# 概述

KMT层用来建立kernel-mode和user-mode的通信。

# 初始化

KMT的打开和关闭在<openclose.c>文件中实现。

KMT打开由函数**hsaKmtOpenKFD()**完成。

该函数先获取环境变量以初始化KMT配置，主要是debug等级和是否允许0大小frame buffer.然后打开KFD设备文件即**/dev/kfd**文件。文件打开成功后，将文件描述符赋给全局变量**kfd\_fd**。然后获取操作系统的页大小，并赋值给全局变量**PAGE\_SIZE**。

之后，该函数主要进行四个部分的初始化：获取HSA系统信息、fmm初始化、门铃初始化和perf counter初始化。下面依次说明

## 获取HSA系统信息

获取系统信息由函数**topology\_sysfs\_get\_system\_props()**完成。该函数在<topology.c>文件中定义。该函数会填充一个**HsaSystemProperties**结构体的变量，返回给调用者。该结构体包括numa node的个数，platform的名称、ID和版本。

HSA系统信息保存在文件**/sys/devices/virtual/kfd/kfd/topology/system\_properties**中。包括OEM ID，platform ID和platform version三项。

sysfs node的个数通过检查**/sys/devices/virtual/kfd/kfd/topology/nodes/**路径下文件夹的个数来确定，每个文件夹代表一个sysfs node。这里，一个CPU和一个GPU分别代表一个独立的sysfs-node。sysfs-node由3个静态变量维护。**num\_sysfs\_nodes**记录系统全部节点个数。**map\_user\_to\_sysfs\_node\_id**数组记录所有被映射的节点的下标。**map\_user\_to\_sysfs\_node\_id\_size**记录前面数组的大小。注意这里数组和数组大小仍是可被映射的节点个数，不是最终被映射的节点个数。最后通过**topology\_sysfs\_check\_node\_supported()**函数检查节点是否支持has。并将最终映射的节点个数赋值给HsaSystemProperties结构体的NumNodes成员。

检查节点是否支持hsa通过以下步骤：先读取节点的gpu id。即nodes/目录下对应节点目录里的**<gpu\_id>**文件。如果id为0，表示是cpu节点，则一定支持has，直接返回。如果是gpu节点，则继续读取对应节点目录下的**<properties>**文件，该文件记录了该gpu节点的属性信息。读取drm\_render\_minor字段。然后调用open\_drm\_render\_device()函数打开DRM(Direct Rendering Module)设备。如果打开成功，则节点支持hsa。

drm\_render\_minor值对应/dev/dri/renderDxxx文件，打开DRM设备，就是打开该文件。打开的文件描述符保存在静态数组**drm\_render\_fds**相应的下标中。注意该数组的下标与drm id并不对应，而是从128开始编号。

## 初始化fmm

fmm初始化由函数**fmm\_init\_process\_apertures(NumNodes)**完成，该函数在<fmm.c>文件中定义。其中NumNodes为前述函数获得的被映射的sysfs-node的个数。

### 初始化vm aperture

初始化fmm主要是为每个dGPU填充其对应的两个结构体变量：**gpu\_mem\_t**结构体。填充好的结构体变量保存在静态结构体数组**gpu\_mem**中。静态变量**g\_first\_gpu\_mem**会指向第一个gpu\_mem，静态变量**gpu\_mem\_count**记录了所有gpu\_mem总数。**svm\_t**结构体。填充好的该结构体变量保存在静态结构体变量**svm**中。

首先读取对应节点的properties文件的内容，填充gpu\_mem的id等基础信息。properties文件通过**topology\_sysfs\_get\_node\_props()**函数读取。该文件在2.1节中被打开读取过。放在这里读取是因为需要填充pci\_access信息。

然后通过ioctrl获得所有节点的aperture，通过每个节点的aperture。此时可以填充gpu\_mem的各aperture结构体成员。ioctrl获得的aperture包括lds、scratch和gpuvm的base和limit。然后利用aperture.gpuvm的base和limit，计算出svm.dgpu\_aperture和svm.dgpu\_alt\_aperture的base和limit. 期间还给全局静态变量**cpuvm\_aperture**赋值。这是cpu vm的aperture，align和limit采用硬编码，不与其他相关。

### 获取drm vm

这里需要打开gpu对应的drm文件，然后使用ioctrl向drm获取vm，以供kfd\_fd使用。打开drm文件的方法与2.1节中相同。

### 映射mmio

这一步由函数**map\_mmio()**完成。主要是从GPU上分配1 page的物理内存。然后把该物理内存映射到GPU。

首先利用kfd\_fd的ioctrl分配一页物理内存，该操作会得到所分配内存的句柄和偏移量。然后为分配的内存创建两个描述符。一个是**vm\_area\_t**类型，这是个分配显存的链表，记录了每次分配的显存在gpuvm中的地址和大小，并保存在aperture中。另一个是**vm\_object\_t**类型。他主要记录分配得到的句柄，用来做后面的映射。

然后将这个drm\_render\_fd文件映射到CPU的gpuvm地址处。

最后利用kfd\_fd的ioctrl将该句柄映射到GPU上。

这样map mmio就完成了。

### 初始化mmio aperture

最后填充gpu\_mem的mmio\_aperture结构体成员。mmio\_aperture的base和limit就是上一节中分配得到的gpuvm的base和大小。

### 各aperture

整理一下本节中被赋值的aperture如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 类型 | 位置 |  |
| kfd\_dev\_aperture | kfd\_process\_device\_apertures | 局部变量 |  |
| lds\_aperture | aperture\_t | 静态变量  gpu\_mem |  |
| scratch\_aperture |  |
| mmio\_aperture |  |
| gpuvm\_aperture | manageable\_aperture\_t |  |
| scratch\_physical |  |
| cpuvm\_aperture |  |
| dgpu\_aperture | 静态变量  svm |  |
| dgpu\_alt\_aperture |  |

最终用于mmio映射的是dgpu\_alt\_aperture。

# 其他

## 层次关系

OpenCL/hip

rocr->HAS -> libhsa\_

roct->Thunk -> libhsakmt.so

rock->Driver

## 项目地址

HSA:https://github.com/RadeonOpenCompute/ROCR-Runtime

Thunk:https://github.com/RadeonOpenCompute/ROCT-Thunk-Interface

## 编译

需要安装libpci：

apt-get install libpciaccess-dev

apt-get install pciutils-dev

hsa编译时会include /opt/rocm/libhsakmt/include/libhsakmt/下的文件，如果hsa与其版本不一致，则可能导致常量未定义，比如HSA\_IOLINK\_TYPE\_XGMI未定义。此时需要把/home/feifei/ROCR-Runtime/src/core/runtime/amd\_topology.cpp文件中HSA\_IOLINK\_TYPE\_XGMI相关case注释掉即可

cd src

mkdir build

cd build

cmake ..

make

## 使用

include: /home/feifei/ROCR-Runtime/src/inc/

link: /home/feifei/ROCR-Runtime/src/build/

library: hsa-runtime64

## 问题

sysfs-node

drm

fmm