Tensorflow - 0.12

DirectSession Run {
- RunOptions
- named Tensor List <pair<string, Tensor>
- ~~vect~~ output_names
- target_nodes

vector<Tensor*>* outputs

ExecutorsAndKeys ← GetOrCreateExecutors

RunStateArgs

RunStates

CancellationManager

SendInputs

Parallel Executors

```
ExecutorBarrier* barrier = new ExecutorBarrier(
    num_executors, run_state.rendz, [&run_state](const Status& ret) {
      { mutex_lock l(run_state.mu_);
        run_state.status.Update(ret);
      }
      run_state.executors_done.Notify();
    });

Executor::Args args;
args.step_id =
args.rendezvous = run_state.rendz;
args.cancellation_manger = &step_cancellation_manager;
args.runner = [this, pool](Executor::Args::Closure c) {
    SchedClosure(pool, std::move(c));
};
  ⋮
  for (const auto& item : executors_and_keys->items) {
    item.executor->RunAsync(args, barrier->Get());
  }
```

②

Allocator

API
- Name
- AllocateRaw
- DeallocateRaw

AllocateRaw ⟷ SubAllocator

SubAllocator
- API: Alloc / Free
- Data:
  - list of visitors for alloc
  - list of visitors for free
- visitor: std::function <void (void*, int index, size

AllocatorFactorRegistry

Allocator 之注册器 ⇒ FactoryEntry list
- name
- priority
- AllocatorFactory* factory
- Allocator ☆
- SubAllocator

通过类的ctor来调用Registry的Register方法

Registry通常提供singleton的接口. 在Registration的ctor中进行调用

这样就可以提供macro来帮助一次性注册Allocator的 AllocatorFactory*.

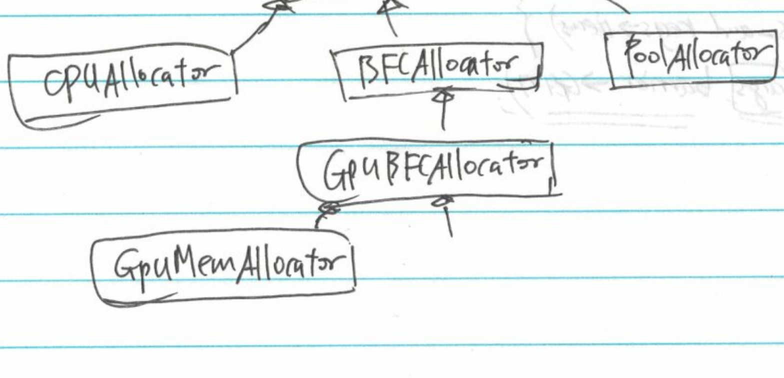
```
#define REGISTER_MEM_ALLOCATOR_UNIQ (ctr, file, line, name, priority, factor_class) \
  static AllocatorFactoryRegistration allocator_factory_reg_##ctr (
      file, line, name, priority, new factory_class)

REGISTER_MEM_ALLOCATOR (name, priority, factory_class)
```

GetAllocator 从register的list中找到一个最高优先级entry的AllocatorFactory

然后再调用 Factory -> CreateAllocator()

↳ 放入entry中的Allocator, cache起来



cpu_allocator_impl.cc

Allocator

CPUAllocator { AllocateRaw: port::AlignedMalloc
DeallocateRaw: port::AlignedFree.

AllocatorFactory

CPUAllocatorFactory { CreateAllocator: new CPUAllocat
CreateSubAllocator: new CPUSubAllocator (new CPUAllocat

REGISTER_MEM_ALLOCATOR("DefaultCPUAllocator", 100, CPUAllocatorFactory);

MKL_cpu_Allocator

Allocator

MklCpuAllocator { MklSmallSizeAllocator* ⇒ small size
BFCAllocator* ⇒ large size

BasicCPUAllocator

MklSubAllocator (use)

SubAllocator

BasicCPUAllocator { Alloc
Free

pool_allocator.h

都继承于 Allocator

Allocator

BFCAllocator

GpuBFCAllocator

GpuCudaMallocAllocator { AllocRaw :> cuMemAlloc
DeallocRaw :> cuMemFree

GpuDebugAllocator (wrap, debug)

Allocator*

head (redzone) tail

BFCAllocator

④

Tensor — Handle–body 类

Data: {
- TensorShape shape_
- TensorBuffer* buf_;

RefCounted (init: 1; {Ref, Unref})

TensorBuffer ←→ void* data_buffer

TensorShapeBase<TensorShape> → TensorShapeRep

TensorShape

⇓ 张量维数较小时，静态存储；维数较大时，用vector来动态存储大小

Device

DeviceFactory
- BaseGPUDeviceFactory
  - GPUDeviceFactory
- ThreadPoolDeviceFactory (cpu)
- SYCLDeviceFactory

{ AddDevices
  NewDevices ⇒ Device*

static ListAllPhysicalDevices

ListPhysicalDevices ⇒ API

CreateDevices

static Register(device_type, DeviceFactory* factory, ...).

static GetFactory(devicetype) ⇒ DeviceFactory*

unordered_map<string, FactoryItem>
{ unique_ptr<DeviceFactory>
  priority
  is_pluggable_device.

REGISTER_LOCAL_DEVICE_FACTORY("CPU", ThreadPoolDeviceFactory, 60);

ThreadPoolDevice

DeviceBase
- Device
  - LocalDevice
    - SYCLDevice (opencl)
    - ThreadPoolDevice (cpu)
    - BaseGPUDevice (cuda)

thread::ThreadPool
- eigen_threadpool_
  - Eigen::ThreadPoolTempl<EigenEnvironment>

EigenThreadPoolInterface*

Device {
- Compute(OpKernel*, OpKernelContext*)
  - ↳ opkernel→Compute(context);
- Sync(DoneCallback)

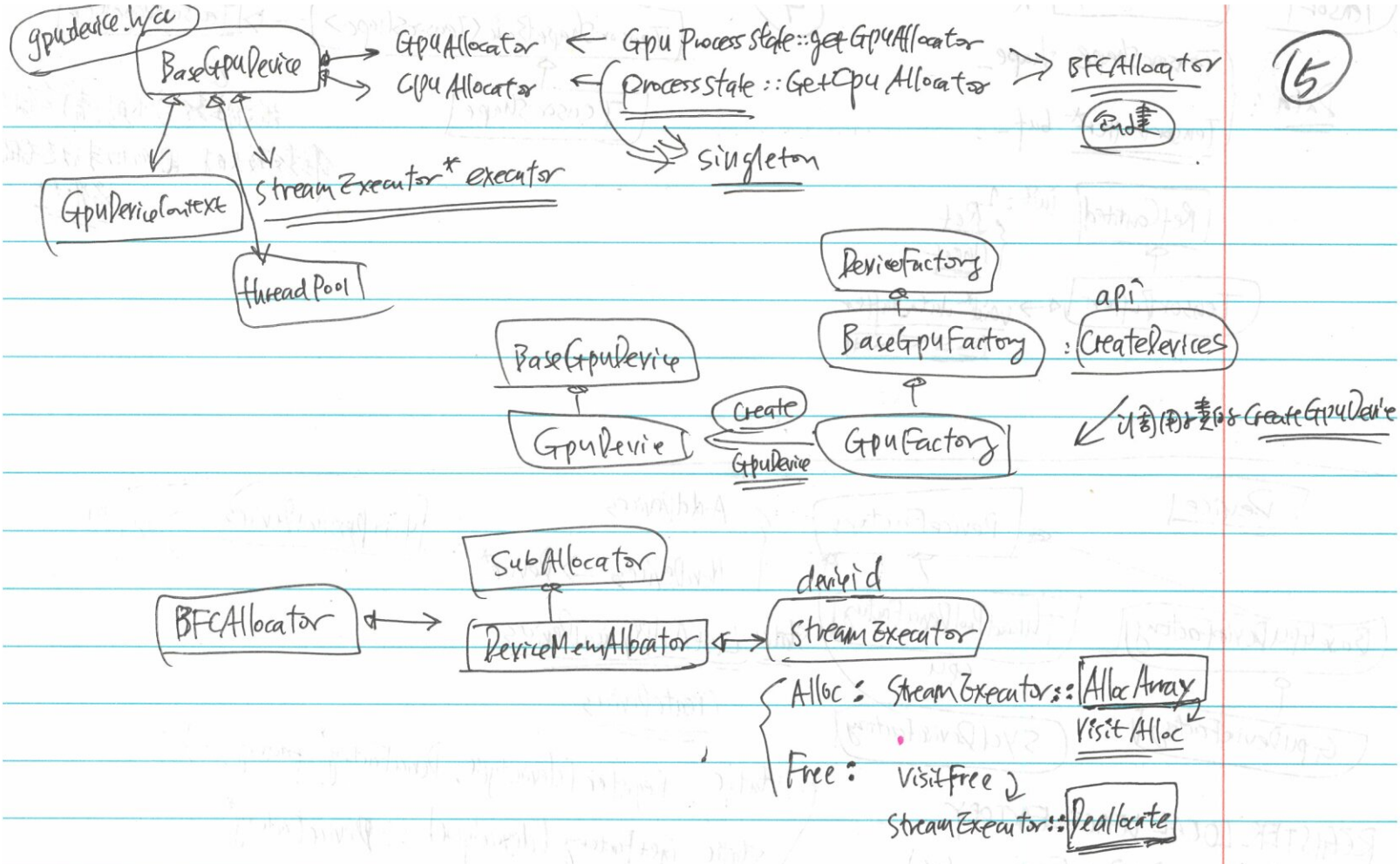
EigenEnvironment 描述一个函数家族 { 如何创建线程 / 如何创建Task / 如何运行Task.

借助 Env 的接口
{ StartThread with a function.
  CreateTask → Task
  ExecuteTask → ...

DeviceContext
- GPUDeviceContext
- SYCLDeviceContext

TaskImpl {
  std::function<void()> f;
  Context context;
  uint64 trace_id
};

task { std::unique_ptr<TaskImpl> f; }


gpudevice.h/cc
BaseGpuDevice
GpuAllocator
GpuAllocator
GpuProcessState::getGpuAllocator
ProcessState::GetCpuAllocator
BFCAllocator
創建
singleton
GpuDeviceContext
StreamExecutor* executor
ThreadPool
DeviceFactory
BaseGpuFactory
api
CreateDevices
BaseGpuDevice
GpuDevice
Create
GpuDevice
GpuFactory
以當初參數呼叫CreateGpuDevice
SubAllocator
BFCAllocator
DeviceMemAllocator
device
StreamExecutor
Alloc: StreamExecutor::AllocArray
visitAlloc
Free: visitFree
StreamExecutor::Deallocate
DeviceHostAllocator
StreamExecutor
Alloc: StreamExecutor::HostMemoryAllocate
Free: StreamExecutor::HostMemoryDeallocate
Allocator
GpuCudaMallocAllocator
AllocateRaw: cuMemAlloc
DeallocateRaw: cuMemFree
Allocator
GpuManagedAllocator
AllocateRaw: cuMemAllocManaged
DeallocateRaw: cudaFree

gpu_util


```
template <typename T>
Static se::DeviceMemory<T> AsDeviceMemory(const Tensor& t) {
  T* ptr = reinterpret_cast<T*>(const_cast<void*>(DMAHelper::base(&t)));
                                                  ↳ t.base();
  return se::DeviceMemory<T>(se::DeviceMemoryBase(ptr, t.TotalBytes()));
```

DeviceMemoryBase un-typed memory description
- void* : ptr
- size_t : size
- size_t : payload

DeviceMemory <T>

TfToPlatformDeviceIdMap Singleton object.
TfDeviceId ⟹ platformDeviceId

GpuIdManager 管理器
TfToPlatformDeviceId API

core/common_runtime
core/framework
core/platform
core/kernel
stream_executor

gcc defines little Endian:

```
constexpr bool kLittleEndian = __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__;
```

↓

0x4d2

core/platform/default

SubProcess

bool Start(); pipe/fork/execv

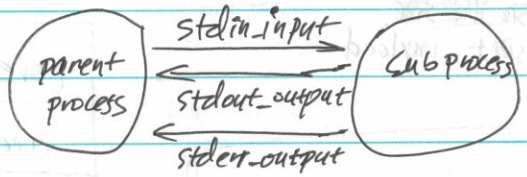
bool Kill(int signal);

bool Wait();

void SetProgram(string file, const vector<string>& argv);

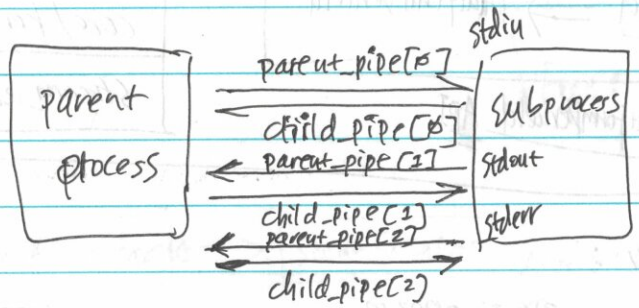
int Communicate(const string* stdin_input, string* stdout_output, string* stderr_output);

exit status



wait until subprocess exit.

Start:



fcntl(parent_pipe[i], F_SETFL, O_NONBLOCK);

fcntl(child_pipe[i], F_SETFD, FD_CLOEXEC);

父进程关闭: child_pipe[0] child_pipe[1] child_pipe[2]

子进程关闭: parent_pipe[0] parent_pipe[1] parent_pipe[2]

```
unique_ptr<SubProcess> CreateSubProcess(const vector<string>& argv) {
    unique_ptr<SubProcess> proc(new subprocess);
    proc->SetProgram(argv[0], argv);
    proc->SetChannelAction(CHAN_STDERR, ACTION_DUPPARENT);
    proc->SetChannelAction(CHAN_STDOUT, ACTION_DUPPARENT);
    return proc;
}
```

Kill(int signal) 可运行于其子进程

Wait()

StackTrace

StackTraceHandler → 在signal的打印出callstack

SIGSEGV, SIGABRT, SIGBUS, SIGILL, SIGPIPE

CurrentStackTrace

core/framework

op.h

OpRegistryInterface

virtual api: Lookup(string op_type_name, OpRegistrationData** op_reg_data)

LookupOpDef(string op_type_name, const OpDef** op_def)

not api virtual

OpListRegistry | OpRegistry

Register(const OpRegistrationDataFactory& op_data_factory)

OpRegistry::Global()->Register(

[](OpRegistrationData* op_reg_data)->Status{

//populate *op_reg_data

return Status::OK();

});

std::unordered_map<string, const OpRegistrationData*> registry_;

ArgDef[n]

input_arg → OpDef → ArgDef[n] output_arg.

KernelDef

AttrDef attr[n]

input string[n]

NodeDef

name | op | device (string)

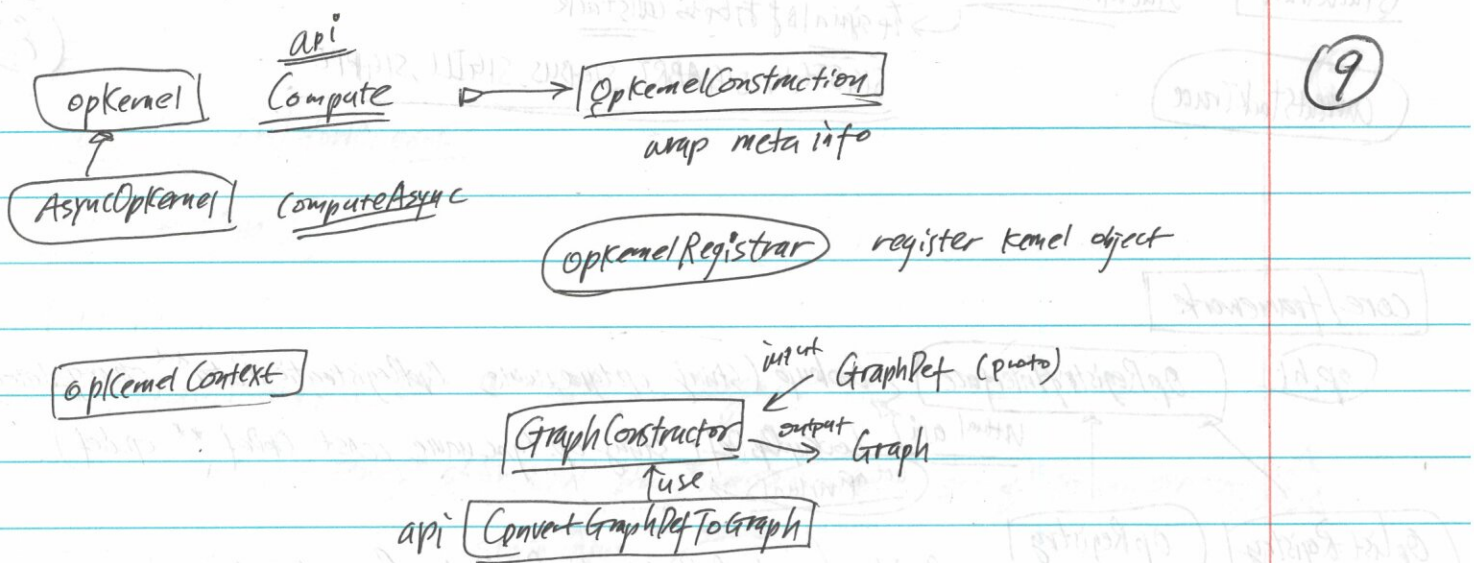
(map) attr

Graph { vector<Node*> vector<Edge*>

Node { id_ : int, class_ : NodeClass, in_edges_ : EdgeSet, out_edges : EdgeSet.

EdgeSet { array set.

Node* src → Edge → dst Node*.



```
REGISTER_OP("myop_name")
    .Attr("<name>:<type>")
    .Attr("<name>:<type>=<default>")
    .Input("<name>:<type-expr>")
    .Input("<name>:Ref(<type-expr>)")
    .Output("<name>:<type-expr>")
    .Doc(R"(
xyz
....
...
)");
```

```
static tf::InitOnStartupMarker const register_op ATTR
    TF_INIT_ON_STARTUP_IF(is_system_op || SHOULD_REGISTER_OP(name))
    << tf::register_op::OpDefBuilderWrapper(name)
```

REGISTER_OP : (name) |

```
TF_NEW_ID_FOR_INIT(REGISTER_OP_IMPL, name, false)
```

↳

```
#define TF_NEW_ID_FOR_INIT_2(m, c, ...) m(c, __VA_ARGS__)
#define TF_NEW_ID_FOR_INIT_1(m, c, ...) TF_NEW_ID_FOR_INIT_2(m, c, __VA_ARGS__)
#define TF_NEW_ID_FOR_INIT(m, ...) \
    TF_NEW_ID_FOR_INIT_1(m, __COUNTER__, __VA_ARGS__)
```

usage: M_IMPL(id, a, b) ...

M(a,b) TF_NEW_ID_FOR_INIT(~~M_IMPL~~) (M_IMPL, a, b)

```
struct InitOnStartupMarker {
  InitOnStartupMarker operator<<(InitOnStartupMarker) const {
    return *this;
  }
  template <typename T>
  InitOnStartupMarker operator<<(T&& v) const {
    return std::forward<T>(v)();
  }
};

!cond ? InitOnStartupMarker{} : (InitOnStartupMarker{} << f)

#define TF_INIT_ON_STARTUP_IF(cond) \
  (std::integral_constant<bool, !(cond)>::value) \
    ? tensorflow::InitOnStartupMarker{} \
    : tensorflow::InitOnStartupMarker{}

InitOnStartupMarker OpDefBuilderWrapper::operator()() {
  OpRegistry::Global()->Register(
      [builder = std::move(builder_)](OpRegistrationData* op_reg_data) -> Status {
        return builder.Finalize(op_reg_data);
      });
  return {};
}

OpDef op_def(proto)
OpShape(inference_fn. shape_inference_fn.
bool is_function_op = false.

REGISTER_OP_IMPL
  TF_INIT_ON_STARTUP_IF(cond) |
    << tensorflow::register_op::OpDefBuilderWrapper(name).Attr(...)
                                                          .Input(...)
                                                          .Output(...)
                                                          .Doc(...)
                                                          :
```

调用 operator()()

构造者模式

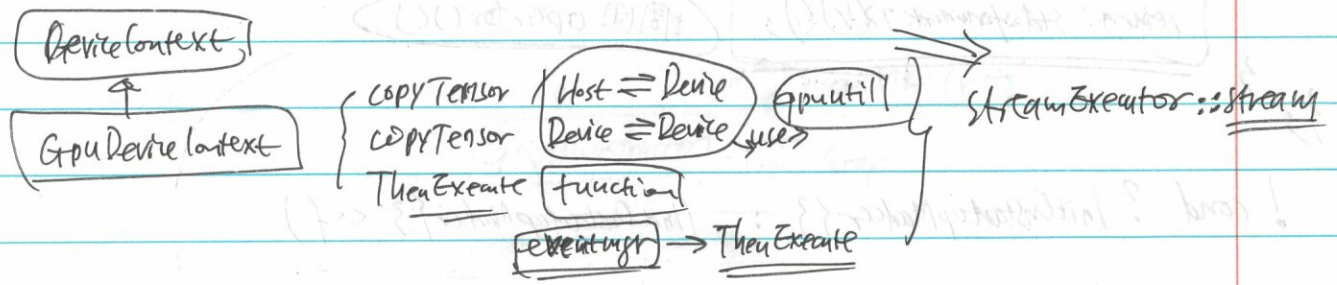
return OpDefBuilderWrapper&

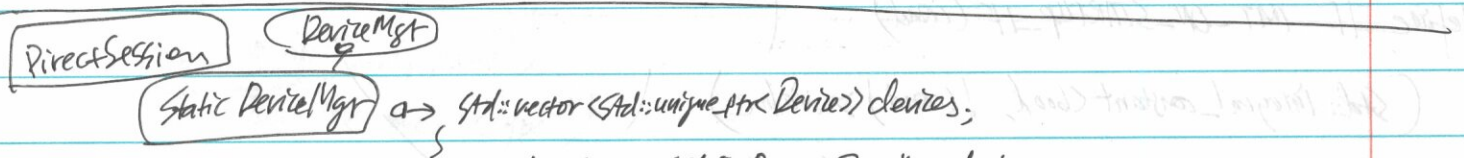
OpDefBuilder builder_

core/common_runtime/device ☑

core/common_runtime/gpu ☑

core/common_runtime/ ⇒





cuda_dnn.h/cc

stream_executor/platform/default/

```
define REGISTER_MODULE_INITIALIZER(name, body) \
  -> REGISTER_INITIALIZER(module, name, body)
                          (type)
     -> static void google_init_##type##_##name() { body; }   \ => 定义个函数 (wrap)
        stream_executor::port::Initializer google_initializer_##type##_##name(   \ 用这个函数
            google_init_##type##_##name)                                        来构造个变量
                                                 (在ctor, 直接调用函数)

REGISTER_MODULE_INITIALIZER(register_cudnn, { stream_executor::initialize_cudnn(); });
   <-
pluginRegistry::Instance()->RegisterFactory<PluginRegistry::DnnFactory>(
    cuda::kCudaPlatformId, gpu::kCuDnnPlugin, "cuDNN",
    [] (internal::StreamExecutorInterface* parent) -> dnn::DnnSupport* {
      gpu::GpuExecutor* cuda_executor = dynamic_cast<gpu::GpuExecutor*>(parent);
      gpu::CudnnSupport* dnn = new gpu::CudnnSupport(cuda_executor);
                                          ^
      if (!dnn->Init().ok()) { .... }    (per gpu)
    });

pluginRegistry::Instance()->SetDefaultFactory(
    cuda::kCudaPlatformId, PluginKind::kDnn, gpu::kCuDnnPlugin);
}
```

StreamExecutor ::implementation() 返回 StreamExecutorInterface

[StreamExecutorInterface] <- [GpuExecutor] <- (context)

Init (device_ordinal (int), DeviceOptions device_options)

GpuDriver::Init() => cuInit (call_once)

DirectSession:

RunCallable (CallableHandle handle_, const vector<Tensor>& feed_tensors,
vector<Tensor>* fetch_tensors,
RunMetadata* run_metadata,
const threadPoolOptions& thread_pool_options)

callables_ 获取对应 callable, 从中得到 executorandkeys

{

→ thread::ThreadPoolInterface* inter_op_threadpool
→ thread::ThreadPoolInterface* inter_op_threadpool

Callables_ unordered_map <int64, Callable> ← callables_lock 保护

↑ callableHandle (int64)

struct Callable {
std::shared_ptr<ExecutorsAndKeys> executors_and_keys;
std::shared_ptr<FunctionInfo> function_info;
};

unordered_map < String, std::shared_ptr <ExecutorAndKeys>> executors_, // executor_lock
→ protected by

vector <unique_ptr<FunctionInfo>> functions_;

CallFrameInterface

RunCallableCallFrame call_frame(this, executor_and_keys.get(), feed_tensors(*), fetch_tensors);

ExecutorAndKeys ← input_types
← output_types

→ RunInternal
- step_id (~~global id~~ per_session)
- run_options
- &call_frame
- executor_and_keys.get(),
- run_metadata
- threadpool_options

threadPool: core/platform/threadpool.h
core/lib/core/threadpool.h
include

Executor::Args::Runner default_runner = [pool] (Executor::Args::Closure c) {
pool->Schedule(std::move(c));
}

ThreadPool
→ threadpoolInterface* underlying_thread
= EigenThreadPool
with EigenEnvironment template argument

Executor

succeed/failure, will call DoneCallback

virtual void RunAsync(const Args& args, DoneCallback done)

→ typedef std::function<void(const Status&)> DoneCallback.

同步操作: Status Run(const Args& args) {

```
  Status ret;
  Notification n;
  RunAsync(args, [&ret, &n](const Status& s) {
    ret = s;
    n.Notify();
  });
  n.WaitForNotification();
  return ret;
}
```

Args:

```
int64 step_id = 0;
Rendezvous* rendezvous = nullptr;
StepStatsCollectorInterface* stats_collector = nullptr;
CallFrameInterface* call_frame = nullptr;
CancellationManager* cancellation_manager = nullptr;
SessionState* session_state = nullptr;
string session_handle;
TensorStore* tensor_store = nullptr;
ScopedStepContainer* step_container = nullptr;
CollectiveExecutor* collective_executor = nullptr;
typedef std::function<void()> Closure;
typedef std::function<void(Closure)> Runner;
Runner runner = nullptr;
```

ExecutorBarrier A class to help run multiple executors in parallel and wait until all of them are complete

(15)

```
typedef std::function<void(const Status&)> StatusCallback;

Data {
    Rendezvous* rendez_
    StatusCallback done_cb_
    int pending_

Api { Get => StatusCallback : std::bind(&ExecutorBarrier::WhenDone, this, std::placeholders::_1);

WhenDone <- 每个executor完成后会调用
{  {
     mutex_lock l(mu_);
     if (status_group_.ok() && !s.ok()) {
        error_rendez = rendz_;
        error_rendez->Ref();
     }
     status_group_.Update(s);
     if (--pending_ == 0) { //最后一个
        std::swap(done, done_cb_);
        Status = status_group_.as_summary_status(); //统计 status summary
     }
   }
   if (error_rendez) {
      error_rendez->StartAbort(...);
      error_rendez->Unref();
   }
   if (done) {
      delete this;   //delete 自己
      done(status);  //调用用户传入的callback.
   }
}
```

Executor

↑

ExecutorImpl

⇕

API

↳ RunAsync

↳ (new ExecutorState(args, this))

→ RunAsync(std::move(done));

DefaultExecutorRegistrar

factor中创建 Factory

} 静态注册

16

ExecutorFactory

↑

Factory

实现 NewExecutor:

↳ NewLocalExecutor

{ ExecutorFactory::Register("", factory);

ExecutorFactory::Register("DEFAULT", factory);

Data:

LocalExecutorParams params_;

unique_ptr<const Graph> graph_;

GraphView gview_;

utility function

ExecutorImpl* impl = new ExecutorImpl(params, graph);

impl → Initialize();

LocalExecutorParams

⇓ 最终线程会调用 ExecutorState::Process

处理每个节点 时调用

DirectSession 中创建 executor 时会从 Factory 中找到对应的 factory 进行创建 NewExecutor

---

ExecutorState::RunAsync

TaggedNode

TaggedNodeSeq

TaggedNodeReadyQueue

{ const Node* node;

FrameState* input_frame;

int input_iter;

bool is_dead;

⇓ typedef

gtl::InlinedVector<TaggedNode, 8>

Vector queue { push_back

pop_front

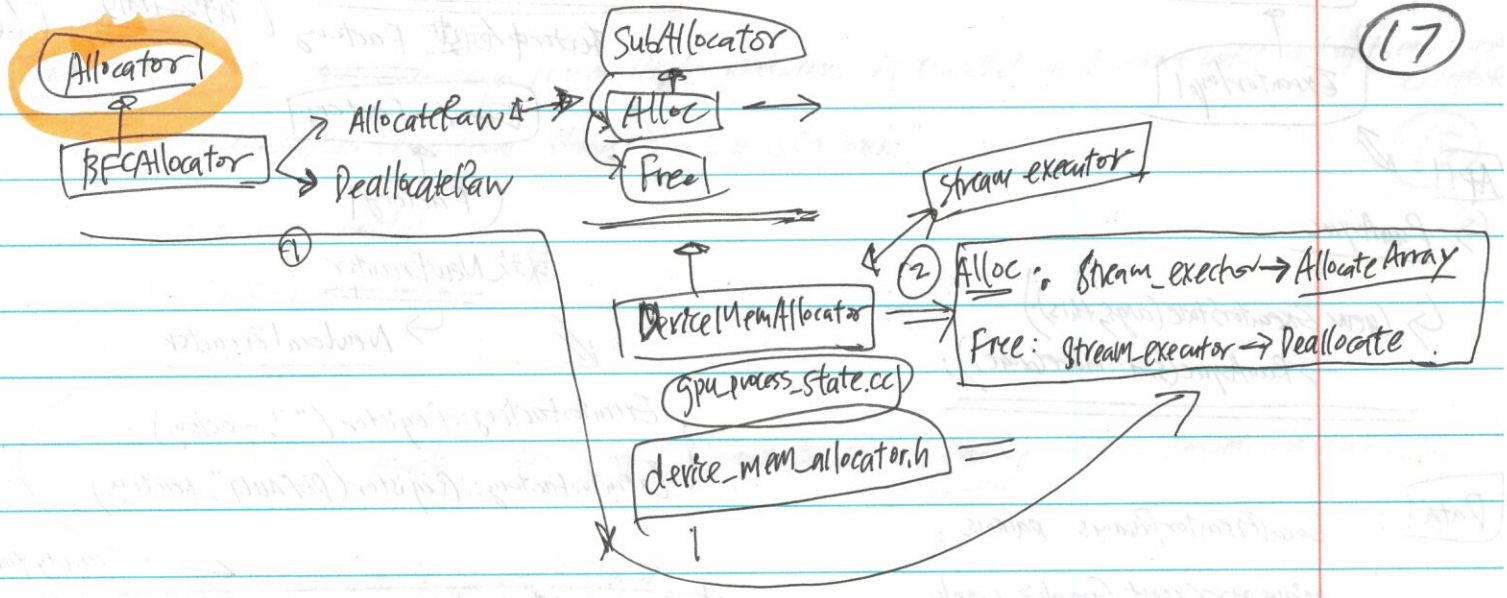
Don't free memory, just adjust front_index

gtl::InlinedVector<TaggedNode, 16> ready_;

initial size

ready.nodes ⇒ 没有 in-edges 的 Nodes

↳ ScheduleReady(ready, nullptr);



stream_executor → AllocateArray

↳ stream_executor_pimpl.h

↳ AllocateArray (stream_executor_pimpl.cc)

③ ↳ implementation → Allocate

type ↳

StreamExecutorInterface

GpuExecutor → Allocate ⑥

gpu_executor.h cuda_gpu_executor.cc

④

GpuContext

GpuDriver::DeviceAllocate(context_, size) · cuda_driver.h/cc

⑤

调用 cuMemAlloc

to的 device context

gpu allocate memory 的整个流程

Deallocate

Free

GpuDriver::DeviceDeallocate(context_, ptr)

调用 cuMemFree

Executor::RunAsync (TF-2.5)

↳ ExecutorState(T)::RunAsync

↳ scheduReady(&ready, nullptr);

↳ if (with_all_kernels_inline) : // sequentially run in singled thread (通常为false)

RunTask([this, ready=std::move(*ready), scheduled_usec]() {

for(auto& tagged_node : ready) {

process(tagged_node, scheduled_usec);

}};

else: // run in thread pool (in edge# != 0 root) (看这里)

for (auto& tagged_node : *ready) {

RunTask([=]() { process(tagged_node, scheduled_nsec); });

}

RunTask wrap. runner_([....] { closure(c); }); ← closure function. ← 运行在 thread pool 中 !!! ← 在个线程处理所有节点

alignas(64) static std::atomic<int64t> num_enqueue_ops(0);

alignas(64) static std::atomic<int64t> num_dequeue_ops(0);

⇓ cache line, avoid false-sharing.

process tagged_node (root)

↳ processSync (NodeItem item, OpKernelContext::Params* params...)

↳ OpKernelContext ctx(params, item.num_outputs);

OpKernel* op_kernel = item.kernel; ← each graph node has one kernel to associate. (when to associate ???)

Device* device = ...;

device->Compute(op_kernel, &ctx);

S = ProcessOutput(item, &ctx, outputs->data(), stats);

⇓ 在 NewLocalExecutor 时

① new executor

② call executor::Initialize

Executor_and_keys items | Vector | PerPartitionExecutorAndLib

↳ | executor |

const bool can_execute_synchronously = executor_and_keys->items.size() == 1 && call_timeout == 0;

graph不会被partition 故为true. true

if (can_execute_synchronously) {

item.executor->Run(args)

↳ Notification n;

```
RunAsync(args, [&ret, &n](...) {
    ret = s;
    n.Notify();
});
n.WaitForNotification();
```

⇒ 仍然被送入runner的 threadpool schedule

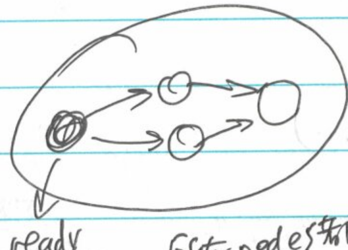
```
default_runner = [pool](closure c) {
    pool->Schedule(std::move(c));
};
```

device.h   DeviceBase / Device

```
Compute(OpKernel* op_kernel, OpKernelContext* context) {
    op_kernel->Compute(context);
}
```

↓ core/kernels/ xx (identity_op.h)

↑ gpuDevice 重新实现了'Compute'.



所有nodes都在process中处理

process 运行在 threadpool 中作为一个 task

故单线程处理所有节点.

某个OpKernel到device上运行.

对gpuDevice. 需要switch gpu context

和设置计算流 (cudastream)

Executor
↑
ExecutorImpl

api: RunAsync

Data: ImmutableExecutorState (core/common_runtime/)

ImmutableExecutorState

Data
- LocalExecutorParams params_ ← LocalExecutorParams (local_executor_params.h)
  - **Data**
    - Device*
    - const SessionMetaData*
    - FunctionLibraryRuntime*
    - function (Status(const shared_ptr<NodeProps>, OpKernel**> **create_kernel**;
    - function (void (OpKernel*)> **delete_kernel**;
    - ⇓ 在 DirectSession的 CreateExecutor中 populate
- GraphView gview_;
- **vector<const NodeItem*> root_nodes_.**

Api: Initialize (const Graph& graph).

dtor: for loop gview_.num_nodes() {
    NodeItem* item = gview_.node(i);
    if (item) {
        params_.delete_kernel(item→kernel);
    }
}

GraphView ← Graph (Initialize)

(item) NodeItem* ← Node* (n)

preprocess "all nodes" and create op kernel for each node

item ← gview_.node(n→id())

由 node id 去 index nodeItem 在graphview

NodeItem:

Data
- int node_id
- **OpKernel* kernel**
- boolean flags (lots of)
- ⋮

→ **实际上每个NodeItem分配空间时, NodeItem后面包含更多的bytes.** nodeitem | ... → 提供api在NodeItem里中进行访问 — var() api ↕ 实现 NodeItemBytes()

GraphView — array of NodeItem → placement new

NodeItem* node (int32 id) {
    (0 ≤ id < num_nodes_);
    offset = node_offsets_[id];
    (NodeItem*)(**space_ + node_offsets_[id]**);
    (=offset)

space_ char[] → [- - - ▢ - -] → NodeItem.

node_offsets_ 每个node所在spaces_的offset (字节)

FunctionInfo: simple data structure

Data → unique_ptr<FunctionLibraryDefinition> flib_def

Data → unique_ptr<ProcessFunctionLibraryRuntime> proc_flr_ → per-device

containing Data → unordered_map<Device*, std::unique_ptr<FunctionLibraryRuntime>> flr_map_

interface class

FunctionLibraryRuntime — function.h

OpRegistryInterface

FunctionLibraryDefinition

/core/framework

FunctionLibraryRuntimeImpl

Data

const DeviceMgr* const device_mgr_

Device* const device_;

Env* const env_;

const FunctionLibraryDefinition* const base_lib_def_; ← GetFunctionLibraryDefinition()

Graph::Optimizer optimizer_;

Executor::Args::Runner default_runner_; → pool->Schedule(task)

std::function<Status(const std::shared_ptr<const NodeProperties>&, OpKernel**)> create_kernels_;

flat_hash_map<Handle(uint64), std::unique_ptr<Item>> items_;

FunctionHandleCache function_handle_cache_;

ProcessFunctionLibraryRuntime → function key

(Data)

unordered_map<string, FunctionLibraryRuntime::Handle> table_;

unordered_map<FunctionLibraryRuntime::Handle, unique_ptr<FunctionData>> function_data_;

unordered_map<FunctionLibraryRuntime::Handle, unique_ptr<MultipleDeviceFunctionData>> mdevice_data_;

unordered_map<Device*, std::unique_ptr<FunctionLibraryRuntime>> flr_map_;

associate

device_mgr->ListDevices()

NewFunctionLibraryRuntime — utility function

new FunctionLibraryRuntimeImpl(....);

OpKernel ・ /core/framework/

↑

IdentityOp ・ /core/kernels/

```
IdentityOp(OpKernelConstruction* context)
  : OpKernel(context) {}

void Compute(OpKernelContext* context) {
  if (IsRefType(context->input_dtype(0))) {
    context->forward_ref_input_to_ref_output(0, 0);
  } else {
    context->set_output(0, context->input(0));
  }
}

bool IsExpensive() { return false; }
```

macro ↗ class Name ↗

REGISTER_KERNEL_BUILDER(Name("Identity").Device(DEVICE_CPU), IdentityOp);

⇓ ⇐ 通过几个 helper macro，展开以下的参数。

ctr, op_name, kernel_builder_expr, is_system_kernel, ...)

ctr = __COUNTER__ ; is_system_kernel → false ; ... ← IdentityOp (class Name)

⇓

TF_EXTRACT_KERNEL_NAME(REGISTER_KERNEL_BUILDER_IMPL_2, kernel_builder, is_system_kernel, __VA_ARGS__)

⇓ (m, kernel_builder, ...)

TF_EXTRACT_KERNEL_NAME_IMPL(m, TF_EXTRACT_KERNEL_NAME_##kernel_builder, __VA_ARGS__)

⇓ (m, ...) Name("Identity").Device(DEVICE_CPU)

m(__VA_ARGS__)

⇒ REGISTER_KERNEL_BUILDER_IMPL_2 (TF_EXTRACT_KERNEL_NAME_Name("Identity").Device(DEVICE_CPU), ...

⇓

TF_NEW_ID_FOR_INIT(REGISTER_KERNEL_BUILDER_IMPL_3, op_name, kernel_builder_expr, is_system_kernel, ...)

(name_str)

name_str, tensorflow::register_kernel::Name(name_str).Device(DEVICE_CPU)

```
namespace register_kernel {
  struct Name : public KernelDefBuilder {
    Name(const char* op) : KernelDefBuilder(op) {}
  };
}
```

↑ 创建一个临时指向 KernelDefBuilder，然后调用它的 Device(DEVICE_CPU).

REGISTER_KERNEL_BUILDER_IMPL_3 (....)

```
static tensorflow::InitOnStartupMarker const register_kernel_##ctr =
    TF_INIT_ON_STARTUP_IF (cond) <<
    [] (tensorflow::KernelDef const* kernel_def) {
        tensorflow::kernel_factory::OpKernelRegistrar registrar(
            kernel_def, #__VA_ARGS__,
            [] (tensorflow::OpKernelConstruction* context)
                -> tensorflow::OpKernel* {
                return new __VA_ARGS__(context);
            });
        return tensorflow::InitOnStartupMarker{};
    } (kernel_builder_expr.Build());
```

← 会创建个 InitOnStartupMarker

InitOnStartupMarker 类提供 operator<<(...)

(其中一个是参数为 const InitOnStartupMarker& 的版本)

(另个是函数模板(T),实现为调用它的 T::operator())

#__VA_ARGS__ ← (变名字符串)

← 方法具体实现, just new 一个 OpKernel 对象, 用 OpKernelConstruction* 来初始化

这是个 lambda 表达式, 返回 InitOnStartupMarker 临时变量,

输入参数为 const KernelDef*

调用时参数 kernel_builder_expr.Build() 返回值

⇓

macro 展开为 .Name("Identity").Device(DEVICE_CPU) .Build() → 返回个 kernel_def 指针

⇓

KernelDefBuilder 对象. 调用 Build() 函数

lamda 表达式实现体

用 kernel_def 来初始化 OpKernelRegistrar 对象, 在它的 ctor 中会调用 Register 函数.

OpKernelRegistrar

↳ ctor InitInternal(......)

OpKernelFactory ↑ ptrOpKernelFactory

实际上就是将 OpKernel 存入全局变量 map 中
factory 对应的 OpKernelFactory.

API : OpKernel* Create(OpKernelConstruction* context) override

GlobalKernelRegistry() ⇐ new KernelRegistry (static)

globalregistry→registry.emplace(key, KernelRegistration(*kernel_def, kernel_class_name, factory

KernelRegistry { mutex
unordered_multimap<string, KernelRegistration> registry;

Data { KernelDef
kernel_class_name
OpKernelFactory.

在前面的 IdentityOp 的注册过程,并没有 OpKernelFactory 与其对应,临时创建一个 lambda 表达式. OpKernelRegistrar 提供 2 种构造方法. { 从一个 factory* 来构造
从一个函数指针来构造
↳ 函数指针可以用来构造一个 PtrOpKernelFactory
Registrar 提供 ↗
而实现的 Create 函数 override 只是转调用这个函数指针.

(DirectSession)
func_info→proc_flr→GetFLR(device_name)
functionLibrary runtime

FunctionLibraryRuntimeImpl (lib) (function.cc)

↳ CreateKernel

input: NodeProperties ⟶ output: OpKernel*

wrap

LocalExecutorParams
↳ create_kernel [function pointer]

FunctionLibraryDefinition ? (map)

find fail ↳ CreateNonCachedKernel

↳ CreateOpKernel (in op_kernel.cc)

{ ① according to nodeProperties's node_def (Key)
find KernelRegistration(item) from KernelRegistry
② call KernelRegistration's factory's Create → OpKernel Kernel
new 一个 OpKernel

OpKernelConstruction 是一个聚合类，聚合大量的参数，用它来构造 OpKernel

StreamExecutor /stream_executor/..., ⇒ StreamExecutor是放在 executor cache中的，BaseGpuDevice只是reference它.

A lot of APIs to operate on stream → StreamExecutorInterface (stream_executor_internal.h)

- memcpy
- synchronize
- launch kernel
- ...
- memalloc
- memfree
- create Blas
- create Dnn
- create FFT

负责对接 Event / stream 创建接口

GpuExecutor (who create it? own it?) — Data { GpuContext* context_; int device_ordinal_; }

在创建平台时有的 StreamExecutor对象，同时会创建 executor

CudaPlatform::GetUncachedExecutor.

StreamExecutor是对各种 Executor的 wrap (platform相关)

StreamExecutor ◇→ StreamExecutorImpl ← HostExecutor, GpuExecutor

LocalDevice ← BaseGPUDevice — Data

GpuDeviceContext { 3个stream*: 1→compute, 1→h2d, 1→d2h, 4→d2d }

StreamExecutor* (not owned?)

API: Compute(OpKernel* op_kernel, OpKernelContext* context)

描述 kernel 运行时输入/出参数，(整个运行环境)

在 BaseGpuDeviceFactory::CreateDevices ⇒ (each gpu) CreateGpuDevice ⇒ new GpuDevice (→ BaseGpuDevice)

.init() → BaseGpuDevice::Init 在这时候创建：

DeviceUtil::ExecutorForTfDeviceId

↙ GpuMachineManager()

↘ stream_executor::Platform* (gpu_init...)

↙ se::MultiPlatformManager::PlatformWithName(GpuPlatform...) ↙ "CUDA"

(stream_executor/multi_platform_manager.h)

MultiPlatformManager 全局是静态函数 just shell ⇒ 真的用 MultiPlatformManagerImpl { mutex; unordered_map<string, platform*> table; }

static Impl() { static ... } 全局local static 访问

stream_executor / platform (platform.h) 接口

enum PlatformKind { CUDA, ROCM, OpenCL, Host, Mock }

platform

HostPlatform → platform
CudaPlatform → platform

HostPlatform → HostExecutor (host_gpu_executor.h)
CudaPlatform → GpuExecutor

API: GetExecutorForDevice ⇒ (StreamExecutor*)

REGISTER_MODULE_INITIALIZER(host_platform,
 stream_executor::host::InitializeHostPlatform());

(Init body)

new HostPlatform();
MultiplatformManager::RegisterPlatform( );

StreamInterface
HostStream (对线程的一种抽象) (有线程/有队列)

API: BlockUntilDone: 等待所有Task都完成.

```
Notification done;
EnqueueTask([&done]() { done.Notify(); });
done.WaitForNotification();
```

放入1个task, 让task自己通知. Good

platformId : void*. 如何让每个platform有不同的platformId呢?

```
namespace {
  int plugin_id_value; // 全局变量, 初始化时会有不同的地址
}.
const platform::Id ID_VAR_NAME = &plugin_id_value;
```

HostExecutor: memcpy, memset, memzero 全部都是生成1个 lambdaTask 放入到 HostStream 去执行 (另外一个线程)

HostExecutor → StreamExecutorInterface

fft
blas
dnn 不支持.

ExecutorCache

container
Data { mutex
map<int, Entry> cache_,

int → device ordinal
Entry → { mutex → per_entry mutex
vector<
 pair<StreamExecutorConfig,
 unique_ptr<StreamExecutor>
> table;

↓ 支持int key.

per_entry mutex allow concurrent initialization of different entries, since initializing an executor may be expensive

```
typedef std::unique_ptr<StreamExecutor> ExecutorFactory();
std::function<ExecutorFactory> factory;
```

StreamExecutor 持有: { platform*
StreamExecutorInterface*
device_ordinal.

每个platform 都Create一个 StreamExecutor对象

```
new StreamExecutor(this, new HostExecutor,
                   device_ordinal)
```

不同platform有不同的Executor

外部接口

ExecutorForTFDeviceId
↳ 将TFDeviceId转成platform id
↳ 根据gpu platform*, 去调用它的 GetExecutorForDevice.

从MultiPlatform的registry中找到name为"CUDA"的platform*.

DirectSessionFactory:

⇒ NewSession
- ① DeviceFactory::AddDevices
  - CPU → threadpooldevice
  - 会根据GPU (所有) 创建gpudevices.
  - 每1个gpudevice对应一个 StreamExecutor
    - gpuexecutor
    - platform
    - bfcallocator
- ② new DirectSession

GpuDevice (Data)

- Streamgroup
  - compute stream
  - host2device stream
  - device2host stream
  - device2device stream.
- GpuDeviceInfo (own)
  - Stream*
  - DeviceContext*.
  - gpuid
- threadPool* — 这个threadPool 只在 TF_GPU_THREAD_MODE 不为"global"时才会创建
- gpuAllocator* / GpuAllocator* (not owned) ← processstate::GetGpuAllocator — GpuProcessState (singleton) own (由于它是singleton 直到退出才会 delete Allocator) — GetGpuAllocator
- StreamExecutor* ← 设计时创建, 用于cache, 不own.
- GpuDeviceContext* (reference counted) → DeviceContext → refcontext (refcounted)

(Session有了 device*(GpuDevice) 让它们来进行 Compute)

StreamExecutor / cuda

- cuda blas
- cuda fft
- cuda event
- cuda dnn
- cuda stream
- cuda timer

blas / fft / dnn，以plugin的方式 register到 pluginRegistry 中。

pluginkind::kBlas　pluginkind::kFFT　pluginkind::kDnn

↓

REGISTER_MODULE_INITIALIZER (register_xxx, { stream_executor::initialize xxx();

RegisterFactory [] {
new gpu::CUDABlas(gpuexe…
new gpu::CUDAFFT(gpuexe…
new gpu::CUDADnn(gpuexe…

(lambda expression)

});

调用 → init

每个plugin都有一个plugin id 其实是个 void*，
定义在每个 cc文件里面的全局变量，有唯一地址。
(编译单元)

StreamExecutor::AsDnn() ← (什么时候会调用 / who调用)
{
dnn_ ← void* / pluginId
↖ Implementation → CreateDnn()
↓
StreamExecutorInterface
↑
GpuExecutor

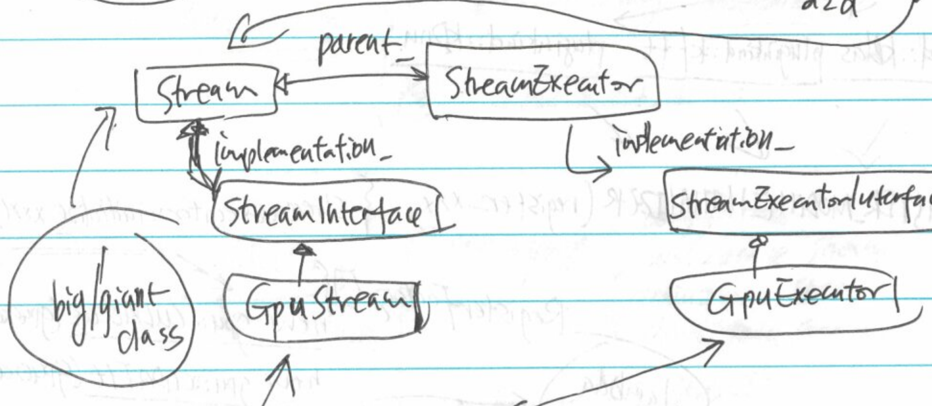
CreateDnn. On Demand call pluginfactory 的 Create 函数

pluginRegistry* registry = PluginRegistry::Instance();

pluginRegistry::DnnFactory * fn = Registry → GetFactory<PluginRegistry::DnnFactory>
(cuda::kCudaPlatformId,
plugin_config_.dnn());
↓
void* / pluginId

fn(this)　new CuDnnSupport*

GpuDevice 创建时 Init 时会 Create StreamGroup { compute / h2d / d2h / d2d } Stream (cudaStream with default flag)



parent_->implementation()->GetStreamImplementation().

↳ new GpuStream(this);

Stream::Init() {

parent_->AllocateStream(this) ⟹ gpuexecutor->AllocateStream(..) (implementation)

↳ AsGpuStream(stream)->Init();

↓ implementation

GpuStream object pointer. 调用

{

↳ 最后其实是让 GpuStream 作初始化

⇓ init()

调用 GpuDriver::CreateStream.

onCreateStream (stream, ∅)

→ 不是 non-blocking stream!!! default

non-blocking stream 表示不会 default stream ∅ (legacy stream) sync.

default stream 表示会和 legacy stream sync

per-thread default stream

不是个 non-blocking stream,

只是个 default flag stream, 故会

legacy stream sync