简述

1. 参考英伟达开源的内核态GPU驱动NVIDIA/open-gpu-kernel-modules: NVIDIA Linux open GPU kernel module source
2. 22年5月发布，支持Turing和Ampere等较新的架构
3. 用户态驱动（OpenGL, CUDA）等仍是闭源，难以进行逆向等操作，且最终都是要通过内核态来进行显存，算力的分配
4. 该开源代码允许他人使用，修改和发布

编译模块

1. `nvidia.ko`（核心）

- **核心驱动模块**，是所有其他模块的基础，管理GPU硬件资源
- 功能
  - 提供 GPU 与内核之间的基本交互接口，如内存管理等
  - 负责设备初始化、PCIe 通信、显存管理、命令队列等核心功能
  - 为其他模块（`nvidia-drm.ko`、`nvidia-modeset.ko` 和 `nvidia-uvm.ko`）提供 API
- 首先加载，其他模块依赖于它

2. `nvidia-drm.ko`

- **Direct Rendering Manager (DRM) 接口模块**，用于支持 Linux 图形显示堆栈。
- 功能
  - 实现与 Linux DRM 子系统的集成
  - 负责 KMS（Kernel Mode Setting），用于配置显示输出设备（如分辨率和刷新率）
  - 为图形显示提供基础支持（如 Xorg、Wayland 和 Vulkan）
- 依赖 `nvidia.ko` 和 `nvidia-modeset.ko` 提供的底层功能

3. `nvidia-modeset.ko`

- **显示模式设置模块**，主要用于多显示器和显示设备的管理，与2. nvidia-drm.ko组成显示控制层
- 功能
  - 提供显示模式设置（Mode Setting）功能
  - 管理显示控制器的分辨率、刷新率、色深等配置
  - 支持动态调整屏幕输出的特性
- 依赖
  - 依赖 `nvidia.ko` 提供的设备访问能力。
  - 被 `nvidia-drm.ko` 使用来完成 DRM 层的显示管理。

4. `nvidia-uvm.ko` (重要)

- **Unified Virtual Memory (UVM)统一内存管理模块**，用于支持 GPU 和 CPU 的统一虚拟内存。
- 功能
  - 管理 GPU 与 CPU 的内存共享和一致性
  - 支持 CUDA 应用程序进行内存分页和传输操作
- 依赖依赖 `nvidia.ko` 进行设备访问和内存操作。

各模块一般分为两个部分：

- **OS - agnostic：** 与操作系统无关的部分，一般以一个二进制文件打包，用户不必每次都编译；nvidia.ko的该部分为nv-kernel.o_binary， nvidia-modeset.ko的该部分为nv-modeset-kernel.o_binary. nvidia-drm.ko和nvidia-uvm.ko都没有OS - agnostic部分
- **kernel interface layer：** 与Linux内核版本以及配置有关，只能根据用户的平台编译构建

加载顺序

`1. nvidia.ko`：

- 必须最先加载，提供底层核心功能；初始化 GPU 设备和硬件接口。

`2. nvidia-modeset.ko`：

- 在 `nvidia.ko` 加载后加载；显示模式设置

`3. nvidia-drm.ko`：

- 依赖 `nvidia.ko` 和 `nvidia-modeset.ko`。

`4. nvidia-uvm.ko`：

- 最后加载，加载后 CUDA 等高性能计算框架才能使用 GPU 的统一内存。

项目结构

```
.
├── kernel-open       --内核态驱动模块的源代码和配置文件，用于构建各个ko文件
|   ├── Kbuild        -- 构建规则
|   ├── common        --一些头文件
|   ├── conftest.sh   --检测环境
|   ├── header-presence-tests.mk
|   ├── nvidia          --nvidia.ko的代码
|   ├── nvidia-drm
|   ├── nvidia-modeset
|   ├── nvidia-peermem    --GPU间共享内存
|   └── nvidia-uvm
├── nouveau       --关联不大，支持开源Nouveau驱动的代码
|   ├── extract-firmware-nouveau.py
|   └── nouveau_firmware_layout.ods
├── nv-compiler.sh
├── src           --实现功能
|   ├── common
|   ├── nvidia
|   └── nvidia-modeset
├── utils.mk
└── version.mk

12 directories, 19 files
```

- **kernel-open** : 与linux内核交互的接口层，包含了很多linux内核的函数与头文件，<linux/pci> <linux/module> <linux/firmware>等
- **src**： 驱动功能实现
- **nouveau**： 与Nouveau开源驱动的交互代码（与GPU虚拟化基本无关）

kernel-open/nvidia

1. 硬件抽象和接口层

文件/文件夹	作用
`nv.c(核心)`	驱动的核心入口文件，包含主要的初始化、资源分配和清理逻辑，整个模块的核心
`nv-acpi.c`	实现与 ACPI（高级配置与电源接口）的交互，用于管理电源状态和设备资源
`nv-pci.c`	处理 PCI 设备的初始化、探测、资源分配和通信逻辑

文件/文件夹	作用
`nv-pci-table.c`	管理支持的 GPU PCI 设备信息表，用于设备匹配和识别
`nv-msi.c`	管理 MSI（消息信号中断）机制，优化 GPU 的中断处理性能
`nvlink_linux.c`	NVIDIA NVLink 的实现，处理 GPU 之间的高速通信接口
`os-pci.c`	OS 层面与 PCI 的交互函数，例如读取 PCI 配置空间等

2. 内存管理相关

文件/文件夹	作用
`nv-dma.c`	实现 DMA机制，用于高效的数据传输
`nv-mmap.c`	提供用户态程序访问 GPU 显存的内存映射功能
`nv-memdbg.c`	用于调试内存管理问题，例如检测内存泄漏和无效访问
`nv-usermap.c`	实现用户空间与内核空间之间的内存映射支持
`nv-vm.c`	GPU 显存管理核心代码，处理分配、释放和虚拟内存映射逻辑
`nv_uvm_interface.c`	UVM（统一虚拟内存）相关的接口代码，用于与 `nvidia-uvm` 模块交互

3. 显示管理和错误处理

文件	作用
`nv-modeset-interface.c`	提供与模式设置相关的接口，用于控制分辨率、刷新率等显示属性
`nv-modeset-interface.h`	与模式设置功能相关的头文件，定义了接口函数和结构体
`nv-report-err.c`	错误报告和记录模块，用于诊断和定位 GPU 驱动问题
`nv-report-err.h`	错误报告相关的头文件，定义了错误代码和报告机制

4. 操作系统相关适配

这些文件处理驱动与操作系统的交互。

文件	作用
`os-interface.c`	提供跨平台操作系统接口的实现，例如锁机制、延迟等系统服务调用
`os-mlock.c`	管理内存锁定操作，确保关键内存不会被交换出内存
`os-registry.c`	提供操作系统注册表或配置文件的读取和写入功能

5. 特殊硬件支持

文件	作用
`nv-ibmnpu.c`	支持 IBM NPU 的特殊功能，如内存操作和通信
`nv-ibmnpu.h`	IBM NPU 支持的头文件，定义了接口和相关数据结构
`i2c_nvswitch.c`	管理与 NVIDIA NVSwitch 的 I2C 通信接口
`ioctl_common_nvswitch.h`	NVSwitch 相关的 IOCTL 定义文件，用于用户空间与内核交互
`linux_nvswitch.c`	提供 NVSwitch 的具体实现，支持 GPU 间的高效通信

5. 安全和加密

提供加密和签名支持。

文件	作用
`libspdm_aead.c`	实现 AEAD（Authenticated Encryption with Associated Data）加密机制
`libspdm_hmac_sha.c`	HMAC-SHA 的实现，用于数据完整性校验
`libspdm_x509.c`	X.509 证书的解析和验证功能

6. 通用工具和支持

这些文件提供通用的驱动支持功能。

文件	作用
`nv-dmabuf.c`	支持 DMA-BUF 共享缓冲区机制，用于与其他驱动共享数据
`nv-cray.c`	针对 Cray 超算系统的优化和支持代码（可能涉及 GPU 在 HPC 环境中的使用）
`nvlink_caps.c`	提供 NVLink 的功能检测和管理功能

kernel-open/nvidia-uvm

1. 核心

文件	作用
`uvm.c *`	uvm模块主入口，初始化(init)和清理(exit)的核心
`uvm_gpu.c *`	gpu的有关初始化和配置等
`uvm_global.c`	全局模块的资源管理器，管理整个模块的全局状态
`uvm_gpu_access_counters.c`	GPU 的访问计数器
`uvm_gpu_isr.c`	实现 GPU 中断服务程序

2. 硬件架构支持

目录/文件	作用
`hwref/ampere(hopper/turing)`	针对不同架构的硬件寄存器定义、特性支持等
架构 `.c` 文件	`uvm_ampere.c`、`uvm_hopper.c` 等分别实现了不同架构的内存管理和故障处理逻辑

3. 内存管理

文件	作用
`uvm_va_space.c*`	管理虚拟地址空间（Virtual Address Space），协调 CPU 和 GPU 的地址分配
`uvm_va_block.c`	管理地址块（VA Block）的分配与释放，支持地址映射的细粒度管理
`uvm_va_range.c`	管理地址范围（VA Range），实现虚拟内存的分段管理
`uvm_mem.c`	实现内存分配和管理功能，包括显存、系统内存的分配
`uvm_migrate.c`	数据迁移模块，处理 GPU 与 CPU 之间的内存数据迁移
`uvm_hmm.c`	HMM（Heterogeneous Memory Management）相关功能，支持 CPU 和 GPU 的异构内存管理

4. 故障处理

文件	作用
`uvm_fault_buffer*.c`	管理 GPU 的故障缓冲区（Fault Buffer），记录内存访问故障信息。
`uvm_gpu_replayable_faults.c`	处理可重现的内存访问故障（Replayable Faults），支持错误恢复机制。
`uvm_gpu_non_replayable_faults.c`	处理不可重现的内存访问故障。
`uvm_gpu_isr.c`	中断处理逻辑，捕获和响应 GPU 故障中断。

5. 性能优化

通过事件、预测等机制优化内存访问性能

文件	作用
`uvm_perf_events.c`	管理性能事件（如数据预取、访问热点），收集和记录性能指标
`uvm_perf_heuristics.c`	性能启发式算法模块，优化数据迁移和内存访问路径
`uvm_perf_prefetch.c`	数据预取（Prefetching）模块，减少内存访问延迟

6. 设备间通信

文件	作用
`uvm_pushbuffer.c`	实现 GPU 推送缓冲区（Push Buffer），用于协调多 GPU 通信
`uvm_pmm_gpu.c`	管理 GPU 内存池（Physical Memory Manager），为设备间通信分配内存
`uvm_pte_batch.c`	批量处理页表更新，优化设备间内存访问

7. 与操作系统交互

文件	作用
`uvm_linux.c`	实现 UVM 模块在 Linux 系统上的适配代码
`uvm_ioctl.h`	定义 IOCTL 接口，用于用户态和内核态的通信
`uvm_procfs.c`	提供 `/proc` 文件系统接口，暴露调试信息给用户态

8. 测试和调试

文件	作用
`uvm_test*.c`	各种功能测试代码，如内存分配、数据迁移、故障恢复等。
`uvm_mem_test.c`	测试内存分配和管理功能。
`uvm_push_test.c`	测试 GPU 推送缓冲区相关功能。

---

kernel-open/nvidia-drm

kernl-open/nvidia-modeset

src/nvidia

```
.
├── arch
|   └── nvalloc  --不同架构的内存管理
|       ├── common
|       └── unix
├── generated
|   ├── g_bindata.c
|   ├── g_bindata_xxx.c
|   ├── g_ccsl_nvoc.c
|   |...  --大量以g开头的文件，可能是NVIDIA驱动内核相关的自动生成代码
|   ├── g_kernel_vgpu_mgr_nvoc.c
|   ├── g_kernel_vgpu_mgr_nvoc.h  --可能是与VGPU（虚拟GPU）管理相关的模块
|   ├── g_gpu_vaspace_nvoc.c / h  --处理GPU虚拟地址空间，可能与显存管理有关
```

```
|   ├── g_hypervisor_nvoc.c / h  --可能涉及与虚拟化管理器（Hypervisor）的交互
|   ├── g_mem_mgr_nvoc.c / h  --可能涉及显存分配与管理的核心实现。
|   ├── g_kern_hwpm_nvoc.c / h  --与硬件性能监控（Hardware Performance Monitor）相关
|   └── rmconfig.h
├── inc
|   ├── kernel*  --定义内核态的各种接口，结构体，常量等
|   |   ├── compute
|   |   ├── core
|   |   ├── diagnostics
|   |   ├── disp
|   |   ├── gpu
|   |   ├── gpu_mgr*
|   |   ├── gpuvideo
|   |   ├── mem_mgr*
|   |   ├── os
|   |   ├── platform
|   |   ├── power
|   |   ├── rmapi
|   |   ├── vgpu*
|   |   └── virtualization*
|   ├── lib   -- 工具函数库（base_utils.h等）
|   |   ├── base_utils.h
|   |   ├── protobuf
|   |   ├── ref_count.h  --引用计数
|   |   └── zlib  --压缩库实现
|   ├── libraries  --高层工具库（封装更高）
|   |   ├── containers
|   |   ├── crashcat
|   |   ├── eventbufferproducer.h
|   |   ├── field_desc.h
|   |   ├── ioaccess
|   |   ├── mmu
|   |   ├── nvlog
|   |   ├── nvoc
|   |   ├── nvport
|   |   ├── poolalloc.h
|   |   ├── prereq_tracker
|   |   ├── resserv
|   |   ├── tls  --线程本地存储线程本地存储
|   |   └── utils
|   └── os
|       └── dce_rm_client_ipc.h
├── interface  -- 内核态和用户态的接口
|   ├── acpidsmguids.h  --ACPI（高级配置与电源接口）相关 GUID 定义
|   ├── acpigenfuncs.h
|   ├── deprecated
|   |   ├── rmapi_deprecated.h
|   |   ├── rmapi_deprecated_allocmemory.c
|   |   ├── rmapi_deprecated_control.c
|   |   ├── rmapi_deprecated_misc.c
|   |   ├── rmapi_deprecated_utils.c
|   |   ├── rmapi_deprecated_vidheapctrl.c
|   |   └── rmapi_gss_legacy_control.c
|   ├── nv-firmware-registry.h
|   ├── nvRmReg.h
|   ├── nv_sriov_defines.h
```

```
|   ├── nv_uvm_types.h
|   ├── nvacpitypes.h
|   ├── nvrm_registry.h
|   ├── rmapi  -- 资源管理 API 源代码接口
|   |   └── src
|   └── rmp2pdefines.h
├── kernel
|   └── inc
|       └── gpuvideo  -- kernel/inc/gpuvideo 视频处理相关
├── nv-kernel.ld  --链接脚本文件，用于构建内核模块（
├── src**
|   ├── kernel  --内核态模块的核心实现,与 inc/kernel 一一对应
|   |   ├── compute
|   |   ├── core
|   |   ├── diagnostics
|   |   ├── disp
|   |   ├── gpu
|   |   ├── gpu_mgr
|   |   ├── mem_mgr
|   |   ├── os
|   |   ├── platform
|   |   ├── power
|   |   ├── rmapi
|   |   ├── vgpu
|   |   └── virtualization
|   ├── lib
|   |   ├── base_utils.c
|   |   ├── protobuf
|   |   ├── ref_count.c
|   |   └── zlib
|   └── libraries
|       ├── containers
|       ├── crashcat
|       ├── eventbuffer
|       ├── fnv_hash
|       ├── ioaccess
|       ├── libspdm
|       ├── mmu
|       ├── nvbitvector  --  位向量实现
|       ├── nvoc
|       ├── nvport
|       ├── poolalloc
|       ├── prereq_tracker
|       ├── resserv
|       ├── tls
|       └── utils
├── srcs.mk

79 directories, 462 files
```

src/nvidia和kernel-open/nvidia

`src/nvidia` **文件夹**主要负责实现驱动的库和工具，例如与内核驱动通信的接口、GPU配置、用户态 API 等。此外，还包含一些内核模块编译所需的中间代码或头文件

`kernel-open/nvidia` **文件夹**主要包含 NVIDIA 驱动的内核态代码（即运行在 Linux 内核中的部分），与 `nvidia.ko` 模块高度相关。提供对 GPU 低层硬件的访问，包括设备初始化、内存管理、任务调度、硬件中断处理等，通常更偏向底层。

- **用户态与内核态**：
  - 用户态代码（`src/nvidia`）通过系统调用或 IOCTL 接口与内核态代码（`kernel-open/nvidia`）进行通信
  - 内核态代码会将 GPU 的底层操作抽象成 API，用户态代码通过这些 API 访问 GPU 功能

相比之下，GPU 虚拟化应重点关注 `kernel-open/nvidia` 中的代码，因为它直接涉及内核中的 GPU 资源管理和调度机制。

nvidia.ko解析

入口点 kernel-open/nvidia/nv.c

```
    2
    1 module_init(nvidia_init_module);
 6248 module_exit(nvidia_exit_module);
```

使用了`module_init`和`module_exit`宏，将`nvidia_init_module`函数注册为模块的初始化函数，将`nvidia_exit_module`函数注册为模块的清理函数。

而`module_init`宏在`linux/module.h`中被定义，`__initcall`又在`linux/init.h`中被定义

```
2 extern int init_module(void);
1 extern void cleanup_module(void);
0
9 #ifndef MODULE
8 /**
7  * module_init() - driver initialization entry point
6  * @x: function to be run at kernel boot time or module insertion
5  *
4  * module_init() will either be called during do_initcalls() (if
3  * builtin) or at module insertion time (if a module).  There can only
2  * be one per module.
1  */
#define module_init(x)  __initcall(x);
```

```
#define core_initcall(fn)        __define_initcall(fn, 1)
#define core_initcall_sync(fn)       __define_initcall(fn, 1s)
#define postcore_initcall(fn)        __define_initcall(fn, 2)
#define postcore_initcall_sync(fn)   __define_initcall(fn, 2s)
#define arch_initcall(fn)        __define_initcall(fn, 3)
#define arch_initcall_sync(fn)       __define_initcall(fn, 3s)
#define subsys_initcall(fn)      __define_initcall(fn, 4)
#define subsys_initcall_sync(fn)     __define_initcall(fn, 4s)
#define fs_initcall(fn)          __define_initcall(fn, 5)
#define fs_initcall_sync(fn)         __define_initcall(fn, 5s)
#define rootfs_initcall(fn)      __define_initcall(fn, rootfs)
#define device_initcall(fn)      __define_initcall(fn, 6)
#define device_initcall_sync(fn)     __define_initcall(fn, 6s)
#define late_initcall(fn)        __define_initcall(fn, 7)
#define late_initcall_sync(fn)       __define_initcall(fn, 7s)

#define __initcall(fn) device_initcall(fn)
```

`__initcall(x)` 是内核用于注册初始化函数的宏，最终会将 `x` 注册为模块加载时调用的初始化函数。进一步追踪 `__initcall` 的定义：

```
#define __initcall(fn) device_initcall(fn)
```

`device_initcall` 用于指定初始化函数的执行优先级，模块初始化函数在模块加载时被调用，而设备初始化函数通常在设备驱动加载时调用。

`device_initcall` 会将函数注册到一个特定的段（section），由链接器处理。例如：

```
#define device_initcall(fn) __define_initcall(fn, 6)
```

`6` 是执行优先级，表明这是设备相关的初始化。

nvidia_module_init()

```
static int __init nvidia_init_module(void) // __init是宏，标志该函数为初始化函数
{
    int rc;
    NvU32 count;
    nvidia_stack_t *sp = NULL;  //栈指针
    const NvBool is_nvswitch_present = os_is_nvswitch_present();  //是否有NVSwitch
设备

    nv_memdbg_init();  // 内存调试初始化

    rc = nv_procfs_init();  //初始化 procfs文件系统，用于提供调试信息的用户接口
    if (rc < 0)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: failed to initialize procfs.\n");
        return rc;
    }

    rc = nv_caps_root_init();  //初始化 NVIDIA 驱动的能力管理模块（capabilities）
    if (rc < 0)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: failed to initialize capabilities.\n");
        goto procfs_exit;
    }
```

```
    rc = nv_caps_imex_init();  //初始化 IMEX（显存导入导出）相关模块，用于显存的共享或特定交互
    if (rc < 0)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: failed to initialize IMEX channels.\n");
        goto caps_root_exit;
    }

    rc = nv_module_init(&sp);  //调用驱动的初始化，分配和初始化一些核心数据结构，重点
    if (rc < 0)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: failed to initialize module.\n");
        goto caps_imex_exit;
    }

    count = nvos_count_devices();  // 检测和初始化设备，检查是否有NVIDIA设备
    if ((count == 0) && (!is_nvswitch_present))
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: No NVIDIA GPU found.\n");
        rc = -ENODEV;
        goto module_exit;
    }

    rc = nv_drivers_init();  //注册 PCI 驱动，将设备与驱动程序绑定。
    if (rc < 0)
    {
        goto module_exit;
    }

    if (num_probed_nv_devices != count)
    {
        ...  //报错
    }

    if ((num_probed_nv_devices == 0) && (!is_nvswitch_present))
    {
        rc = -ENODEV;
        nv_printf(NV_DBG_ERRORS, "NVRM: No NVIDIA devices probed.\n");
        goto drivers_exit;
    }

    if (num_probed_nv_devices != num_nv_devices)
    {
        nv_printf(NV_DBG_ERRORS,
            "NVRM: The NVIDIA probe routine failed for %d device(s).\n",
            num_probed_nv_devices - num_nv_devices);
    }

    if ((num_nv_devices == 0) && (!is_nvswitch_present))
    {
        rc = -ENODEV;
        nv_printf(NV_DBG_ERRORS,
            "NVRM: None of the NVIDIA devices were initialized.\n");
        goto drivers_exit;
    }
```

```
    /*
     * Initialize registry keys after PCI driver registration has
     * completed successfully to support per-device module
     * parameters.
     */
    nv_registry_keys_init(sp); //注册表键

    nv_report_applied_patches();

    nv_printf(NV_DBG_ERRORS, "NVRM: loading %s\n", pNVRM_ID);

#if defined(NV_UVM_ENABLE)  //uvm初始化
    rc = nv_uvm_init();
    if (rc != 0)
    {
        goto drivers_exit;
    }
#endif

    /*
     * Register char devices for both the region of regular devices
     * as well as the control device.
     *
     * NOTE: THIS SHOULD BE DONE LAST.
     */
   // 注册字符设备（/dev/nvidia*），用于用户空间程序与驱动交互，将接口注册到设备文件
    rc = nv_register_chrdev(0, NV_MINOR_DEVICE_NUMBER_REGULAR_MAX + 1,
            &nv_linux_devices_cdev, "nvidia", &nvidia_fops);
    if (rc < 0)
    {
        goto no_chrdev_exit;
    }
    //注册控制设备/dev/nvidiactl
    rc = nv_register_chrdev(NV_MINOR_DEVICE_NUMBER_CONTROL_DEVICE, 1,
            &nv_linux_control_device_cdev, "nvidiactl", &nvidia_fops);
    if (rc < 0)
    {
        goto partial_chrdev_exit;
    }

    __nv_init_sp = sp;

    return 0;

partial_chrdev_exit:
    nv_unregister_chrdev(0, NV_MINOR_DEVICE_NUMBER_REGULAR_MAX + 1,
        &nv_linux_devices_cdev);

no_chrdev_exit:
#if defined(NV_UVM_ENABLE)
    nv_uvm_exit();
#endif

drivers_exit:
    nv_drivers_exit();
```

```
module_exit:
    nv_module_exit(sp);

caps_imex_exit:
    nv_caps_imex_exit();

caps_root_exit:
    nv_caps_root_exit();

procfs_exit:
    nv_procfs_exit();

    return rc;
}
```

初始化模块环境： 包括内存管理、调试工具、功能模块等。

设备检测和驱动绑定： 确认系统中 NVIDIA 设备的存在，并将设备与驱动程序绑定。

注册用户接口： 通过字符设备和 `/proc` 文件系统提供用户空间的交互通道。

支持可选功能： 初始化 UVM 或其他扩展功能模块。

错误处理与资源管理： 遇到问题时，及时清理资源，确保系统的稳定性。

nv_module_init()

nv_module_init()会调用nv_module_resources_init()等，进行一系列的资源init工作

```
static int __init
nv_module_init(nv_stack_t **sp)
{
    int rc;

    rc = nv_module_resources_init(sp); //分配和初始化驱动模块所需的资源，堆栈、内存等，保存到sp
    if (rc < 0)
    {
        return rc;
    }

    rc = nv_cap_drv_init(); //初始化驱动的能力管理模块（Capabilities Driver），用于配置和管理驱动功能权限
    if (rc < 0)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: nv-cap-drv init failed.\n");
        goto cap_drv_exit;
    }

    rc = nvlink_drivers_init();
    if (rc < 0)
    {
        goto cap_drv_exit;
    }
```

```
    nv_init_rsync_info();
    nv_detect_conf_compute_platform();

    if (!rm_init_rm(*sp))  //调用资源管理器（Resource Manager，rm）的初始化函数
rm_init_rm
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: rm_init_rm() failed!\n");
        rc = -EIO;
        goto nvlink_exit;
    }

    rc = nv_module_state_init(*sp);
    if (rc < 0)
    {
        goto init_rm_exit;
    }

    return rc;

init_rm_exit:
    rm_shutdown_rm(*sp);

nvlink_exit:
    nv_destroy_rsync_info();
    nvlink_drivers_exit();

cap_drv_exit:
    nv_cap_drv_exit();
    nv_module_resources_exit(*sp);

    return rc;
}
```

文件操作接口

```
static int           nvidia_open           (struct inode *, struct file *);
static int           nvidia_close          (struct inode *, struct file *);
static unsigned int  nvidia_poll           (struct file *, poll_table *);  //事件
轮询
static int           nvidia_ioctl          (struct inode *, struct file *,
unsigned int, unsigned long);  //用户态用ioctl系统调用发来控制命令，但是似乎以及不再使用
static long          nvidia_unlocked_ioctl (struct file *, unsigned int, unsigned
long);  //现代版本的ioctl接口，比ioctl高效，避免了大锁（Big Kernel Lock）

/* character device entry points*/
static struct file_operations nvidia_fops = {  //file_operations是Linux内核结构，定
义设备的操作接口
    .owner     = THIS_MODULE,
    .poll      = nvidia_poll,   //轮询检测
    .unlocked_ioctl = nvidia_unlocked_ioctl,   //用户态发来控制命令
#if NVCPU_IS_X86_64 || NVCPU_IS_AARCH64
    .compat_ioctl = nvidia_unlocked_ioctl,   //对32位用户程序提供兼容
#endif
    .mmap      = nvidia_mmap,   //设备内存映射到用户态
```

```
    .open       = nvidia_open,    //设备打开
    .release    = nvidia_close,   //设备关闭
};
```

交互流程

- **打开设备文件:**
  调用 `open("/dev/nvidia", ...)`，触发 `nvidia_open`，完成设备上下文初始化
- **发送控制命令:**
  调用 `ioctl(fd, cmd, arg)`，进入 `nvidia_unlocked_ioctl`，解析用户命令并执行相应操作
- **状态检测:**
  调用 `poll`，进入 `nvidia_poll`，检查设备是否可用
- **关闭设备:**
  调用 `close(fd)`，触发 `nvidia_close`，释放相关资源

nvidia_open

建立用户态和GPU驱动之间的连接，当用户使用open('/dev/nvidia')时，调用该函数，该进程的打开文件表里会有/dev/nvidia的inode等结构，此后每当需要调用GPU时，可以通过ioctl来进行控制

```
/*
** nvidia_open
**
** nv driver open entry point.  Sessions are created here.
*/
int
nvidia_open(
    struct inode *inode,
    struct file *file
)
{
    nv_state_t *nv = NULL;
    nv_linux_state_t *nvl = NULL;
    int rc = 0;
    nv_linux_file_private_t *nvlfp = NULL;
    nvidia_stack_t *sp = NULL;

    nv_printf(NV_DBG_INFO, "NVRM: nvidia_open...\n");
    //分配空间和调用栈
    nvlfp = nv_alloc_file_private();
    ...
    rc = nv_kmem_cache_alloc_stack(&sp);
    ...
    NV_SET_FILE_PRIVATE(file, nvlfp); //存储打开操作的上下文
    nvlfp->sp = sp;

    /* for control device, just jump to its open routine */
    /* after setting up the private data */
    if (nv_is_control_device(inode))  //查看是否是控制设备
    {
        rc = nvidia_ctl_open(inode, file);
```

```
        if (rc != 0)
            goto failed;
        return rc;
    }

    rc = nv_down_read_interruptible(&nv_system_pm_lock);
    if (rc < 0)
        goto failed;

    /* nvptr will get set to actual nvl upon successful open */
    nvlfp->nvptr = NULL;

    init_completion(&nvlfp->open_complete);

    LOCK_NV_LINUX_DEVICES();

    nvl = find_minor_locked(NV_DEVICE_MINOR_NUMBER(inode)); //查找设备实例，minor是
次设备号
    if (nvl == NULL)
    {
        rc = -ENODEV;
        UNLOCK_NV_LINUX_DEVICES();
        up_read(&nv_system_pm_lock);
        goto failed;
    }

    nv = NV_STATE_PTR(nvl);

    if (nv_try_lock_foreground_open(file, nvl) == 0) //前台打开
    {
        /* Proceed in foreground */
        /* nvl->ldata_lock is already taken at this point */

        UNLOCK_NV_LINUX_DEVICES();

        rc = nv_open_device_for_nvlfp(nv, nvlfp->sp, nvlfp);

        up(&nvl->ldata_lock);

        /* Set nvptr only upon success (where nvl->usage_count is incremented) */
        if (rc == 0)
            nvlfp->nvptr = nvl;

        complete_all(&nvlfp->open_complete);
    }
    else
    {
        /* Defer to background kthread */
        int item_scheduled = 0;

        /*
         * Take nvl->open_q_lock in order to check nvl->is_accepting_opens and
         * schedule work items on nvl->open_q.
         *
         * Continue holding nv_linux_devices_lock (LOCK_NV_LINUX_DEVICES)
         * until the work item gets onto nvl->open_q in order to ensure the
```

```
         * lifetime of nvl.
         */
        down(&nvl->open_q_lock);

        if (!nvl->is_accepting_opens)
        {
            /* Background kthread is not accepting opens, bail! */
            rc = -EBUSY;
            goto nonblock_end;
        }

        nvlfp->deferred_open_nvl = nvl;
        nv_kthread_q_item_init(&nvlfp->open_q_item,  //开启后台线程
                               nvidia_open_deferred,
                               nvlfp);

        item_scheduled = nv_kthread_q_schedule_q_item(
                &nvl->open_q, &nvlfp->open_q_item);

        if (!item_scheduled)
        {
            WARN_ON(!item_scheduled);
            rc = -EBUSY;
        }

nonblock_end:
        up(&nvl->open_q_lock);
        UNLOCK_NV_LINUX_DEVICES();
    }

    up_read(&nv_system_pm_lock);
failed:
    if (rc != 0)
    {
        if (nvlfp != NULL)
        {
            nv_free_file_private(nvlfp);
            NV_SET_FILE_PRIVATE(file, NULL);
        }
    }
    else
    {
        nv_init_mapping_revocation(nvl, file, nvlfp, inode);
    }

    return rc;
}
```

nvidia_ioctl

ioctl是用户态程序调用GPU的核心，是NVIDIA驱动处理用户态程序发出的请求的接口，会接受并解析用户态指令，然后完成运算，显存管理等操作。例如当一个程序需要调用GPU做向量加法时，会先将向量数据传递给用户态GPU库（CUDA openGL等），进行一系列计算，当需要申请显存，申请设备时，会生成相关的任务描述符，打包数据等，接着调用ioctl进行一系列设置等

```
int
nvidia_ioctl(
    struct inode *inode,  //inode和file用于指定nvidia设备文件
    struct file *file,
    unsigned int cmd,   //操作码
    unsigned long i_arg) //命令参数
{
    NV_STATUS rmStatus;
    int status = 0;
    nv_linux_file_private_t *nvlfp = NV_GET_LINUX_FILE_PRIVATE(file);
    nv_linux_state_t *nvl;
    nv_state_t *nv;
    nvidia_stack_t *sp = NULL;
    nv_ioctl_xfer_t ioc_xfer;
    void *arg_ptr = (void *) i_arg;
    void *arg_copy = NULL;
    size_t arg_size = 0;
    int arg_cmd;
    //打印命令号，参数等
    nv_printf(NV_DBG_INFO, "NVRM: ioctl(0x%x, 0x%x, 0x%x)\n",
        _IOC_NR(cmd), (unsigned int) i_arg, _IOC_SIZE(cmd));

    if (!nv_is_control_device(inode))//如果调用/dev/nvidiactl
    {
        status = nv_wait_open_complete_interruptible(nvlfp);
        if (status != 0)
            goto done_early;
    }

    arg_size = _IOC_SIZE(cmd);
    arg_cmd  = _IOC_NR(cmd);

    if (arg_cmd == NV_ESC_IOCTL_XFER_CMD) //如果是NV_ESC_IOCTL_XFER_CMD命令
    {
        if (arg_size != sizeof(nv_ioctl_xfer_t))
        {
            nv_printf(NV_DBG_ERRORS,
                    "NVRM: invalid ioctl XFER structure size!\n");
            status = -EINVAL;
            goto done_early;
        }
        //命令通过nv_ioctl_xfer_t传输，驱动从用户态copy数据到内核，
        if (NV_COPY_FROM_USER(&ioc_xfer, arg_ptr, sizeof(ioc_xfer)))
        {
            nv_printf(NV_DBG_ERRORS,
                    "NVRM: failed to copy in ioctl XFER data!\n");
            status = -EFAULT;
            goto done_early;
```

```
        }
        //提取真正的命令号和参数大小，以及数据指针
        arg_cmd  = ioc_xfer.cmd;
        arg_size = ioc_xfer.size;
        arg_ptr  = NvP64_VALUE(ioc_xfer.ptr);

        if (arg_size > NV_ABSOLUTE_MAX_IOCTL_SIZE)
        {
            nv_printf(NV_DBG_ERRORS, "NVRM: invalid ioctl XFER size!\n");
            status = -EINVAL;
            goto done_early;
        }
    }

    NV_KMALLOC(arg_copy, arg_size); //分配内核内存，存储数据
    if (arg_copy == NULL)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: failed to allocate ioctl memory\n");
        status = -ENOMEM;
        goto done_early;
    }

    if (NV_COPY_FROM_USER(arg_copy, arg_ptr, arg_size)) //copy用户态数据
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: failed to copy in ioctl data!\n");
        status = -EFAULT;
        goto done_early;
    }

    /*
     * Handle NV_ESC_WAIT_OPEN_COMPLETE early as it is allowed to work
     * with or without nvl.
     */
    if (arg_cmd == NV_ESC_WAIT_OPEN_COMPLETE) //特殊命令，提前处理
    {
        nv_ioctl_wait_open_complete_t *params = arg_copy;
        params->rc = nvlfp->open_rc;
        params->adapterStatus = nvlfp->adapter_status;
        goto done_early;
    }

    nvl = nvlfp->nvptr;
    if (nvl == NULL)
    {
        status = -EIO;
        goto done_early;
    }

    nv = NV_STATE_PTR(nvl);

    status = nv_down_read_interruptible(&nv_system_pm_lock);
    if (status < 0)
    {
        goto done_early;
    }
```

```
    status = nv_kmem_cache_alloc_stack(&sp);  //内核缓存栈
    if (status != 0)
    {
        nv_printf(NV_DBG_ERRORS, "NVRM: Unable to allocate altstack for
ioctl\n");
        goto done_pm_unlock;
    }

    rmStatus = nv_check_gpu_state(nv); //检查gpu状态
    if (rmStatus == NV_ERR_GPU_IS_LOST)
    {
        nv_printf(NV_DBG_INFO, "NVRM: GPU is lost, skipping nvidia_ioctl\n");
        status = -EINVAL;
        goto done;
    }

    switch (arg_cmd)  //核心逻辑，开始执行指令
    {
        case NV_ESC_QUERY_DEVICE_INTR:  //查询GPU中断状态
        {
            nv_ioctl_query_device_intr *query_intr = arg_copy;

            NV_ACTUAL_DEVICE_ONLY(nv);

            if ((arg_size < sizeof(*query_intr)) ||
                (!nv->regs->map))
            {
                status = -EINVAL;
                goto done;
            }

            query_intr->intrStatus =
                *(nv->regs->map + (NV_RM_DEVICE_INTR_ADDRESS >> 2));
            query_intr->status = NV_OK;
            break;
        }

        /* pass out info about the card */
        case NV_ESC_CARD_INFO:  //获取GPU信息
        {
            size_t num_arg_devices = arg_size / sizeof(nv_ioctl_card_info_t);

            NV_CTL_DEVICE_ONLY(nv);

            status = nvidia_read_card_info(arg_copy, num_arg_devices);
            break;
        }

        case NV_ESC_ATTACH_GPUS_TO_FD:  //绑定GPU与文件描述符
        {
            size_t num_arg_gpus = arg_size / sizeof(NvU32);
            size_t i;

            NV_CTL_DEVICE_ONLY(nv);

            if (num_arg_gpus == 0 || nvlfp->num_attached_gpus != 0 ||
```

```
            arg_size % sizeof(NvU32) != 0)
        {
            status = -EINVAL;
            goto done;
        }

        NV_KMALLOC(nvlfp->attached_gpus, arg_size);
        if (nvlfp->attached_gpus == NULL)
        {
            status = -ENOMEM;
            goto done;
        }
        memcpy(nvlfp->attached_gpus, arg_copy, arg_size);
        nvlfp->num_attached_gpus = num_arg_gpus;

        for (i = 0; i < nvlfp->num_attached_gpus; i++)
        {
            if (nvlfp->attached_gpus[i] == 0)
            {
                continue;
            }

            if (nvidia_dev_get(nvlfp->attached_gpus[i], sp))
            {
                while (i--)
                {
                    if (nvlfp->attached_gpus[i] != 0)
                        nvidia_dev_put(nvlfp->attached_gpus[i], sp);
                }
                NV_KFREE(nvlfp->attached_gpus, arg_size);
                nvlfp->num_attached_gpus = 0;

                status = -EINVAL;
                break;
            }
        }

        break;
    }

    case NV_ESC_CHECK_VERSION_STR:  //检查驱动版本
    {
        NV_CTL_DEVICE_ONLY(nv);

        rmStatus = rm_perform_version_check(sp, arg_copy, arg_size);
        status = ((rmStatus == NV_OK) ? 0 : -EINVAL);
        break;
    }

    case NV_ESC_SYS_PARAMS:  //设置系统参数
    {
        nv_ioctl_sys_params_t *api = arg_copy;

        NV_CTL_DEVICE_ONLY(nv);

        if (arg_size != sizeof(nv_ioctl_sys_params_t))
```

```
            {
                status = -EINVAL;
                goto done;
            }

            /* numa_memblock_size should only be set once */
            if (nvl->numa_memblock_size == 0)
            {
                nvl->numa_memblock_size = api->memblock_size;
            }
            else
            {
                status = (nvl->numa_memblock_size == api->memblock_size) ?
                    0 : -EBUSY;
                goto done;
            }
            break;
        }

        case NV_ESC_NUMA_INFO:  //查询NUMA信息（Non-Uniform Memeory Access）
        {
            nv_ioctl_numa_info_t *api = arg_copy;
            rmStatus = NV_OK;

            NV_ACTUAL_DEVICE_ONLY(nv);

            if (arg_size != sizeof(nv_ioctl_numa_info_t))
            {
                status = -EINVAL;
                goto done;
            }

            rmStatus = rm_get_gpu_numa_info(sp, nv, api);
            if (rmStatus != NV_OK)
            {
                status = -EBUSY;
                goto done;
            }

            api->status = nv_get_numa_status(nvl);
            api->use_auto_online = nv_platform_use_auto_online(nvl);
            api->memblock_size = nv_ctl_device.numa_memblock_size;
            break;
        }

        case NV_ESC_SET_NUMA_STATUS:  //设置NUMA
        {
            nv_ioctl_set_numa_status_t *api = arg_copy;
            rmStatus = NV_OK;

            if (!NV_IS_SUSER())
            {
                status = -EACCES;
                goto done;
            }
```

```
            NV_ACTUAL_DEVICE_ONLY(nv);

            if (arg_size != sizeof(nv_ioctl_set_numa_status_t))
            {
                status = -EINVAL;
                goto done;
            }

            /*
             * The nv_linux_state_t for the device needs to be locked
             * in order to prevent additional open()/close() calls from
             * manipulating the usage count for the device while we
             * determine if NUMA state can be changed.
             */
            down(&nvl->ldata_lock);

            if (nv_get_numa_status(nvl) != api->status)
            {
                if (api->status == NV_IOCTL_NUMA_STATUS_OFFLINE_IN_PROGRESS)
                {
                    /*
                     * Only the current client should have an open file
                     * descriptor for the device, to allow safe offlining.
                     */
                    if (NV_ATOMIC_READ(nvl->usage_count) > 1)
                    {
                        status = -EBUSY;
                        goto unlock;
                    }
                    else
                    {
                        /*
                         * If this call fails, it indicates that RM
                         * is not ready to offline memory, and we should keep
                         * the current NUMA status of ONLINE.
                         */
                        rmStatus = rm_gpu_numa_offline(sp, nv);
                        if (rmStatus != NV_OK)
                        {
                            status = -EBUSY;
                            goto unlock;
                        }
                    }
                }

                status = nv_set_numa_status(nvl, api->status);
                if (status < 0)
                {
                    if (api->status == NV_IOCTL_NUMA_STATUS_OFFLINE_IN_PROGRESS)
                        (void) rm_gpu_numa_online(sp, nv);
                    goto unlock;
                }

                if (api->status == NV_IOCTL_NUMA_STATUS_ONLINE)
                {
                    rmStatus = rm_gpu_numa_online(sp, nv);
```

```
                    if (rmStatus != NV_OK)
                    {
                        status = -EBUSY;
                        goto unlock;
                    }
                }
            }

unlock:
            up(&nvl->ldata_lock);

            break;
        }

        case NV_ESC_EXPORT_TO_DMABUF_FD:  //显存导出为DMA缓冲
        {
            nv_ioctl_export_to_dma_buf_fd_t *params = arg_copy;

            if (arg_size != sizeof(nv_ioctl_export_to_dma_buf_fd_t))
            {
                status = -EINVAL;
                goto done;
            }

            NV_ACTUAL_DEVICE_ONLY(nv);

            params->status = nv_dma_buf_export(nv, params);

            break;
        }

        default:
            rmStatus = rm_ioctl(sp, nv, &nvlfp->nvfp, arg_cmd, arg_copy,
arg_size);
            status = ((rmStatus == NV_OK) ? 0 : -EINVAL);
            break;
    }

done:
    nv_kmem_cache_free_stack(sp);

done_pm_unlock:
    up_read(&nv_system_pm_lock);

done_early:
    if (arg_copy != NULL)
    {
        if (status != -EFAULT)
        {
            if (NV_COPY_TO_USER(arg_ptr, arg_copy, arg_size))
            {
                nv_printf(NV_DBG_ERRORS, "NVRM: failed to copy out ioctl
data\n");
                status = -EFAULT;
            }
        }
```

```
        NV_KFREE(arg_copy, arg_size);
    }

    return status;
}
```

总而言之ioctl主要用于

- **设备管理指令**； 如设置 NUMA 参数（`NV_ESC_SYS_PARAMS`）或查询设备信息（`NV_ESC_NUMA_INFO`）。
- **初始化和状态管理**； 包括设备文件的绑定、GPU 状态检查等。
- **低级通信和配置**；例如内存缓冲区的分配、DMA 操作的管理。

直接运算（如向量加法）并不是 `ioctl` 的职责，因为这种高层运算需要复杂的任务调度和 GPU 资源分配，涉及大量上下文切换和状态管理

GPU任务的调用路径

NVIDIA 驱动程序分为两个主要部分：

- **用户空间组件**：如 CUDA Runtime 和 openGL等，它们处理高级运算请求（如向量加法、内存管理）。
- **内核空间组件**：如 `nvidia.ko` 模块，主要负责硬件资源管理、底层指令转发和安全性。

GPU 运算任务的调用路径大致如下：

1. **用户程序调用高层 API**：
    - 程序调用 CUDA API，例如：

      ```
      cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
      vectorAdd<<<numBlocks, threadsPerBlock>>>(d_a, d_b, d_c);
      cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
      ```

    - 这些 API 被翻译成设备级命令，通常包括：
        - 内存分配、数据传输
        - 运算指令（内核函数）
        - 同步操作
2. **用户空间驱动转译命令**：
    - CUDA Driver 将这些高层命令转译为硬件可理解的指令
3. **通过设备文件传递到内核模块**：
    - 用户空间驱动调用设备文件（如 `/dev/nvidia0`），将这些指令流发送到内核驱动程序。
    - **注意**：设备文件的读写或 `ioctl` 调用可能用于传递管理命令或资源分配请求。
4. **内核模块与 GPU 通信**：
    - 内核模块（如 `nvidia.ko`）处理命令流，管理 DMA 内存缓冲区，负责最终将命令传递给 GPU。

5. **GPU 执行并返回结果**：
    - GPU 执行运算后，内核模块通知用户空间驱动，最终将结果交付用户程序。