PyTorch & Thunder 分享

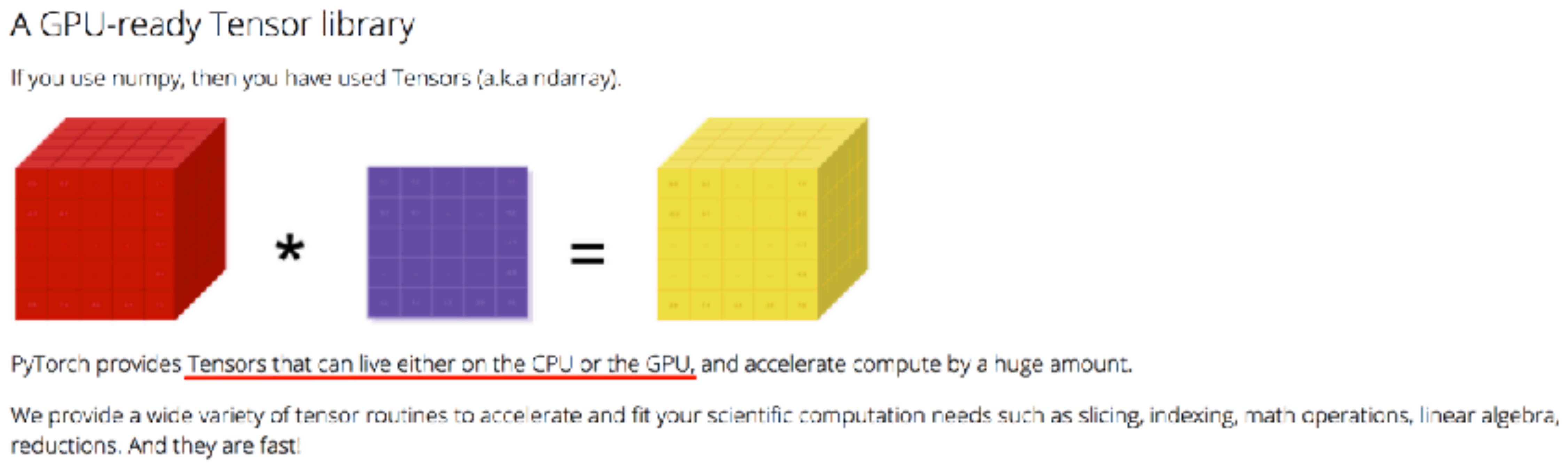
许振雷

PPT大纲

- PyTorch介绍
- PyTorch实现方式
- Autodiff介绍
- GPU与CPU切换实现方式
- CUDA编程
- 自己动手做一个深度学习框架Thunder

PyTorch是什么?

用官方的语言表达就是



A GPU-ready Tensor library

If you use numpy, then you have used Tensors (a.k.a ndarray).

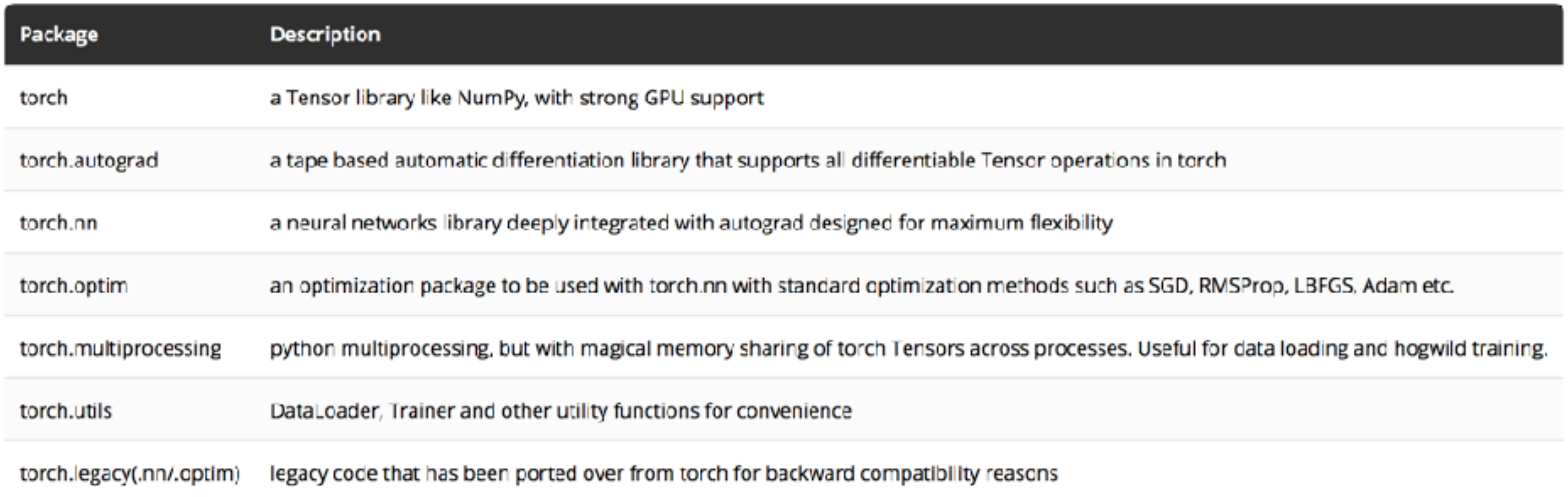
PyTorch provides Tensors that can live either on the CPU or the GPU, and accelerate compute by a huge amount.

We provide a wide variety of tensor routines to accelerate and fit your scientific computation needs such as slicing, indexing, math operations, linear algebra, reductions. And they are fast!

一个支持张量无缝切换的GPU与CPU的框架。(**核心**)

PyTorch是什么？

在前面的基石之上，提出了以下的各种计算库

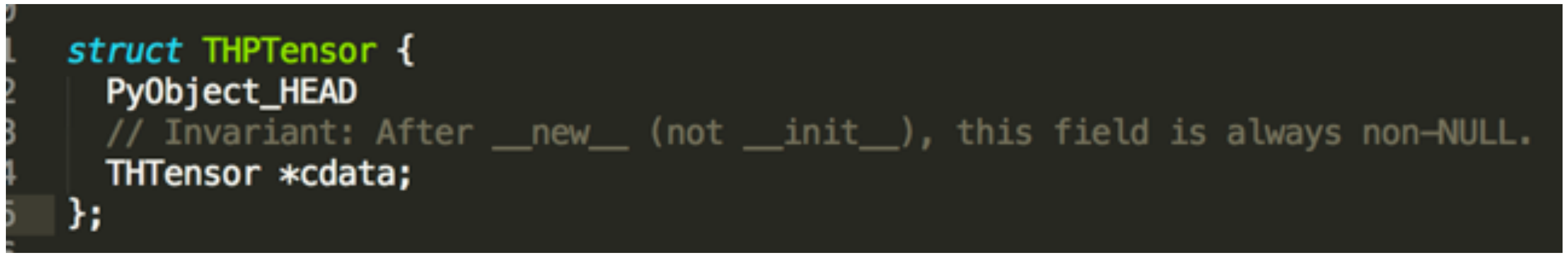


Package	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience
torch.legacy(.nn/.optim)	legacy code that has been ported over from torch for backward compatibility reasons

1. torch 类似numpy的数据结构 能够在GPU和CPU中进行切换
2. autograd 自动求导
3. nn 各种神经网络的定义库
4. optim 各种梯度下降学习函数
5. utils 各种读取数据的函数

PyTorch数据是怎么定义的?

PyTorch使用PyObject对象自定义数据类型



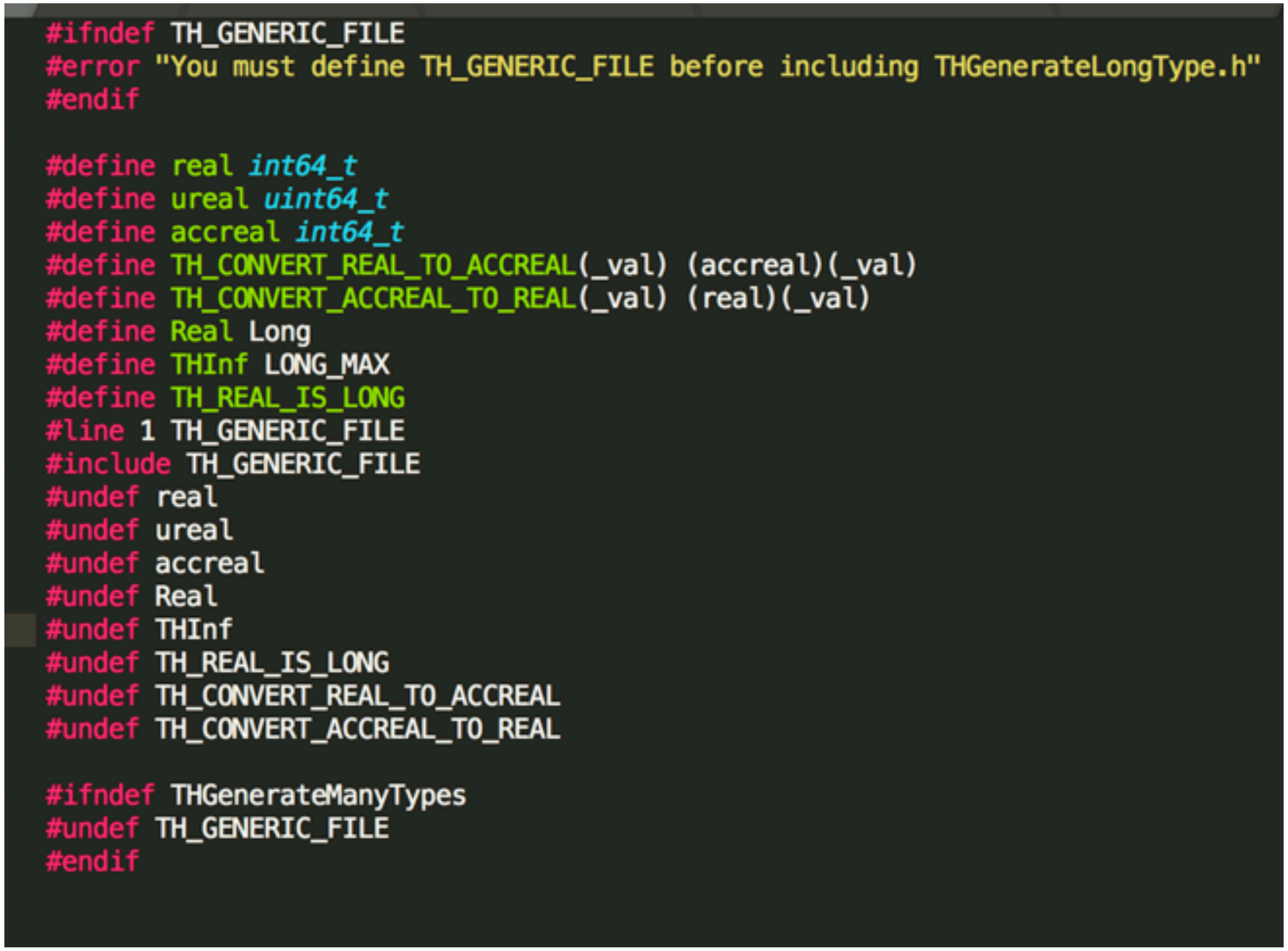
```
struct THPTensor {
  PyObject_HEAD
  // Invariant: After __new__ (not __init__), this field is always non-NULL.
  THTensor *cdata;
};
```

通过一般化定义PyObject对象来建立THPTensor。其中THTensor是通过C语言定义结构体生成，采用C语言的宏来实现多态。(非常有用的技巧)

PyTorch数据是怎么定义的?

TPTensor的定义方式

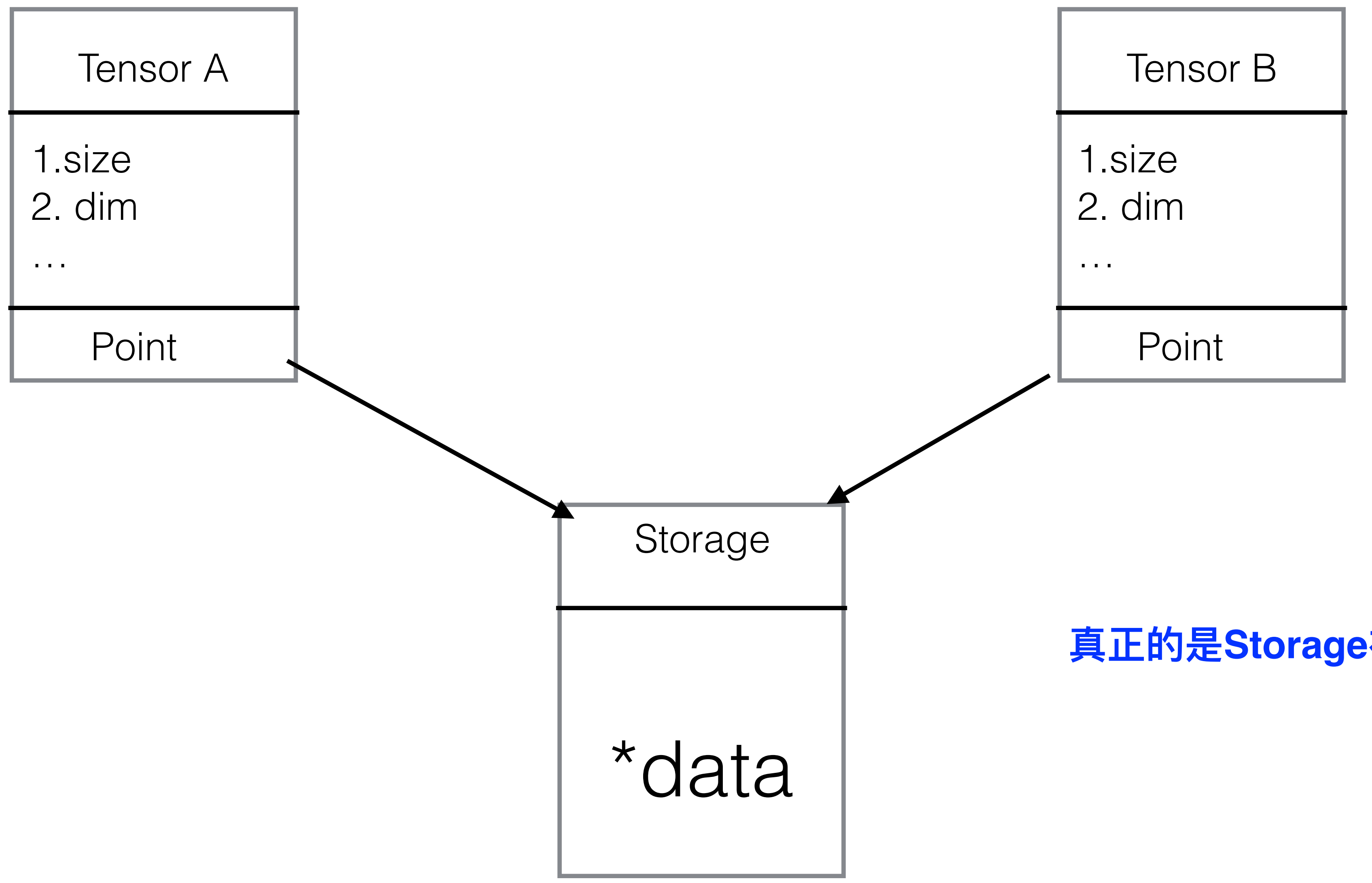
在 pytorch/aten/src/TH/THGenerateLongType.h



```
#ifndef TH_GENERIC_FILE
#error "You must define TH_GENERIC_FILE before including THGenerateLongType.h"
#endif

#define real int64_t
#define ureal uint64_t
#define accreal int64_t
#define TH_CONVERT_REAL_TO_ACCREAL(_val) (accreal)(_val)
#define TH_CONVERT_ACCREAL_TO_REAL(_val) (real)(_val)
#define Real Long
#define THInf LONG_MAX
#define TH_REAL_IS_LONG
#line 1 TH_GENERIC_FILE
#include TH_GENERIC_FILE
#undef real
#undef ureal
#undef accreal
#undef Real
#undef THInf
#undef TH_REAL_IS_LONG
#undef TH_CONVERT_REAL_TO_ACCREAL
#undef TH_CONVERT_ACCREAL_TO_REAL

#ifndef THGenerateManyTypes
#undef TH_GENERIC_FILE
#endif
```

PyTorch数据是怎么定义的?

当Tensor定义完，在PyTorch中所有的Tensor是如下存放的:



真正的是Storage在GPU于CPU中移动

Autodiff怎么实现?

在深度学习框架中，最重要的是如何实现反向求导？传统的求导方式是正向求导

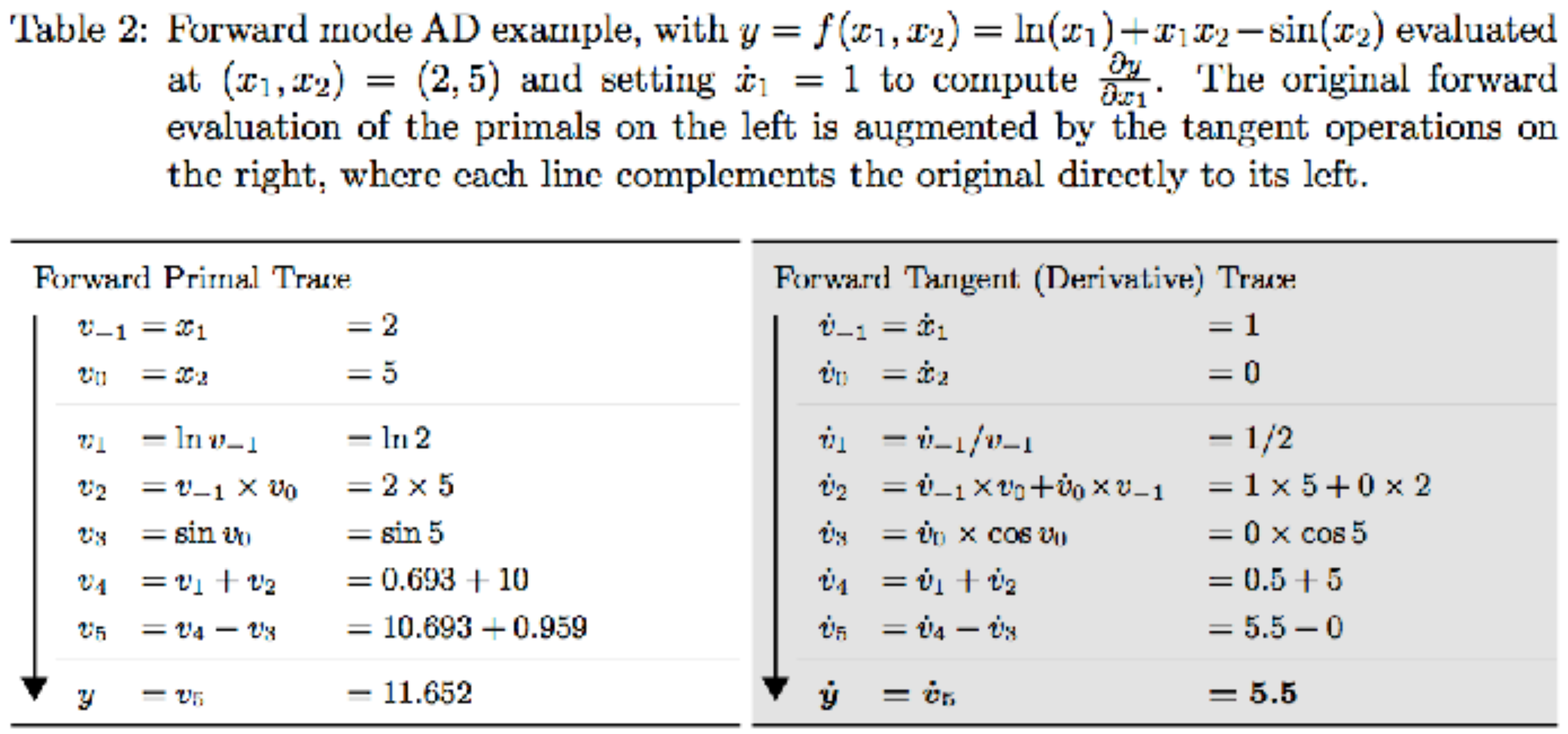


Table 2: Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

https://arxiv.org/abs/1502.05767v4(Automatic Differentiation)

Autodiff怎么实现?

而PyTorch采用“逆”图来实现

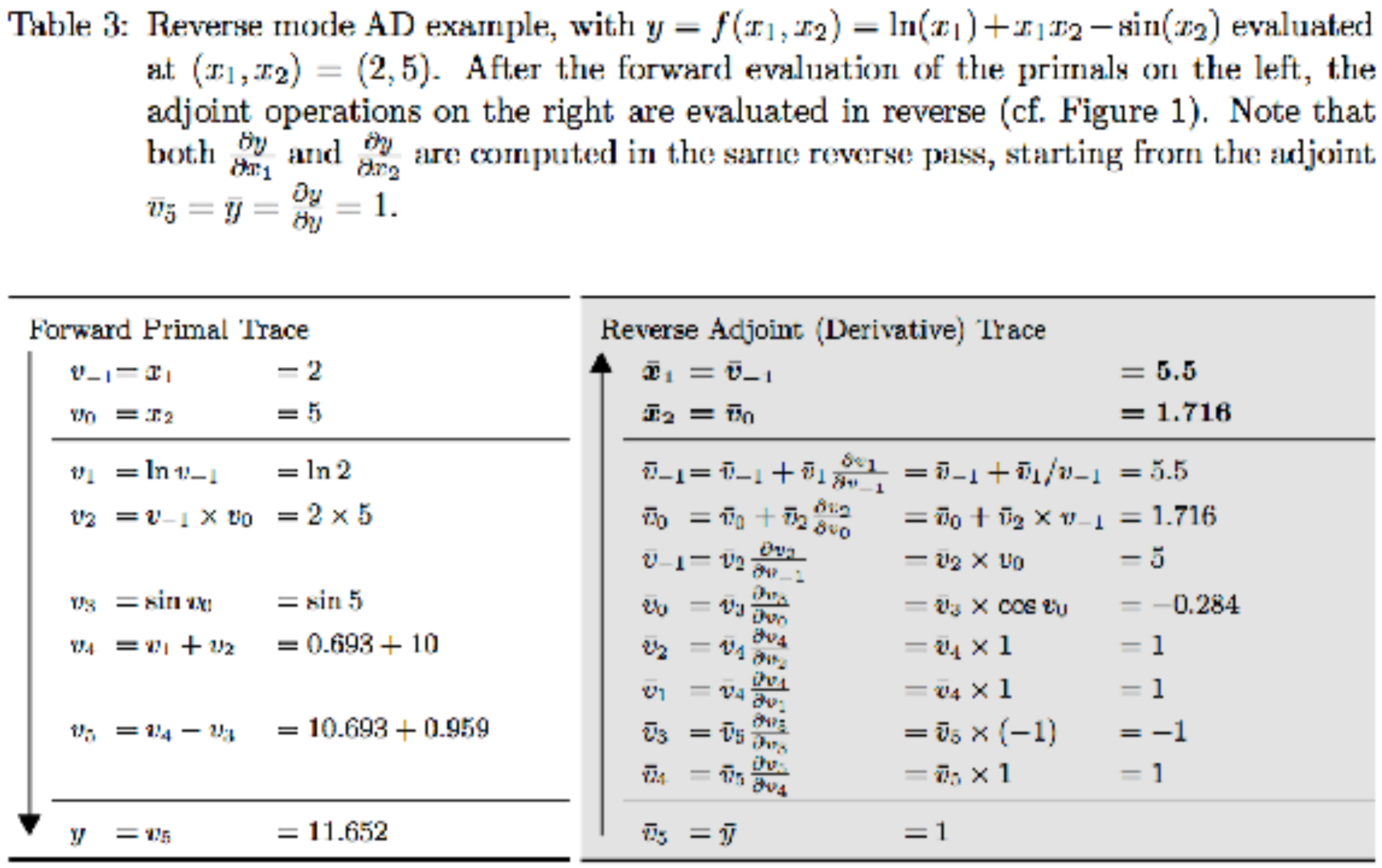


Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

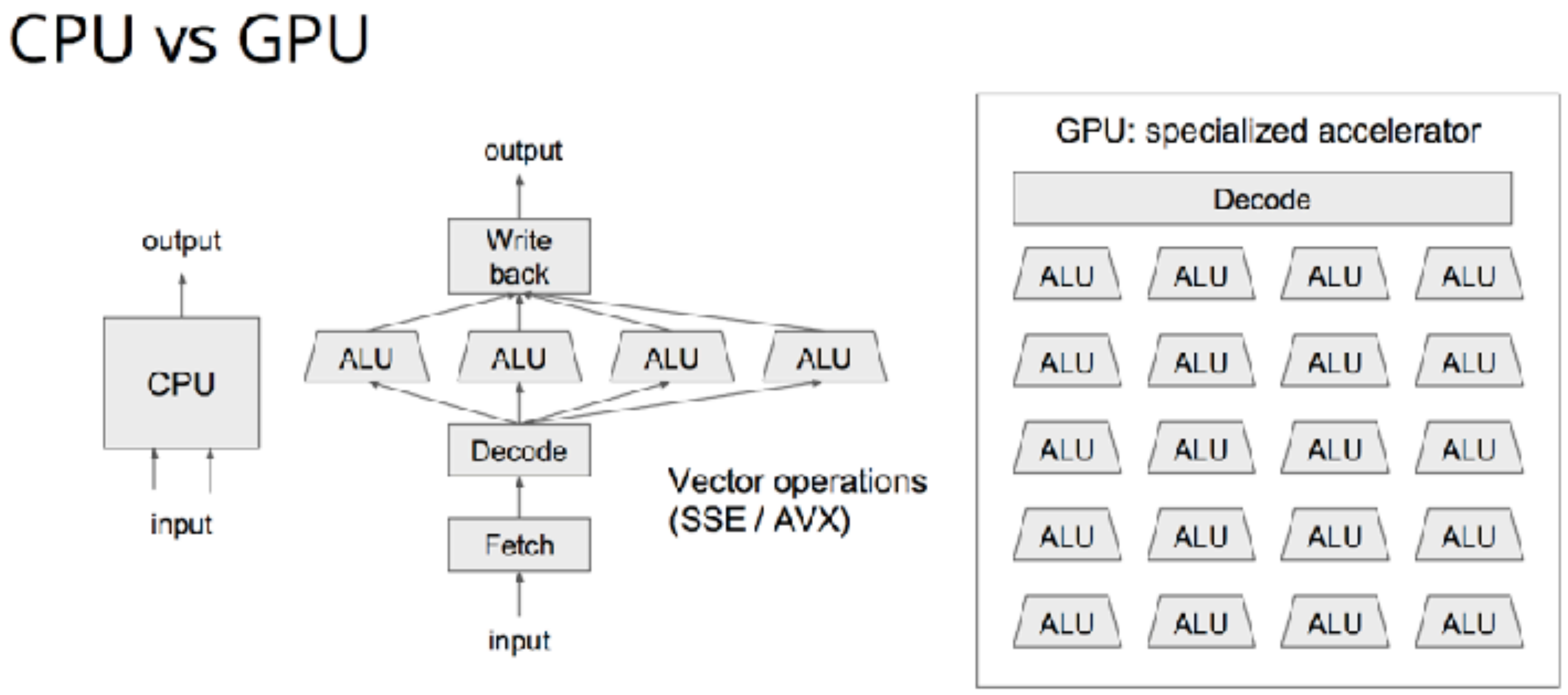
Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Reverse Adjoint (Derivative) Trace			
$\bar{x}_1$	$= \bar{v}_{-1}$		$= 5.5$
$\bar{x}_2$	$= \bar{v}_0$		$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	

https://arxiv.org/abs/1502.05767v4(Automatic Differentiation)

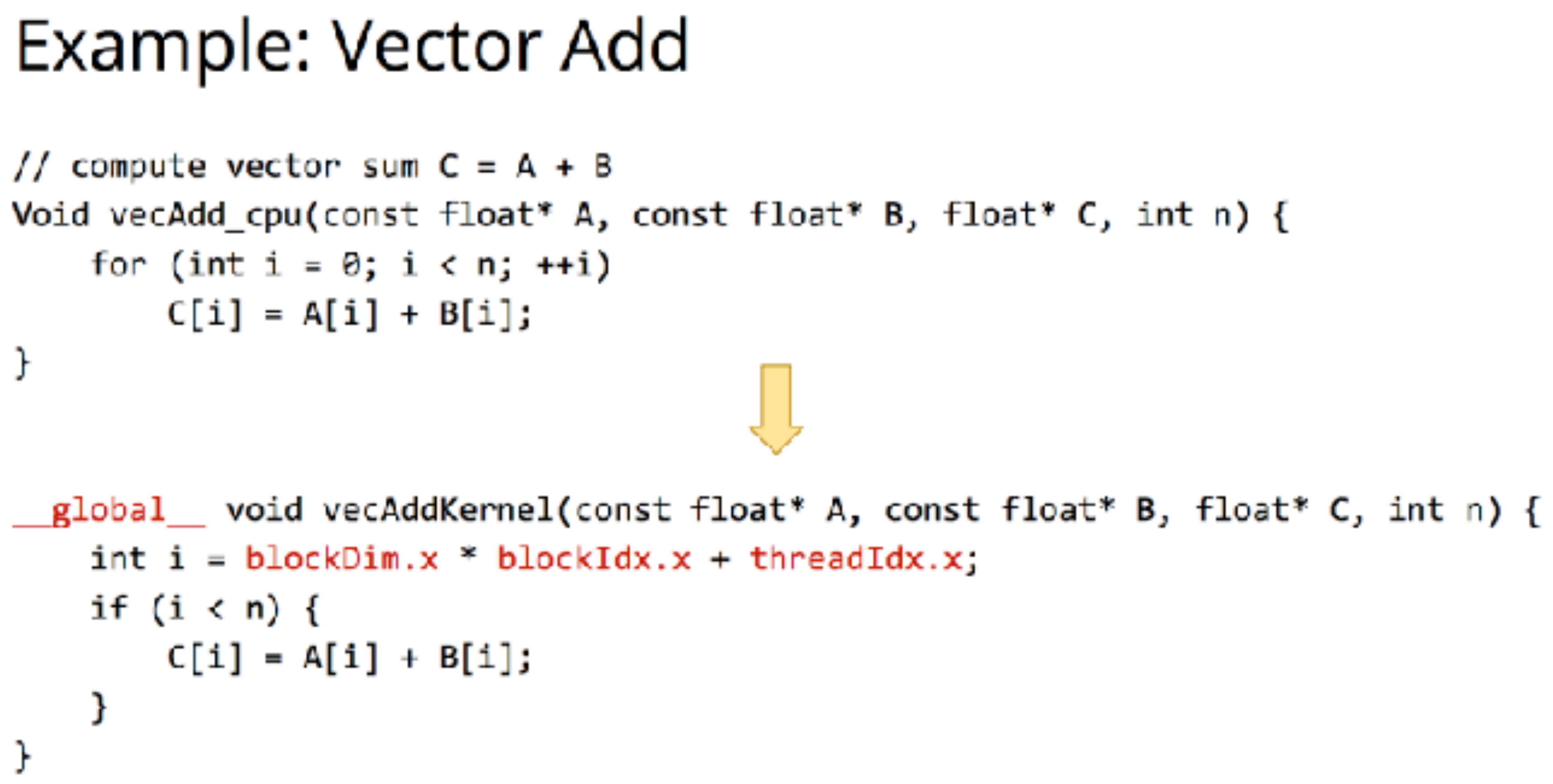
如何实现GPU与CPU切换?

为啥使用GPU，原因就不用说了？ （**快**）



CUDA编程

CUDA是NVIDIA公司提出的一种计算平台，通过编写类C的代码运行在GPU上。



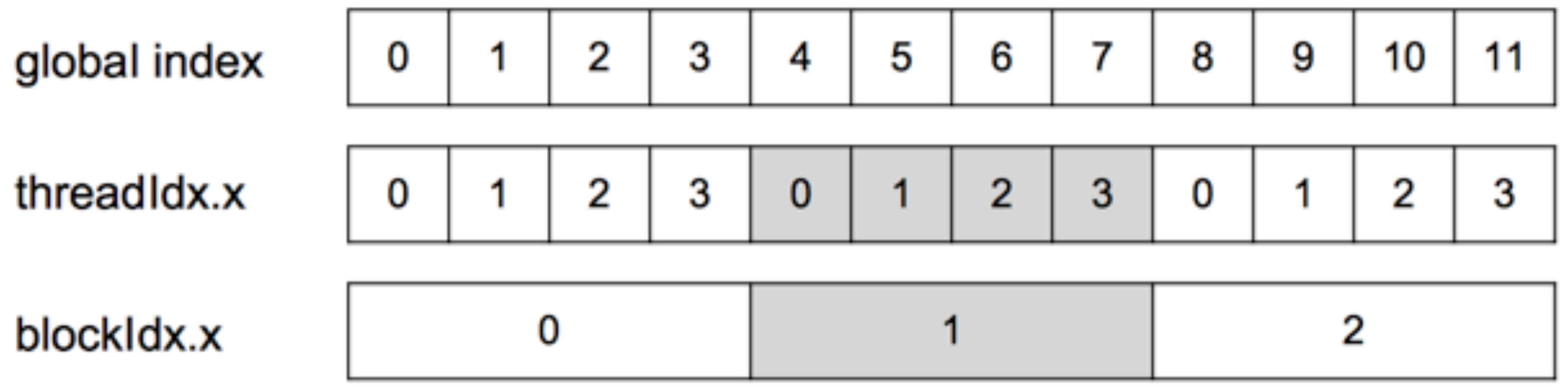
Example: Vector Add

```
// compute vector sum C = A + B
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

CUDA编程

Example: Vector Add

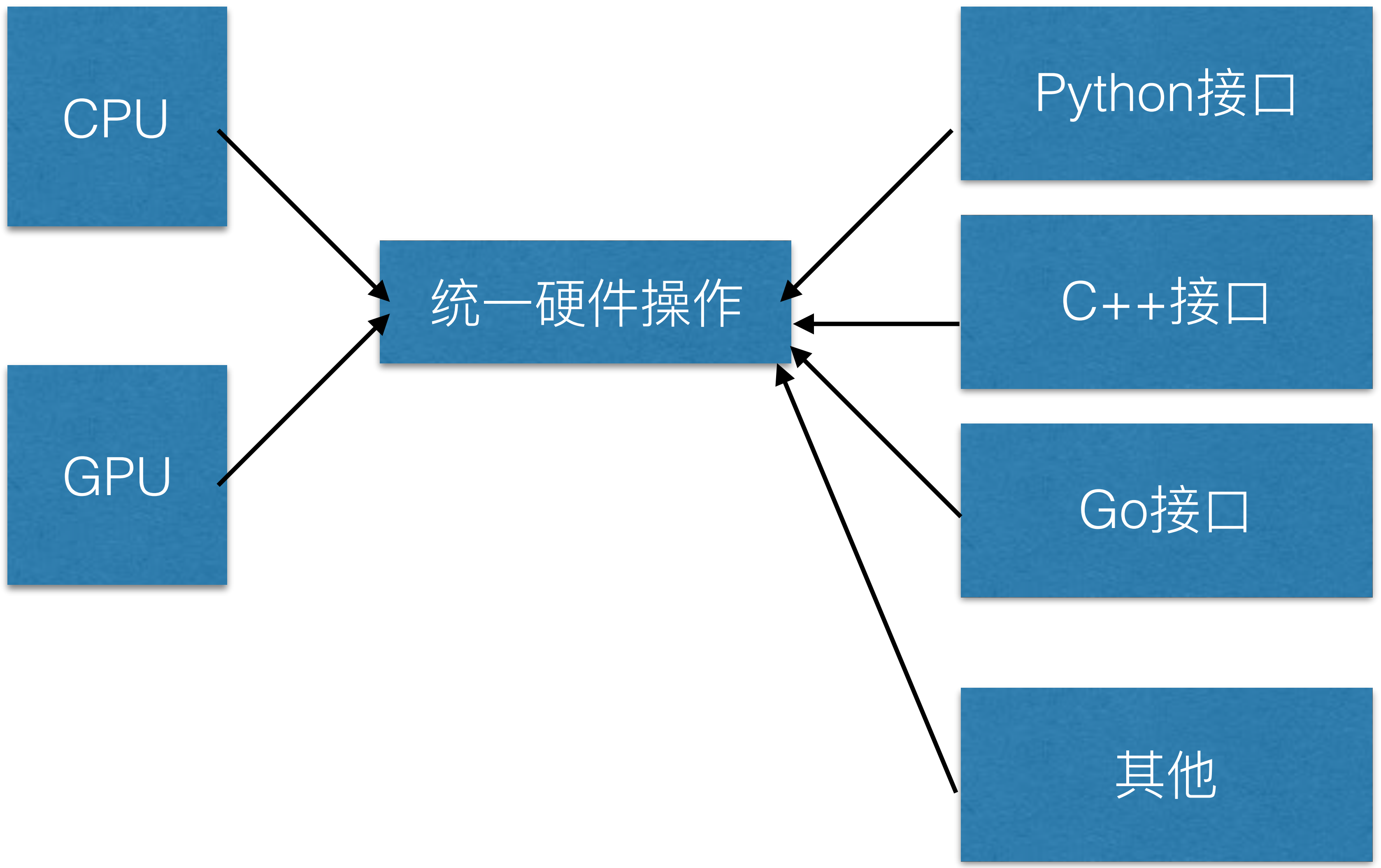


Suppose each block only includes 4 threads: blockDim.x = 4

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Compute the global index

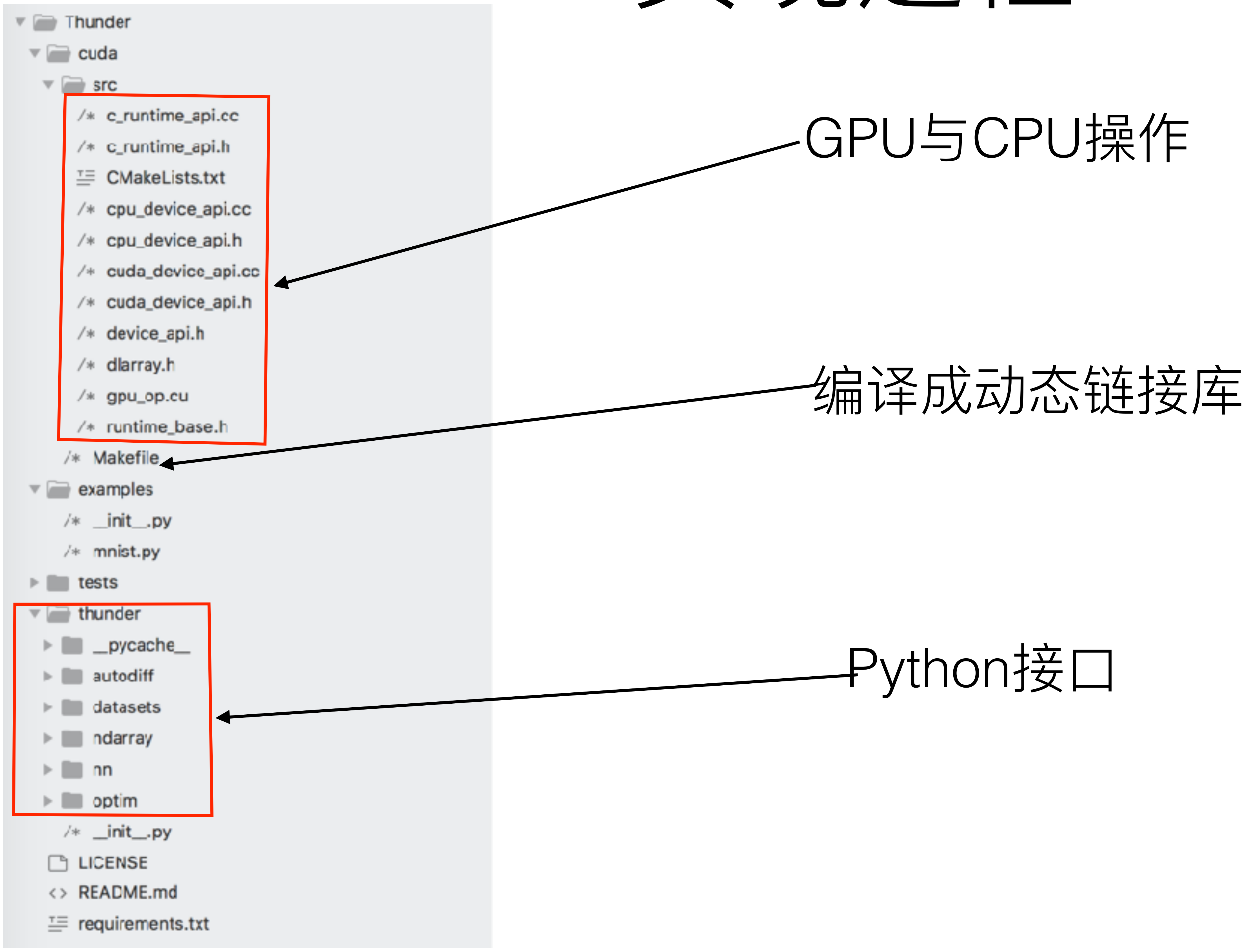
CPU与GPU无缝切换细节



Thunder

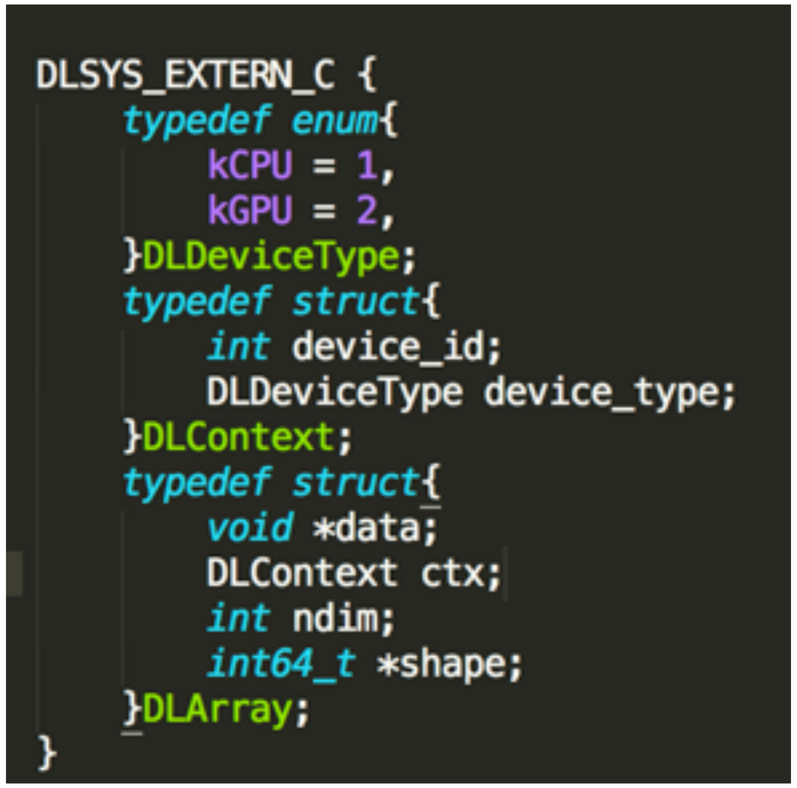
在前面的技术层面上，我编写了自己的深度学习框架Thunder，采用C/C++/CUDA/Python。

Thunder实现过程


```
▼ Thunder
  ▼ cuda
    ▼ src
        c_runtime_api.cc
        c_runtime_api.h
        CMakeLists.txt
        cpu_device_api.cc
        cpu_device_api.h
        cuda_device_api.cc
        cuda_device_api.h
        device_api.h
        dlarray.h
        gpu_op.cu
        runtime_base.h
      Makefile
  ▼ examples
      __init__.py
      mnist.py
  ▶ tests
  ▼ thunder
    ▶ __pycache__
    ▶ autodiff
    ▶ datasets
    ▶ ndarray
    ▶ nn
    ▶ optim
      __init__.py
    LICENSE
    README.md
    requirements.txt
```

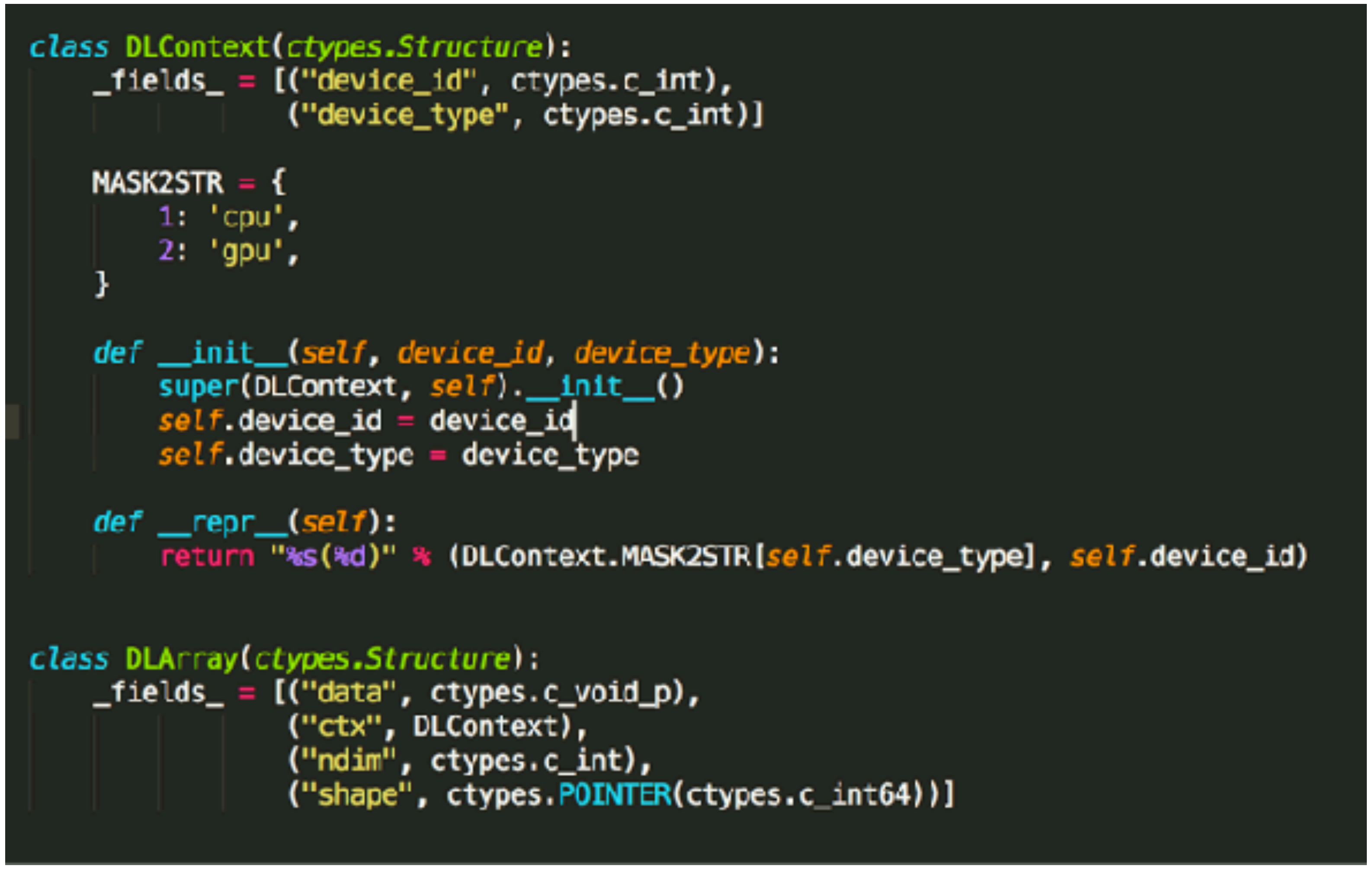
Thunder实现过程

其中，基础Tensor定义为



```c
DLSYS_EXTERN_C {
    typedef enum{
        kCPU = 1,
        kGPU = 2,
    }DLDeviceType;
    typedef struct{
        int device_id;
        DLDeviceType device_type;
    }DLContext;
    typedef struct{
        void *data;
        DLContext ctx;
        int ndim;
        int64_t *shape;
    }DLArray;
}
```

C结构体

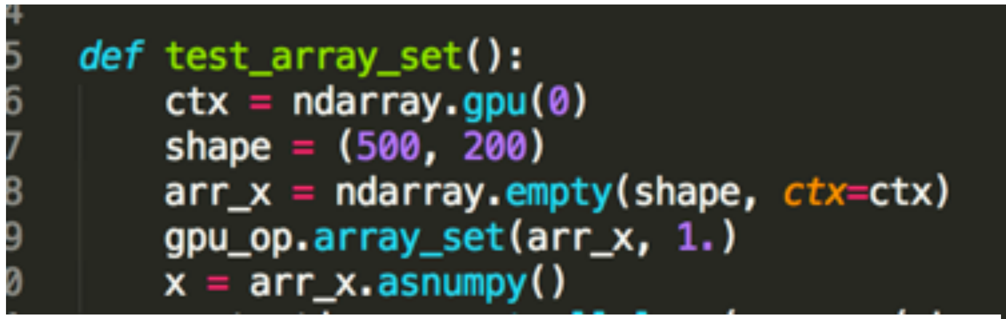


```python
class DLContext(ctypes.Structure):
    _fields_ = [("device_id", ctypes.c_int),
                ("device_type", ctypes.c_int)]

    MASK2STR = {
        1: 'cpu',
        2: 'gpu',
    }

    def __init__(self, device_id, device_type):
        super(DLContext, self).__init__()
        self.device_id = device_id
        self.device_type = device_type

    def __repr__(self):
        return "%s(%d)" % (DLContext.MASK2STR[self.device_type], self.device_id)


class DLArray(ctypes.Structure):
    _fields_ = [("data", ctypes.c_void_p),
                ("ctx", DLContext),
                ("ndim", ctypes.c_int),
                ("shape", ctypes.POINTER(ctypes.c_int64))]
```

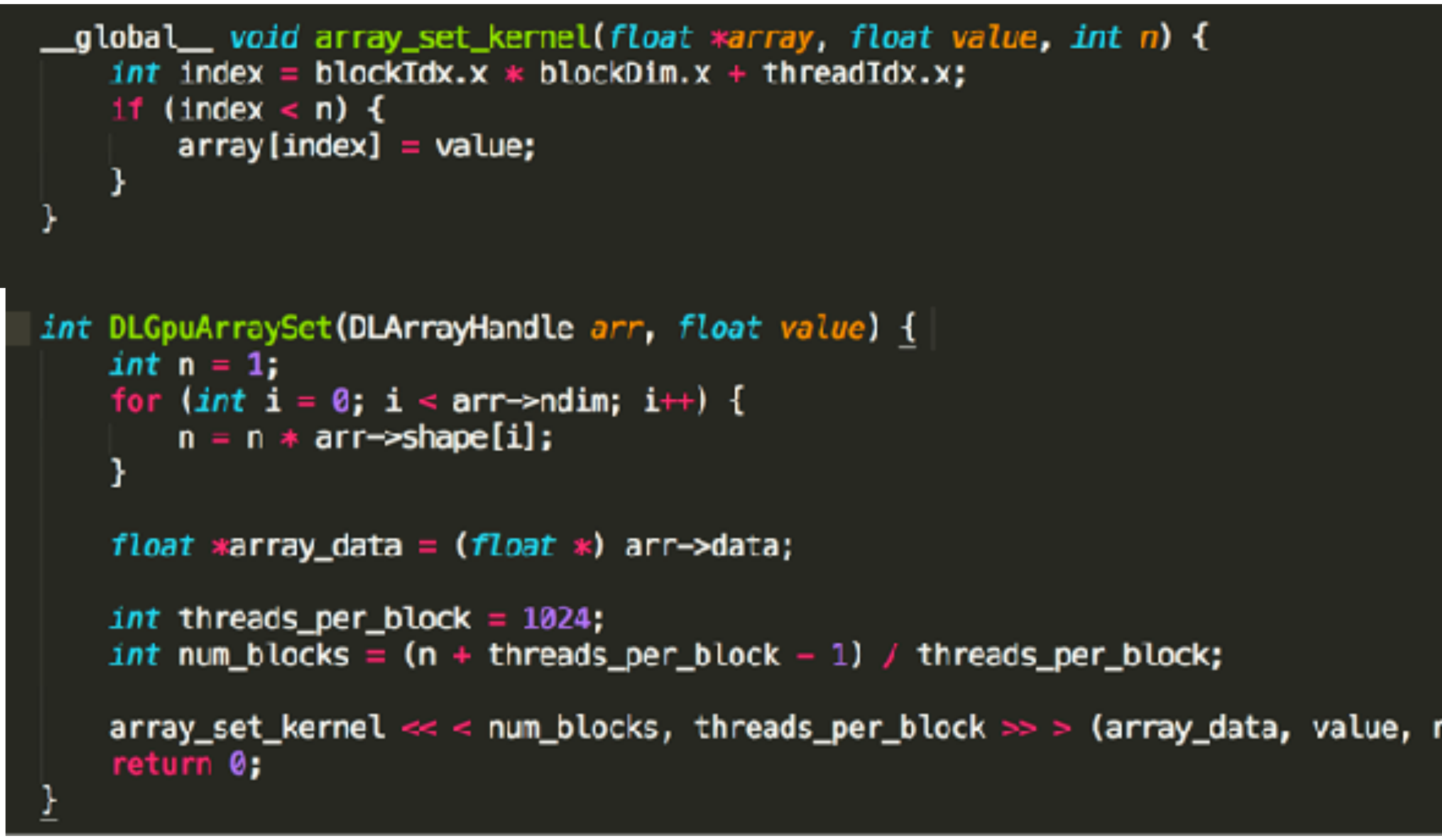
对应的Python结构体

Thunder实现过程

当结构体定义完，可以对数据进行操作，如下所示



```python
def test_array_set():
    ctx = ndarray.gpu(0)
    shape = (500, 200)
    arr_x = ndarray.empty(shape, ctx=ctx)
    gpu_op.array_set(arr_x, 1.)
    x = arr_x.asnumpy()
```



```c
__global__ void array_set_kernel(float *array, float value, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n) {
        array[index] = value;
    }
}

int DLGpuArraySet(DLArrayHandle arr, float value) {
    int n = 1;
    for (int i = 0; i < arr->ndim; i++) {
        n = n * arr->shape[i];
    }

    float *array_data = (float *) arr->data;

    int threads_per_block = 1024;
    int num_blocks = (n + threads_per_block - 1) / threads_per_block;

    array_set_kernel << < num_blocks, threads_per_block >> > (array_data, value,
    return 0;
}
```

1. 定义上下文为GPU
2. 定义shape
3. 在GPU上分配空间
4. 设置为1
5. 转换成cpu数据类型

Thanks