onePNN-0.1

src 实现文件

实现这些api

mkldnn.hpp —include→ mkldnn.h

C++头文件
定义C++struct
↳ handle 封装C struct 指针

C头文件
定义C-api
- 如何创建
- 如何操作
- 如何销毁

C++ handle → * → C struct object

↘ 并不是实体类

- primitive
- Engine
- stream

C++ handle类

primitive ↔ typedef struct mkldnn_primitive *

mkldnn_primitive_t

↓ * 指针类型 C struct

common/ mkldnn_primitive (基类)

↳ execute

C++ handle类

stream ↔ typedef struct mkldnn_stream *

mkldnn_stream_t

↓ *

common/ mkldnn_stream C struct (基类)

↳ submit
↳ wait

实现在namespace中
mkldnn::impl

C++ handle类

engine ↔ typedef struct mkldnn_engine *

mkldnn_engine_t

↓ *

common/ mkldnn_engine C struct (基类)

~~mkldnn::impl/~~

↳ submit

基类 mkldnn_primitive_base_desc_t

被所有下面类包含

- mkldnn_memory_primitive_desc_t
- mkldnn_reorder_primitive_desc_t
- mkldnn_convolution_primitive_desc_t
- mkldnn_pooling_primitive_desc_t
- mkldnn_relu_primitive_desc_t
- mkldnn_lrn_primitive_desc_t
- mkldnn_inner_product_primitive_desc_t

cpu / cpu_engine

mkldnn/impl/cpu

engine // using mkldnn::impl::engine = mkldnn_engine.

engine_factory

cpu_engine_factory → cpu_engine

↳ engine_create接口.

实现 submit 接口 提交 primitive* batch

↳ primitive → execute

每个engine支持不同的primitives . get_primitive_inits 返回 数组指针

cpu_memory

- jit_avx2_convolution_<f32>
- reference_convolution<f32>
- jit_avx2_pooling<f32>
- reference_pooling<f32>
- jit_avx2_relu<f32>
- ~~jit_avx2_lrn<f32>~~
- reference_relu<f32>
- jit_avx2_lrn<f32>
- reference_lrn<f32>
- gemm_inner_product<f32>
- reference_inner_product<f32>.
- jit_avx2_convolution_relu<f32>
- reference_convolution_relu<f32>

多态

每个desc包含一个implementation*, 指向

mkldnn::impl::primitive_impl 结构体

包含 如何创建这个primitive 函数指针

reference 实现是用omp ~~pr~~directive来实现的、

jit实现是手写汇编 用 xbyak 生成 binary code

avx2

Xbyak::CodeGenerator

oneDNN负责 生成asm ← asm jit

jit_generator

jit_avx2relu → xbyak_relu

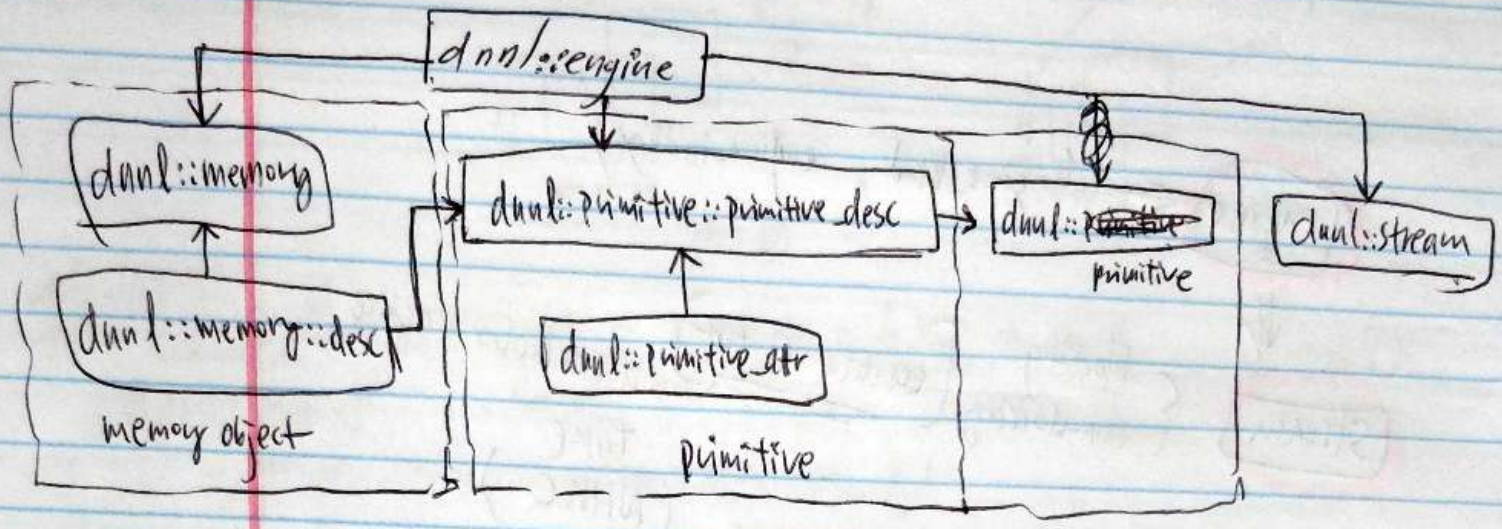
jit_relu* → getCode → Kernel void (*)(const void*)

jit_relu_reminder* → getCode

可执行代码 chunk

OneDNN

programming model: primitives, engines, Streams, and memory objects



primitive encapsulates a particular computation:
- forward convolution
- backward LSTM computations
- a data transformation.

primitive ↔ (different) pure function

primitive ↑ stores state

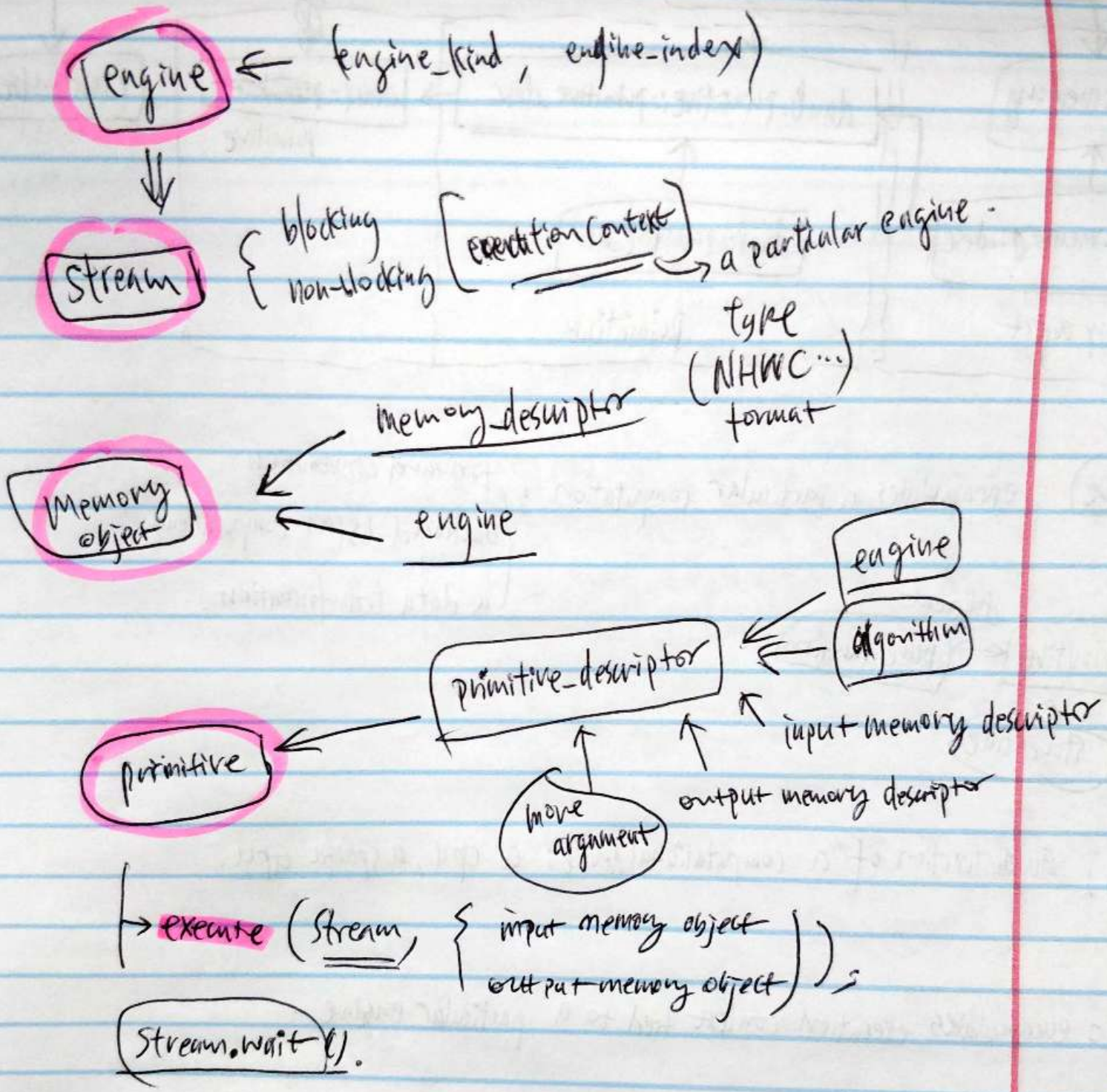
engine: an abstraction of a computational device: a CPU, a specific GPU.

Streams: encapsulates execution context tied to a particular engine.

memory objects: encapsulate handles to memory allocated on a specific engine.
tensor dimensions, data type, and memory format.

dnnl_memory_desc_t ⟺ dnnl::memory::desc

C

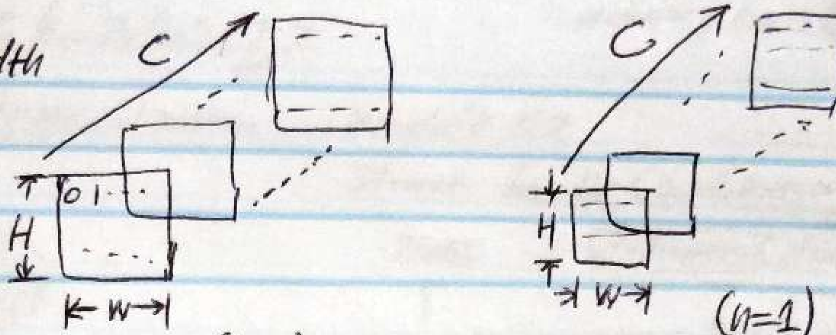

engine
engine_kind, engine_index
Stream
blocking
non-blocking
execution Context
a particular engine.
type (NHWC...) format
memory_descriptor
Memory object
engine
engine
algorithm
primitive_descriptor
input memory descriptor
primitive
output memory descriptor
more argument
execute (Stream, { input memory object, output memory object });
Stream.wait();

Data format (layout)

(caffe default) (TF support)

$\underline{N} \times \underline{C} \times \underline{H} \times \underline{W}$ N=2, C=16, H=5, W=4

batch channel High Width



(n=0) (n=1)

value(n, c, h, w) = n*CHW + c*HW + h*W + w·1 (=offset)

(logical offset) stride stride stride stride

index strides = (CHW, HW, W, 1)

nchw

outer-most dimension → inner-most dimension

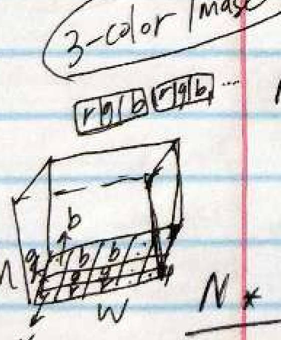
$N \times H \times W \times C$ (tensorflow default)

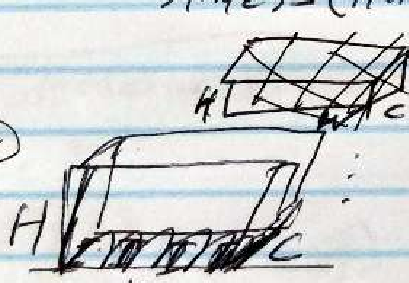
大写C 小写c

offset_nhwc(n, c, h, w) = n*HWC + h*WC + w*C + c*1

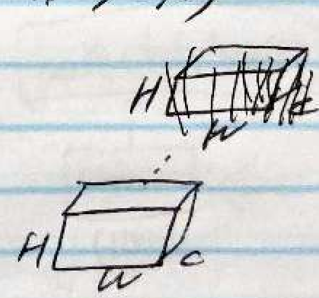
strides = (HWC, WC, C, 1)

3-color image

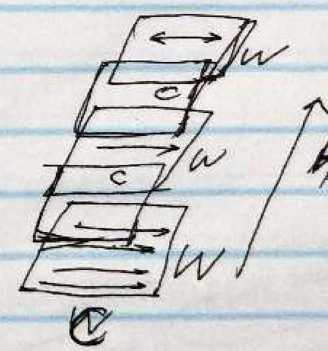




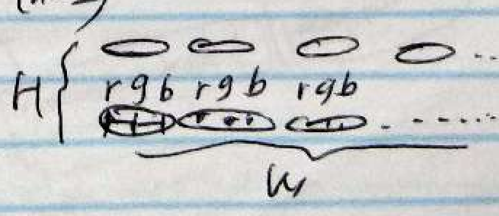
(n=0)



(n=1)



N * H * W * C → C先遍历



handle | template类

template <typename T, typename traits = handle_traits<T>>

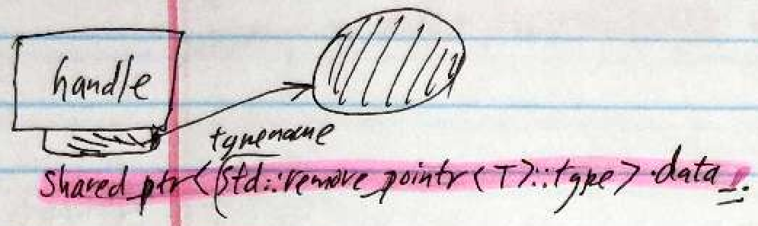
包装一个指针 与 handletrait<T>

primary traits 为空.

针对不同的T, 可多个specialization

```
template <>
struct handle<dnnl_memory_t> {
    static ... destructor (dnnl_memory_t p)
    {
        dnnl_memory_destroy(p);
    }
};
```

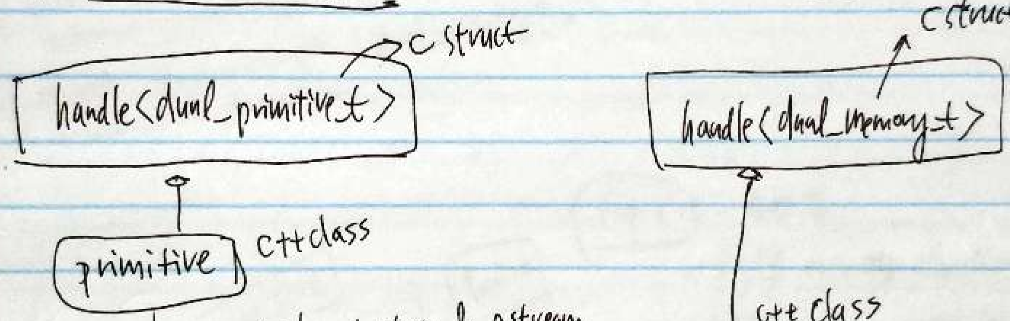
trait类, 提供destructor



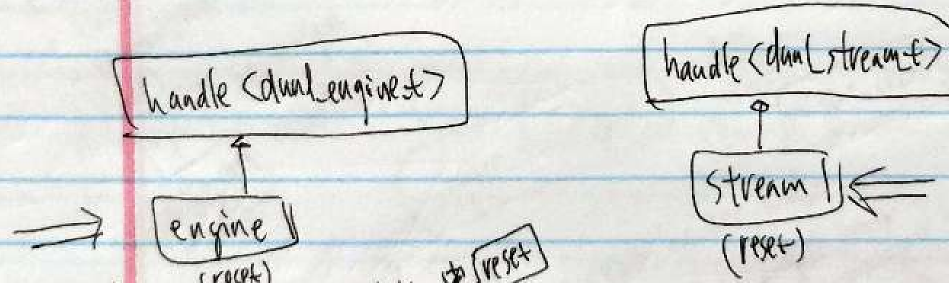
handle<dnnl_memory_t> hdl(一个具体T类型 → 指针, weak?);

类似于 shared_ptr

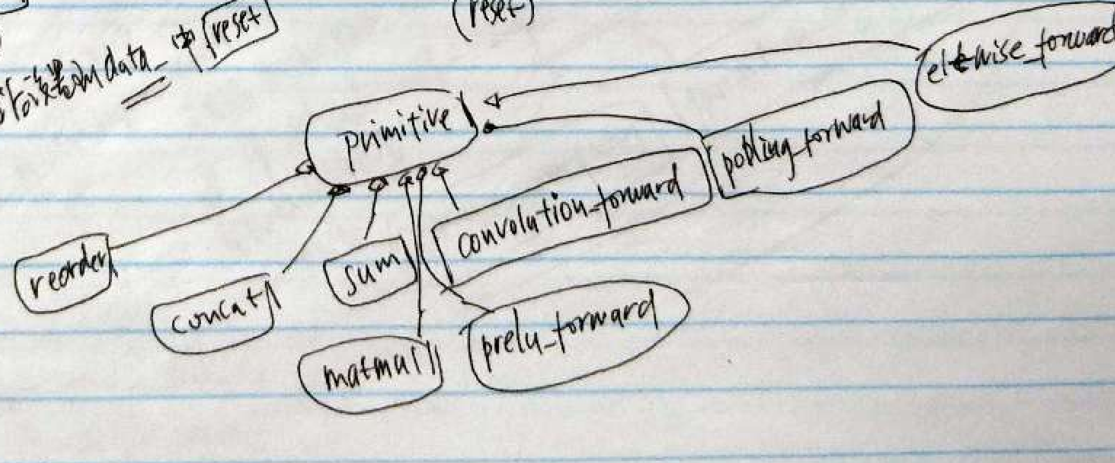
typedef struct dnnl_memory* dnnl_memory_t;



→ execute(const stream& astream,
    const std::unordered_map<int, memory> & args) const;



在ctor中 调用C-API 构造 C-struct, 然后设置到data_中 reset



struct dnnl_memory_t* = dnnl_memory_t;

dnnl_memory_t → dnnl::impl::engine_t* engine_,

dnnl_memory_t → dnnl::impl::memory_desc_t md_;

dnnl_memory_t → std::unique_ptr<dnnl::impl::memory_storage_t> memory_storage_;

dnnl.hpp C++ 头文件

dnnl.h C头文件.

⇐定义在C public头文件中 (dnnl_types.h)

typedef struct dnnl_engine* dnnl_engine_t;

typedef struct dnnl_memory* dnnl_memory_t;

⋮

定在每个具体的实现中

using dnnl::impl::engine_t = dnnl_engine;

engine

runtime_kind
- none
- seq
- omp
- tbb
- threadpool
- ocl
- sycl 支持cpu/gpu.

编译期决定, 预编译选项.

\+ engine_kind
- cpu
- gpu

\+ index

engine: base类
- 提供 memory allocation
- 提供 stream create

engine_factory_t

基类

接口: engine_create

C接口函数中

在dnnl_engine_create中

会根据 engine_kind, runtime_kind,

⇓ 来决定创建什么样的 engine_factory

cpu::cpu_engine_factory

gpu::ocl::ocl_engine_factory

sycl::get_engine_factory

预编译选项 cpu/gpu.

代码结构:

src/common: 通用基类..

cpu: cpu-specific subclass

gpu: gpu-specific subclass

sycl: ......

cpu

dnnl/impl/cpu
namespace

engine_t (dnnl_engine_t)

cpu_engine_t

实现接口：create_memory_storage
create_stream
⋮

engine_factory_t

cpu_engine_factory_t

实现接口 engine_create

engine_t*

new cpu_engine_t

抽象门

memory_storage_t ⟷ engine_t

create_memory_storage：cpu_memory_storage_t ⟷ unique_ptr<void, void (*)(void*)> data_;

either 请求分配 ⇒ free 析构，
or 传入 handle (ptr)
传入 handle release data_deleter
do nothing

类似于 void* data_[n];

malloc
memalign

- destroy (void* ptr) {
  free(ptr);
  }
- release (void* ptr) {
  none;
  }

---

generic 层面 primitive / Stream

using dnnl::impl::stream_t = dnnl_stream

dnnl_stream ⟷ dnnl_engine_t

实现对中
实现 C-API
dnnl_stream_wait(stream_t* stream)
↳ stream->wait();

接口
→ enqueue_primitive
→ before_exec_hook
→ after_exec_hook
→ wait

基类
实现

子类实现在某个具体的 architecture 目录中

dnnl_primitive_execute
C-API

dnnl::impl::primitive_execute : (primitive_iface, exec_ctx_t & ctx) {
auto stream = ctx.stream();
stream->before_exec_hook();
if (get_verbose()) {
profile
stream->enqueue_primitive (primitive_iface, ctx);
print profile runtime.
} else {
stream->enqueue_primitive(primitive_iface, ctx);
}

⇒ dnnl_primitive::execute (exec_ctx_t &)

由 primitive_desc 创建

wrap
shared_ptr

dnnl_primitive ⟷ dnnl::impl::primitive_t

执行
拥有
execute
但不是虚函数

子类继承实现 ⇒ 提供接口 = 0.
execute (const exec_ctx_t &)

转调用