the GPU. The driver writes a bunch of commands to a DMA-accessible memory and send the address of the commands to the command buffer. The scheme is called *Indirect Buffer*. Indirect buffer boosts the performance since it reduces the time of holding the command buffer's lock, and it resembles *Indirect Descriptors* of virtio. Finally, the driver calls the GPU scheduler to instruct the GPU to consume the commands.

4.3 Paravirtualizatoin for GPU Computation

We investigated the potential for designing a paravirtualized GPU frontend and backend interface to do GPU computation. Our discussion is based on virtio-gpu driver and device passthrough is not in the range of discussion here. Currently, virtio-gpu aims for graphics computation, indicating that the virtio-gpu driver and the virtio-gpu device don't support computing API. We conclude that it is challenging due to diverse computing specifications.

Diverse Specifications. The computing libraries, which are implemented based on the API specified by the computing specifications (OpenCL, CUDA, ROCm, OneAPI), need to be integrated with different drivers. That is, an OpenCL library that expects to work with the AMD GPU driver doesn't work with a virtio-gpu driver. Developing a virtio-gpu driver implies that we have to develop the corresponding userspace software components, which is a great amount of work. Furthermore, the most popular computing framework, CUDA, is proprietary. That is to say, it is impossible to implement the CUDA library that incorporates the virtio-gpu driver except for the developers of NVIDIA. Thus, a single virtio-gpu implementation for every computing framework is infeasible.

Moreover, not only the API but also the programming model differs. For example, OpenCL and Vulkan have no concepts about GPU-to-GPU communication, while it is a common technology that CUDA, ROCm, and OneAPI support. Likewise, the hardware abstraction also diverges, too. At the moment, there doesn't exist a unified specification for computing and GPU hardware. This is why we regard developing a practical virtio-gpu interface for computing tasks are rather challenging.

5 CONCLUSION

We present our study on GPU drivers in this report. We introduce the overview of GPU hardware and software architecture, clarifying the functionality of each component. The report intends to give insight into the GPU drivers. We also conducted case studies on several topics, including the data bandwidth bottleneck of GPU, memory management of GPU, and paravirtualization for computation. We expect that the case studies will inform the readers how the mentioned overview of GPU drivers could be applied to real-world problems and illustrate some intricate details related to system software implementation.

REFERENCES

[1] 2018. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform. *CoRR* abs/1809.02839 (2018). arXiv:1809.02839 http://arxiv.org/abs/1809.02839 Withdrawn..
[2] Meta AI. 2024. *pytorch.* https://pytorch.org/
[3] AMD. 2024. *amdgpu.* https://github.com/torvalds/linux/blob/master/drivers/gpu/drm/amd/amdgpu
[4] AMD. 2024. *ROCm.* https://github.com/ROCm
[5] Project Athena. 2024. *The X Window System.* https://www.x.org/wiki/
[6] Intel Corporation. 2024. *OneAPI.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.htm#gs.awnzjw
[7] Linux Kernel Developers. 2024. Linux DRM. https://www.kernel.org/doc/html/latest/gpu/drm-internals.html
[8] Google. 2024. *TensorFlow.* https://www.tensorflow.org
[9] Khronos Group. 2024. Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/
[10] Khronos Group. 2024. *OpenGL.* https://opengl.org/
[11] Khronos Group. 2024. *Vulkan.* https://www.vulkan.org/
[12] Kristian Høgsberg. 2024. *Wayland.* https://wayland.freedesktop.org/
[13] Intel. 2024. *i915.* https://github.com/torvalds/linux/blob/master/drivers/gpu/drm/i915
[14] VMWARE Intel. 2024. *Mesa 3D.* https://www.mesa3d.org/
[15] Linux. 2024. *Buffer Sharing and Synchronization (dma-buf).* https://docs.kernel.org/driver-api/dma-buf.html
[16] Linux. 2024. *PCI Peer-to-Peer DMA Support.* https://docs.kernel.org/driver-api/pci/p2pdma.html
[17] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks are Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Toronto Canada, 2139–2153. https://doi.org/10.1145/3243734.3243831
[18] NVIDIA. 2024. *CUDA.* https://docs.nvidia.com/cuda/doc/index.html
[19] NVIDIA. 2024. *open-gpu-kernel-modules.* https://github.com/NVIDIA/open-gpu-kernel-modules
[20] Jianxin Xiong. 2020. *RDMA With GPU Memory via DMA-BUF.* https://www.openfabrics.org/wp-content/uploads/2020-workshop-presentations/303.-OFI-GPU-DMA-BUF-OFA2020v2.pdf
[21] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Copenhagen Denmark, 960–974. https://doi.org/10.1145/3576915.3616672