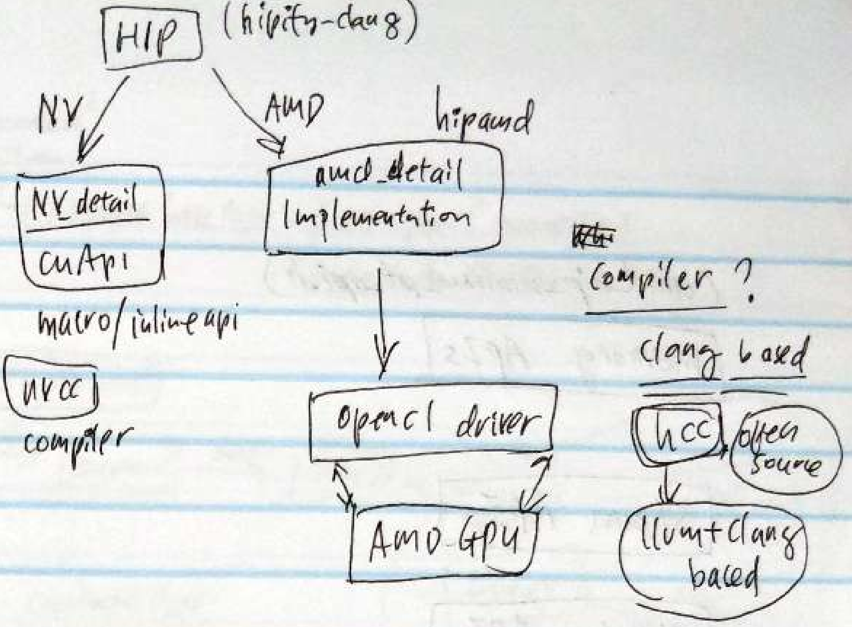
HIP

- device Context management
- memory management (Host Device)
- Kernel Execution/launch



- Command Queue
- buffer (memory)
- Kernel

Allstatic
MemObjMap

map void* ⇒ amd::Memory*.

amd::Monitor (类似 mutex)

↳ runtime/thread
runtime/platform.
runtime/os
runtime/utils

Allstatic
- .ctor
- .dtor → 不能调用

EmbeddedObject → new delete 不能调用

StackObject → new delete 不能调用

MemoryPoolObject ⇒ new delete → 不能调用 → 不做任何事

HeapObject → 提供new/delete .dtor

ReferenceCountedObject

→ 提供new/delete.
→ refcnt . . .

(amd_hip_runtime_pt_api.h)

Memory APIs

Stream APIs

event APIs

Launch APIs

stream   ihipStream_t* $\xrightleftharpoons{typedef}$ hipStream_t

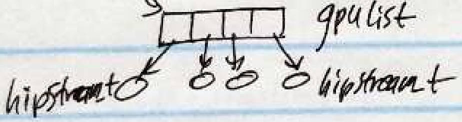
stream == Null 在 per-thread default stream 情况下，会被条件编译选择不同的 ~~out~~ 函数体

在函数体中，判断 == Null ⇒ getPerthreadDefaultStream

指针 : 0→2 ↳ hipStreamPerThread

↳ 把它转换为 hip::tls.stream_per_threa

每个线程 tls 对象 class stream_per_thread



~~hipstream~~ hip::stream
- → priority
- → ~~*~~HostQueue*
- → Device
- → flags

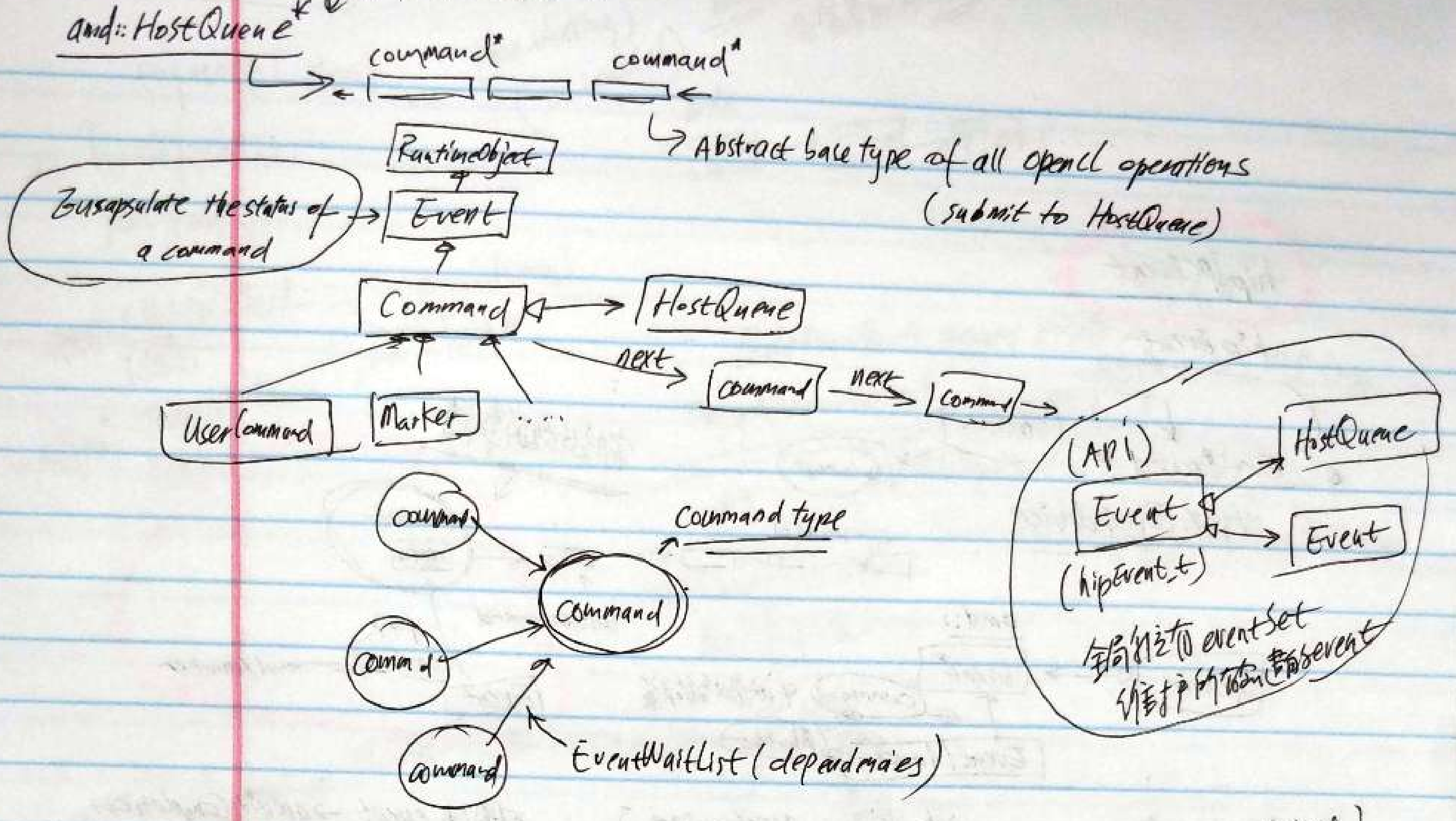
stream 的 synchronize: call HostQueue 的 finish api

ihipStream_t ⇌ hipStream_t 开销大

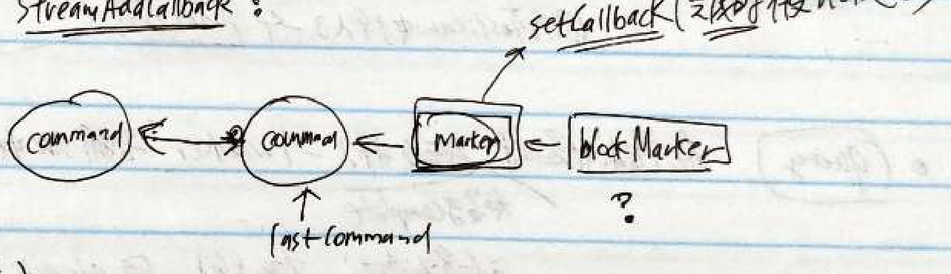
stream : Create 会对应创建个 amd::HostQueue → 会有一个独立的线程

每个 device 会有很多 Queue，添加入 Device 进行管理。

底层全局会有一个 StreamSet (保存指针)，保存所有 device 的所有 streams. 即已创建的所有streams hipstreamQuery → 获取 HostQueue.



StreamAddCallback:

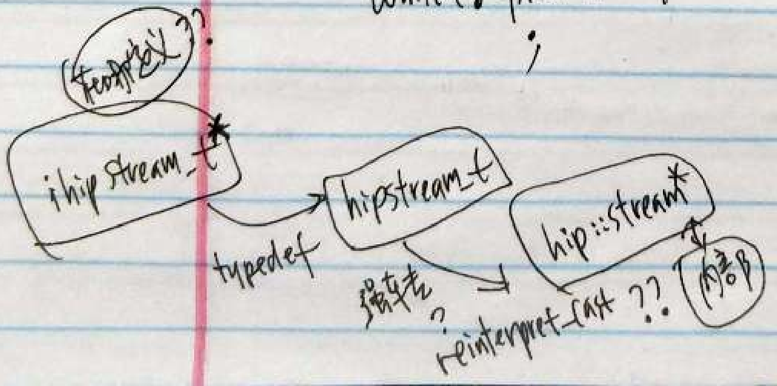


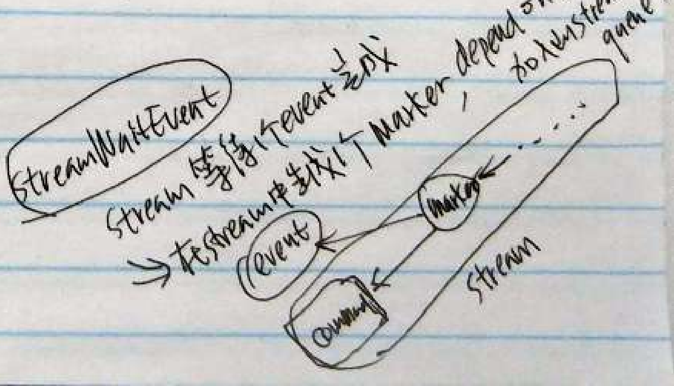
(维护)
Event 的 callback list (单链表)

atomic 类型:

AddCallback: 首先创建个 CallbackEntry*, 保存用户的 callback function/userdata.

```
entry->next = freelist;(head)
while(! freelist.compare_exchange_weak(entry->next, entry))
    ;
```

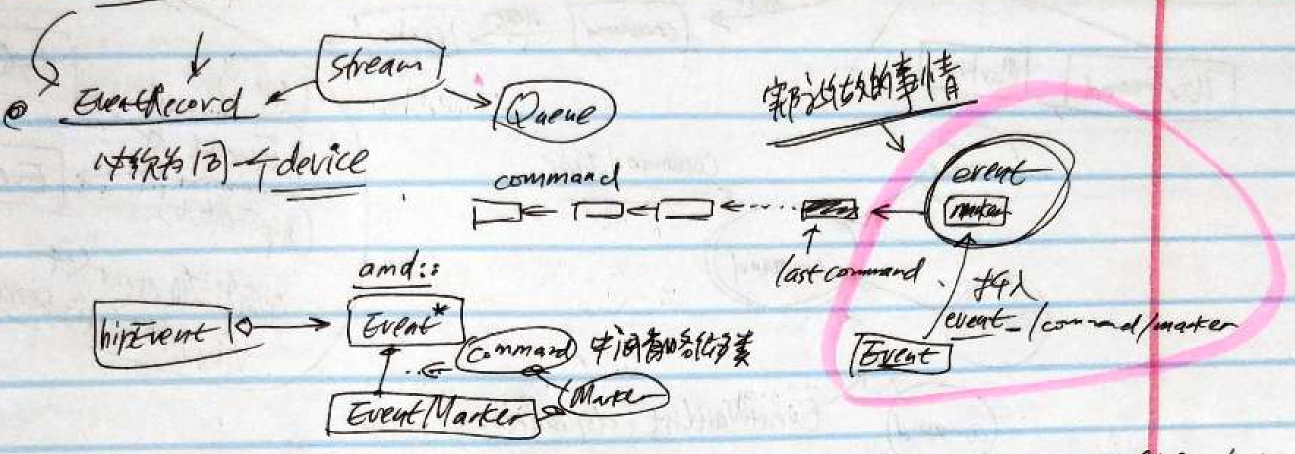




stream 等待个event完成
→ 在stream中生成个 marker, depend on 这个 event, 加入stream queue.

hipIPCEvent

hipEvent



- EventRecord 必须为同一个device

实际该做的事情

Command 中间省略很多类

- synchronize record之后就可以 synchronize了。因为在stream中插入了一个 marker

调用 event_→awaitCompletion.
↓ conditionvariable.wait (monitor) (mutex).
直到 status变成 complete.

- query 在请求状态时，如果未complete 手动插入一个marker，告诉 queue 有人在等。

之后多次请求，不再插入，用 atomic_flag !test_and_set

hipEvent中的数据成员 event_ 也是在这个时候创建。

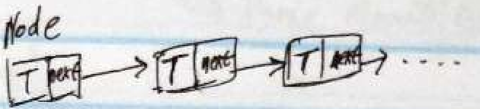
ConcurrentLinkedQueue 基于一篇论文 lockfree (non-blocking) (un-bounded)

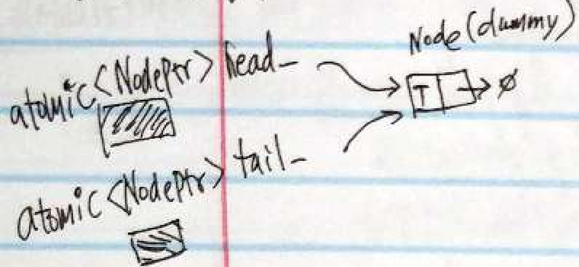
boost::lockfree 也有类似的实现

enqueue(T elem)

T dequeue();

bool empty();





N=5

NodePtr 是个 tagged pointer

低5位 加入了一些 tag

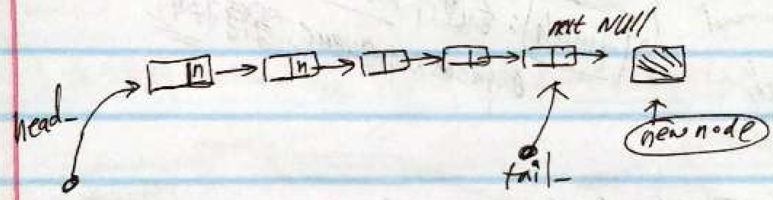
TaggedPointerHelper*

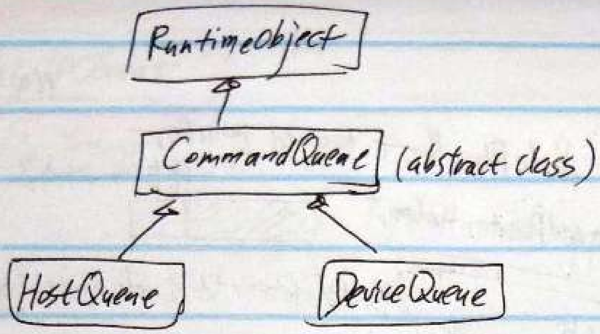
分配内存的地址是 align 到 2^5 的

所以低5位为0，可用于tag 其它东西.

2^5-1 = mask

根据 mask 可快速得到 原始指针 和 tag.





ConcurrentLinkedQueue
<command*> queue_.

有 Thread HostThread
在运行 run 执行command

flush() 触发 thread run

(stream
synchronize

finish：创建个 marker/command
enqueue 到 queue，等待执行.

Thread: pthread_create ← Thread::entry (Thread* thread)

↳ 设置 sigmask

↳ 调用 thread→main

→ (通知调用线程 我线程已启动)

→ (准备一些参数)

↳ ~~调用 run~~ 等待调用线程给它发信号 (start 那个会触发 notify 通知。

↳ run 调用. (虚函数)

start 会传入一些参数
(后期准备好).

run (void*) 接受参数.

HostQueue:: Thread
在run中的~~loop~~循环中
会创建 virtualDevice
↳ 调用 queue→loop(virtualDevice);

HostQueue.

→ 不断从 queue 中 dequeue 个 command,
然后调用 command→submit(virtualDevice).

(虚接口)

(举例子)

virtualDevice.submitKernel (*kernelcmd);

hipDeviceSynchronize → 总是创建个

- → NULL Stream queue finish 会创建marker等待所有default (flag) stream结束.
- → syncNonBlocking streams. 当前设备(device) (不是所有device)

hipDeviceReset

- → destroyAllStreams (current device)
- → Purge MemObjmap. (host pinned memory device memory)

NULL default stream per device. (legacy stream)

Device class 有1个 stream_t 实体对象. 代表NULL stream.

里面 underlying HostQueue可能为NULL.

如果我们用的是per-thread default stream.

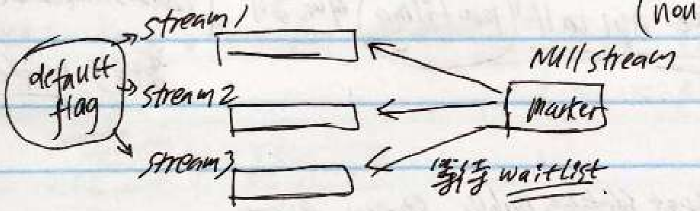
传入0 (Api), renamed-function 会转成 2,

作为 per-thread default stream.

per-thread default stream ⇒ 是一个flag = default stream, 不是non-blocking的flag.

legacy stream 在执行command之前, 会sync 其它 active stream (default flag stream only) (同一device)

(non-blocking 不受影响)



hip::Device 每个device都有一个 Memory Pool default 在create时创建, 也有一个 memory pool 列表

- 一个 list of HostQueue*.
- Device Id
- amd::Context* context; (进程独有的) shared by all devices ⇄ devices list (amd::Device*) (底层类)

hipDevice ⇄ amd::Device

hipStream_t (stream) ⇄ amd::HostQueue

Stream / HostQueue, 在销毁时都没有强制同步.

hipGetLastError

- 返回的是 tls 中的 last error
  不是 gpu device 上的 last error
  (与 cuda 不符合)??
- 拿到后重置为 success

每个 HipApi 在最开始都会 check hip runtime 是否 init.
(public)

hip_prof_api

HIP_INIT_API_INTERNAL
↳ HIP_INIT
↳ trace Api call (profiling) 可以关掉 (借助 stack 上的 object ctor/d
做事情
类似于 profilerruntime

amd hip 用了个技巧 针对所有的 public stream, event, ...
在 runtime.h: typedef ~~struct~~ hipStream_t* hipstream_t.

但是查找不到 struct hipStream_t 的定义. 这里 hipStream_t 只是一个指针.
甚至可以简单的定义为 void*.

但是在实现文件时, 用了另一个 struct 来实现 hipstream_t 的功能.

```
struct myStream {

    ....

};              t ⇒ hipstream_t*
*t ← reinterpret_cast<hipstreamt>(new mystream);
```

api 传入的 hipstream_t (其实为一个指针, 指向一个不存在的结构体)

api 处理时都将其转换为 <mystream*>(t);
mystream* s ← reinterpret_cast
再对 s 进行操作.

MemoryPool 面向stream的memory pool (全局装lock)

amd::Memory

device::Memory

- → Heap busy
- → Heap free
- → AllocateMemory ⇒ 返回 virtual memory address pointer
- → FreeMemory ⇒ 输入 amd::Memory*(+stream*)

Heap 管理
unordered map(amd::memory*,
memoryTimestamp>

API FreeMemory

- 从 busy Heap 找 amd::Memory* (map查找), 去除, 返回 memoryTimestamp
- 创建个 event/marker (到stream) associate with memoryTimestamp ??
- 将 memory 加入到 free heap 进行 track (+ timestamp)

memoryTimestamp 是个什么 ??
维护 event 和 memory object 对应的 stream

当 memory 分配时 Heap 会 track/record amd::memory* → {stream, NULLptr}
→ 哪个stream在用
(memoryTimestamp)

Free 时会更新 event.

```
Kernel1<..st1>(memory, );
FreeMemory(memory, stream1);
```

← 此时 kernel 还未完成, 但是内部会将其放入 Free Heap, 同时加入 event/marker, 监控 memory 在 stream 上的使用.

① 一旦 event 的 status 返回 ready 代表前面的 kernel 已结束, FreeHeap 可安全给其它 stream的 ~~device~~ kernel使用.

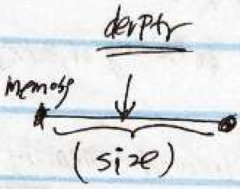
② 而且同一个stream的 Kernel 也可以直接用, 不用等待前面 kernel 完成, 因为 kernel 是顺序执行的.

memory pool 在重用 memory 时暂时 并没有进行切割, 只要大于请求的 size, 而且 stream 这时可以用, 就分配出去。(reuse)

hipFreeAsync ⟸ void* devPtr (virtual memory)
⟸ hipStream_t stream.

↓

- 根据devPtr找到 memory object (amd::Memory*)
  ↳ 所有的memory*, 全部保存在一个全局对象中 map<void*, amd::Memory*>
  map: 虚拟地址 ⟹ amd::Memory*.
  - 排序的 按地址大小
  - upper_bound 找. 第一个大于这个地址 (>)
  - ↓往前退一个 .*
  (lower_bound) 第一个不小于 (≥)



- 提供的stream是null, 默认用 Null Stream 不是 default_stream per thread
  (一个设备一个), 所以用自己创建的stream来用 memory pool
- 从设备~~对象~~中FreeMemory (loop所有的 memory pool) 也可用 perthreadstream (=2) 是否能用、
  (memory object中存有device id)
  ↓ 调用memory pool的 FreeMemory
  成功,则返回

hipMallocAsync ⟸ devPtr**
⟸ size
⟸ stream ==0 ? 用null stream (一个设备一个)
不同于 default stream per ~~thd~~ thread

↓

stream ⟹(找到) device ⟹ 用当前的memory pool ⟹ allocate memory
(memory ~~pool~~ list)

hipMemPoolCreate 是创建memory pool, 并加入 device 中的 memory pool list. 也要从device中移除.

hipMemPoolDestroy : 释放所有free的memory, 如果还有memory未Free, 因为 memory pool refcnt >1, (未delete)
(所以最好在所有memory用完后 destroy) (什么时候free呢?) ⟸ 它们还是可以工作, keep在pool中

memory pool

↳ 里面可用来给不同的stream分配内存.

hipMemallocAsync 指定 stream.

就可以从当前device的当前memory pool中分配.

(stream in device)

↑

default memory pool

或者 自己创建个, 然后 SetCurrent.

- 当前也可以显式的从特定的 memory pool (自己创建的/default) 传入 api 来分配

hipMallocFromPoolAsync (- - - memory pool, stream).

所以 hipFreeAsync 就需要从 device in memory pool list 查找删除.

---

hipCtx_t ⇌ hipDevice 全局会有 vector<hip::Device*> g_devices.

对外可见的数据结构 handle. 内部表示

hip::Device ⟷ amd::Context ⟷ 1:n amd::Device

hip_context.cpp 对 devices 进行了初始化. 每个api调用时都会check是否inited.

同时还有个 host device ⇌ amd::Context

amd::Device 类有个静态变量 vector<Device*> 代表所有 amd devices { DeviceLoad DeviceUnload

Runtime::Init → Flag::init()/option::Init() Device::init()

↳ enumerate all devices → 对于 gpu 类型 调用上述函数 (寻找设备)

每个api在启动被调用时, 都会查找 Runtime::Init 是否被调..

↑

HIP_INIT → hip::init() → Runtime::Init → 初始化 hip::Devices list (+ amd::Context)

并创建

hipInit 显式调用 也达到相同目的.

- Device::Create 会创建 default memory pool.
- 每个Device 会 associate 一个 amd::Context 指针.

hipCtxCreate 创建个新的对应device id 的hipCtx_t.

↳ 其实就是将 g_devices[gpuid] (hip::Device*) (refcnt++) ⇒ push到当前tls stack 的top. (当前线程的tls)

(系统预创建)

每个线程有个stack, 维护 context

hipCtxSetCurrent 将 ctx 作为替换当前的线程的stack的top.

为nullptr, 弹出stack的top.

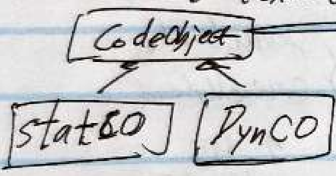
hipCtxPushCurrent 将ctx push到给当前线程的top of stack。其实就是当前线程的当前device变成这个 c

hipCtxGetDevice 返回就是当前的线程对应的device id.

---

platformStat 管理 CodeObject (fatbinary)

- map <hipModule_t, hip::DynCO*>
- hipStatCO // static code object
- texture reference



statCO 管理 modules : map<const void*, FatBinaryInfo*> ← __hipRegisterFatBinary

functions : map<const Void*, Function*> ← __hipRegisterFuncs

vars : map<const void*, Var*> ← __hipRegisterVars

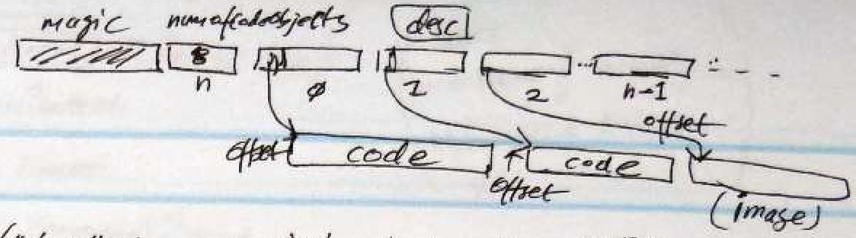
managedVars: vector<Var*> ← __hipRegisterManagedVar

functions : host function* ⇒ Function*

↳ device function code 指针??

code format

```
_clangOffloadBundleHeader {
    const char magic[SIZE];
    uint64_t numOfCodeObjects;
    _clangOffloadBundleInfo desc[1];
};
```



```
↳ {  uint64_t offset;   ⇒ 相对 code object binary 的开头
     uint64_t size;
     uint64_t bundleEntryIdSize;
     const char bundleEntryId[1];   (String)
  }
```

用这个Id[∅] + IdSize 进行loop到下一个desc block

globals :
- DeviceVar
- DeviceFunc ⇌ hipFunction_t 的内部表示. 含有 amd::Kernel*.
- Function : vector<Devicefunc*> 每个device都有. 从FatBinaryInfo中Extract出来.
- Var

hip_module.cpp 定义 如何启动 kernel hipLaunchKernel (const void* hostFunction, ... , shmBytes, stream)

__global__ void KernelFunc(...) { }

由这个 host function 指针找到 对应的 hipFunction_t 指针func

hipModule_t ⇌ ihipModule_t* (amd::Program) 依旧是CL

hipFunction_t* ⇌ struct ihipModuleSymbol_t* (不存在的结构体) (只是个指针类型)

↓ (实际对应类型)

hip::DeviceFunc::asFunction(f)

↳ hip::DeviceFunc* kernel() 返回 amd::Kernel*

(device function) 做成个command.

ihipModuleLaunchKernel(func, ... ,

其实每个 kernel 是可以单独用2个参数 startEvent/stopEvent 来 (profile 它的时间的)

platformstate.getstatFunc

↳ 从 statCO 中获取

↳ functions_map

map<void*, Function*>

!! deviceFunction 也就是从 Fatbinary中提取出来的 code object.

调用 ↳ ihipLaunchKernelCommand 生成一个 command (传入 DeviceFunc)

NDRangeKernelCommand

↳ if(startEvent) 生成1个 marker command, 放入队列.

↳ command→enqueue(); // 放入队列.

↳ if(stopEvent), track bind command event 到 StopEvent.

hip::
DeviceVar

hiModule_t ⟹ cl_program ⟹ amd::Program

(一个gpu device 以下这些信息)
(name,
amd::Memory* memobj_,
hipDeviceptr_t device_ptr_,
size_t)

(OpenCL type) External — Internal (Runtime type)

External		Internal
cl_context	⇌	amd::Context
cl_event	⇌	amd::Event
cl_command_queue	⇌	amd::CommandQueue
cl_kernel	⇌	amd::Kernel
cl_program	⇌	amd::Program
cl_device	⇌	amd::Device
cl_memory	⇌	amd::Memory
cl_sampler	⇌	amd::Sampler

```
amd::Program* program = as_amd (reinterpret_cast <cl_program>(hmod));
                            ⇓
  template <typename cl>
   typename  amd::as_internal <cl>::type*      cl_program* ⟹ amd::Program*
     as_amd (cl* cl_obj)
 { return amd::RuntimeObject::fromHandle (typename amd::as_internal <cl>::type>
                (static_cast <void*>(cl_obj));
 }
```

amd::Program map (device*, device::Program*>

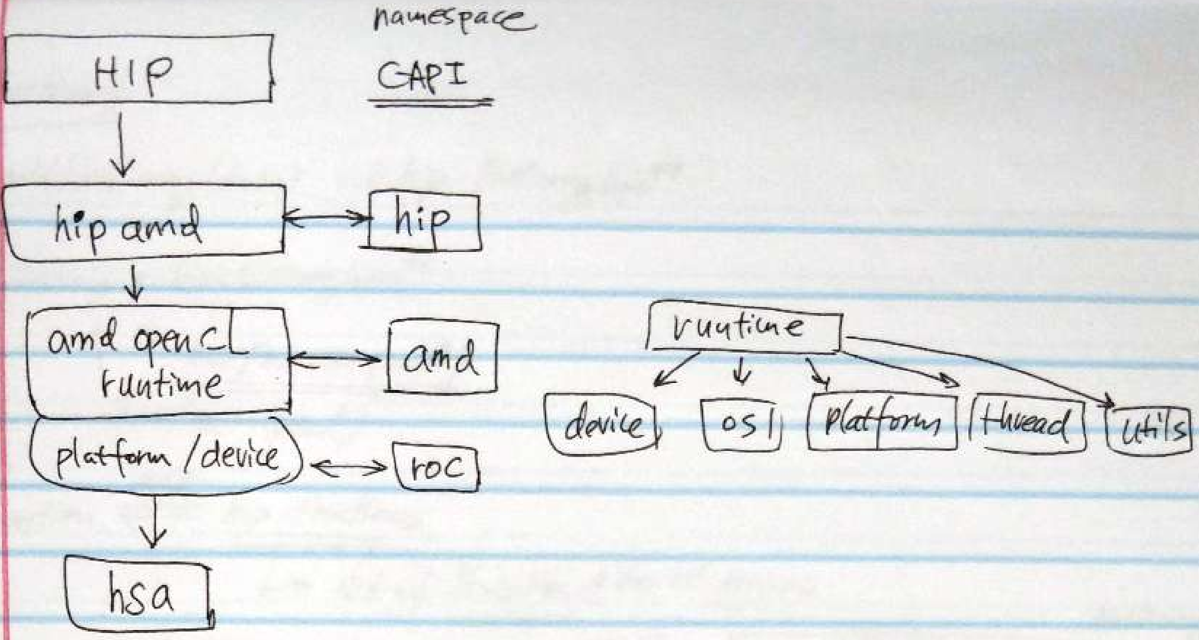
amd::Symbol map <device*, device::Kernel*>
↳ devicefunction
(kernel).

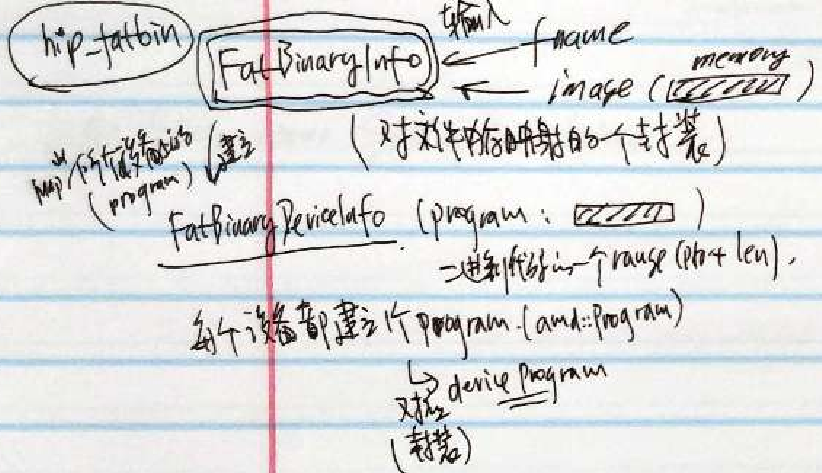
hip::
DeviceFun
(name
amd::Kernel*
对应这个symbol

抽象的概念:
Var : 一个variable在所有device上的 DeviceVar*
Function : 一个function在所有device上的 DeviceFunc*

构架

分层结构：




hip_fatbin
FatBinaryInfo
输入
fname
image (▨)
memory
(对文件内存映射的一个封装)
map 所有设备的 (program) 建立
FatBinaryDeviceInfo (program: ▨)
二进制代码的一个range (ptr+len).
每个设备都建立个program. (amd::Program)
对应 device Program
(封装)

hip-clang links device code from different translation unit together. For each device target, a code object's generated, dev1 code dev2 code ⋮ clang-offload-bunder ⟹ fatbinary ⟹ embed global symbol. in .hip_fatbin section `__hip_fatbin`. ELF

Initialization code for each translation unit for host code compilation

↳ `__hipRegisterFatBinary' ⟹ register the fatbinary embedded in the ELF file.

↳ `__hipRegisterFunction' and `__hipRegisterVar'
(kernel) (device global vars)

Termination

↳ `__hipUnregisterFatBinary'

extern "C"

__hipRegisterFatBinary

↳ addFatBinary(data) ⇒ hip::FatBinaryInfo**.

--hipRegisterFunction ← hip::FatBinaryInfo**

↑ ↖ hostFunction (void*)

deviceFunction(char*)

↓

hostFunction $\overset{map}{\rightleftarrows}$ hip::Functions

↳ list of DeviceFunc* for all devices

↳ DeviceFunc* ⇐ hipFunction_t

↳ kernel

ModuleSymbol*

FatBinarycode

hsA: heterogeneous System Architecture.