



智能计算系统

第五章 编程框架原理

中国科学院计算技术研究所

李威 副研究员

liwei2017@ict.ac.cn

提纲

- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练
- ▶ 本章小结

概述

- ▶ 学习智能计算系统，了解编程框架的原理大有裨益
 - ▶ 编写与框架底层更为契合、性能更优的代码
 - ▶ 定制化扩展编程框架，为新算法、新设备提供支持
- ▶ 智能计算系统中编程框架的四大模块
 - ▶ 必备：计算图构建模块和计算图执行模块
 - ▶ 追求更高性能：深度学习编译模块和分布式训练模块

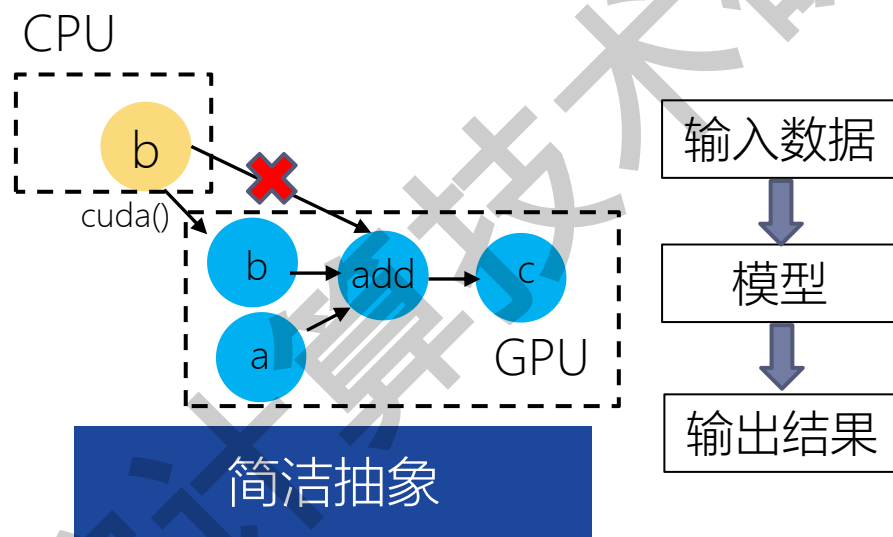
提纲

- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练
- ▶ 本章小结

1、设计原则

- ▶ 简洁性 (Simplicity)

- ▶ 框架提供一套抽象机制，用户仅需关心算法本身和部署策略



- ▶ 易用性 (Usability)

- ▶ 高效性 (Performance)

1、设计原则

- ▶ 简洁性 (Simplicity)
- ▶ 易用性 (Usability)
 - ▶ 熟悉的开发范式：如PyTorch始于Python，忠于Python
 - ▶ 直观且用户友好的接口：如PyTorch提供了命令式的动态图编程方法
- ▶ 高效性 (Performance)

1、设计原则

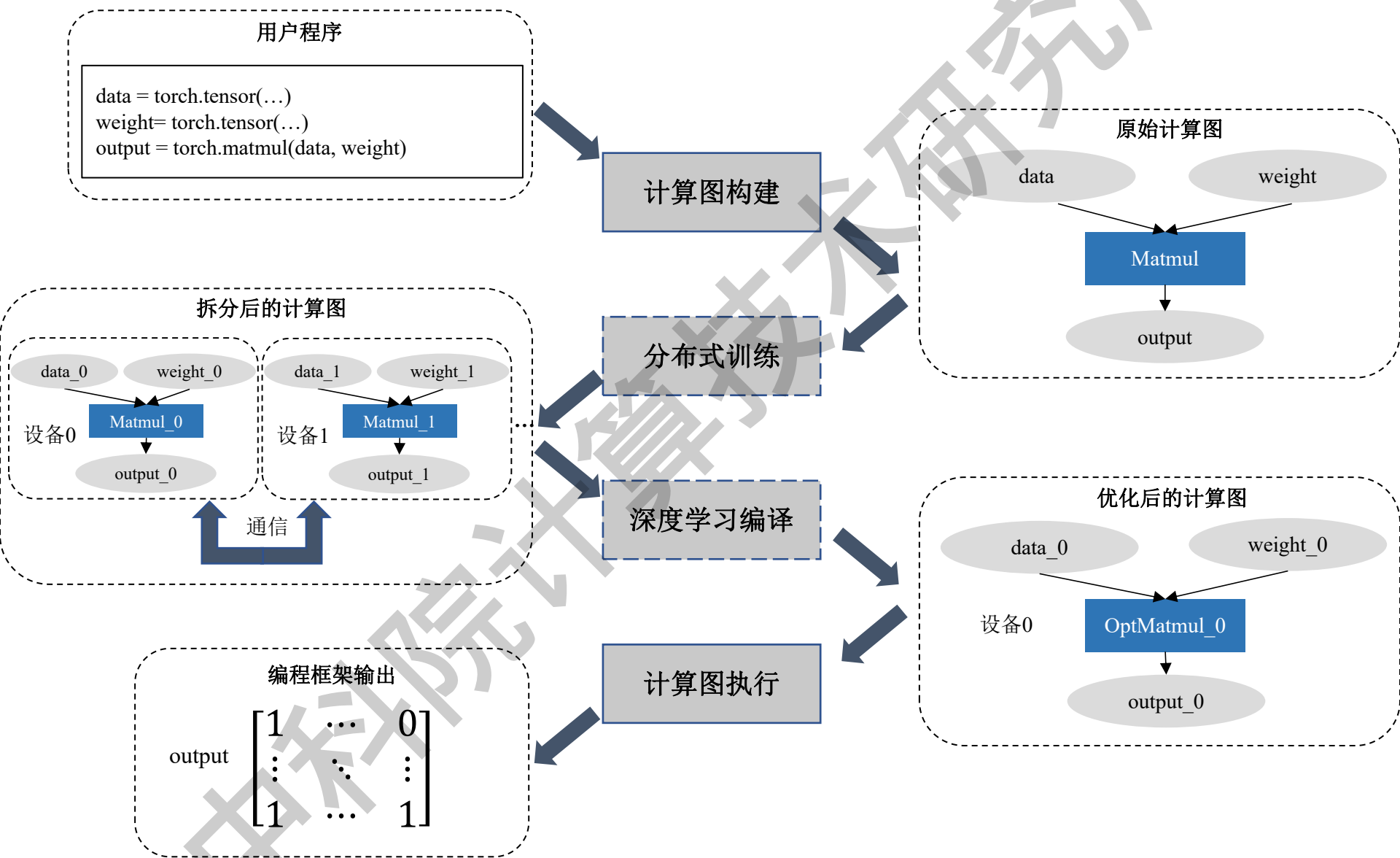
- ▶ 简洁性 (Simplicity)
- ▶ 易用性 (Usability)
- ▶ 高效性 (Performance)
 - ▶ 如采用静态图编程方式，可以生成完整的计算图并进行全局优化，从而尽量提高用户应用程序的运行效率
 - ▶ 支持深度学习编译技术，多层次表示优化，充分利用用户硬件的计算能力
 - ▶ 支持多机多卡条件的分布式训练，从而高效支持大规模深度学习任务

2、整体架构

▶ 四大模块

- ▶ 计算图构建模块：完成从输入的用户程序到编程框架内部原始计算图的转换过程，编程框架的入口模块
- ▶ 分布式训练模块：应对更大规模的神经网络，将训练、推理任务从一台设备扩展到多台设备
- ▶ 深度学习编译模块：对计算图分别进行图层级和算子层级的编译优化，从而提升单设备上的执行效率
- ▶ 计算图执行模块：将优化后的计算图中的张量和操作映射到指定设备上具体执行，并给出编程框架的输出结果

2、整体架构



提纲

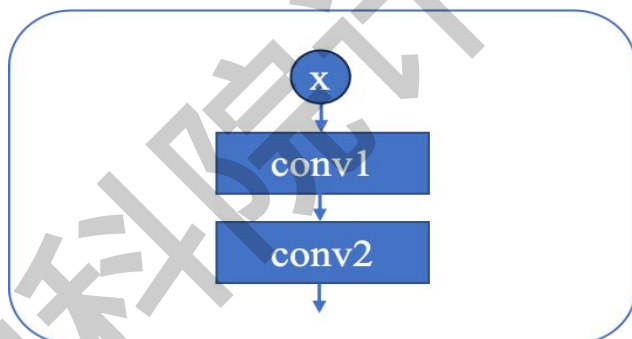
- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练
- ▶ 本章小结

正向图与反向图构建

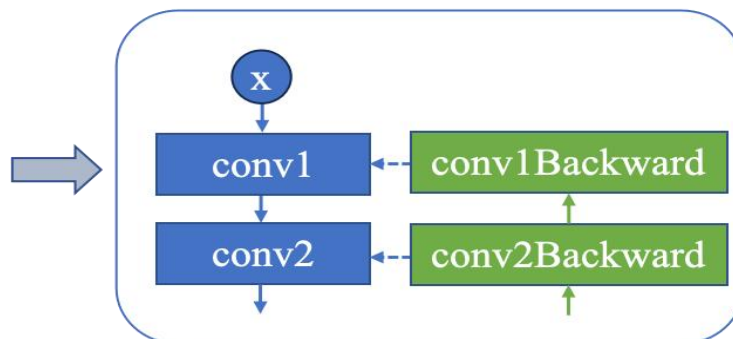
- ▶ 计算图由两个基本元素构成：张量（Tensor）和张量操作（Operation）。计算图是有向图，有向边指明了张量的流动方向

(a) 编写对应程序

```
func layer1(x):  
    x = F.conv2d(x, ...)  
    x = F.relu(x, ...)  
    x = F.conv2d(x, ...)  
    x = F.relu(x, ...)  
    ...
```



(b) 从程序构建正向计算图



(c) 通过自动求导构建反向计算图

1、正向传播

- ▶ 输入张量经过搭建的神经网络层层计算传递，并最终获得计算结果的过程
- ▶ 构建形式
 - ▶ 动态图：在执行函数时，按照函数顺序逐条语句地生成节点，立即计算并返回结果；易调试但性能优化空间有限
 - ▶ 静态图：在执行计算之前构建好所有图上的节点，在图运行时才计算整个计算图并返回最终结果；不易调试但性能好

动态图

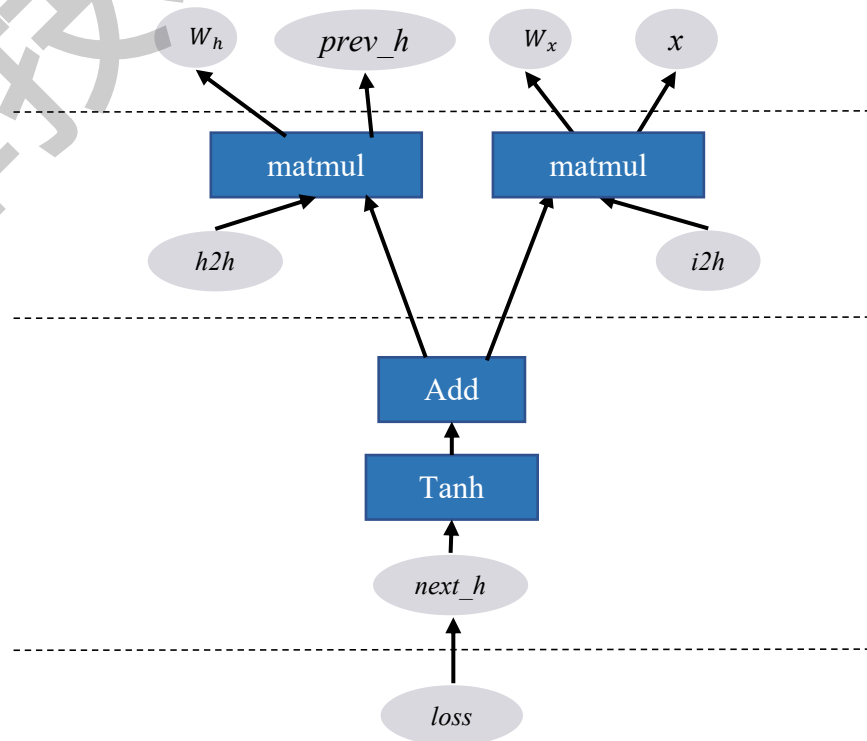
- ▶ 计算图在函数运行过程中逐步构建的 (On-the-fly)
- ▶ 立即 (eager) 模式: 每次调用语句就立刻执行计算
- ▶ PyTorch中的动态图实现: 每次执行, 都会重新被构建

```
W_h = torch.randn(20, 20, requires_grad = True)
W_x = torch.randn(20, 10, requires_grad = True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

```
h2h = torch.matmul(W_h, prev_h.t())
i2h = torch.matmul(W_x, x.t())
```

```
next_h = h2h + i2h
next_h = next_h.tanh()
```

```
loss = next_h.sum()
```

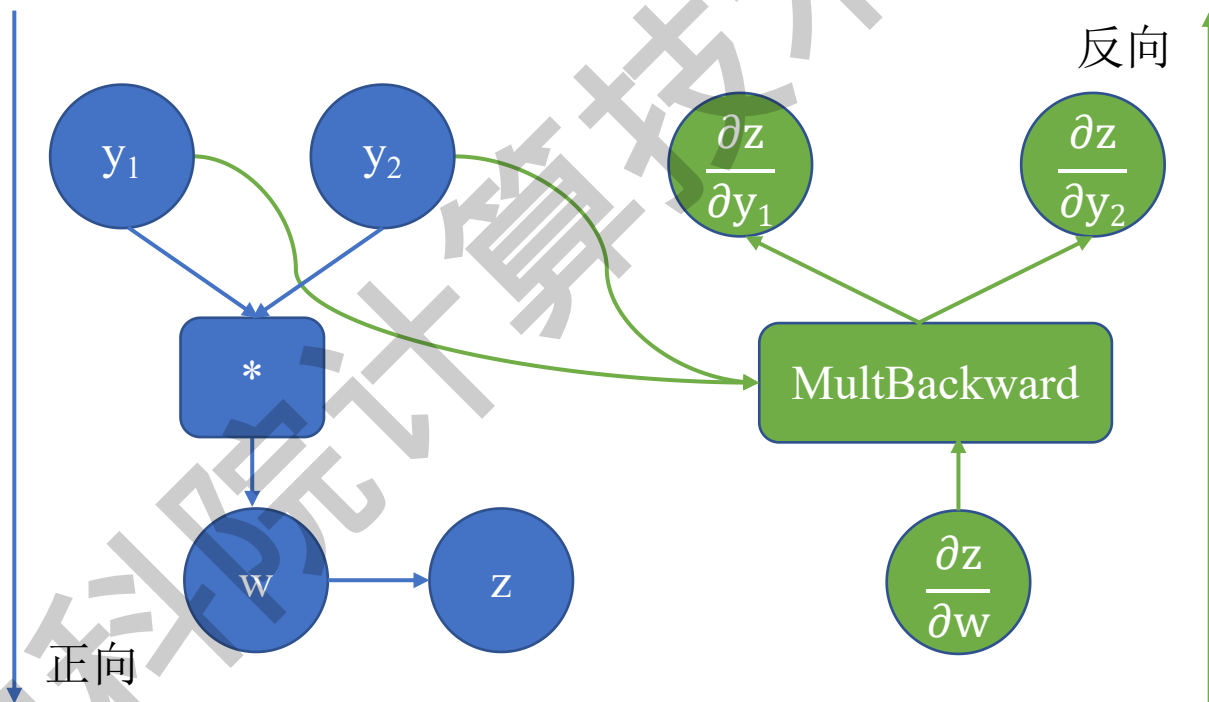


静态图

- ▶ 整个网络的结构会在开始计算前就建立完成计算图
- ▶ 框架执行时接收整个计算图而不是单一语句
- ▶ TensorFlow 1.x中的静态图
 - ▶ 使用若干基本控制流算子（Switch、Merge、Enter、Exit和NextIteration）的不同组合来实现各种复杂控制流场景
- ▶ PyTorch 2.0中的静态图
 - ▶ PyTorch 2.0中采取了图捕获（TorchDynamo）的技术将用户的动态图转化为静态图

2、反向传播

- ▶ 正向计算得到的结果和目标结果存在损失函数值，对其求导得到梯度，并使用该梯度更新参数

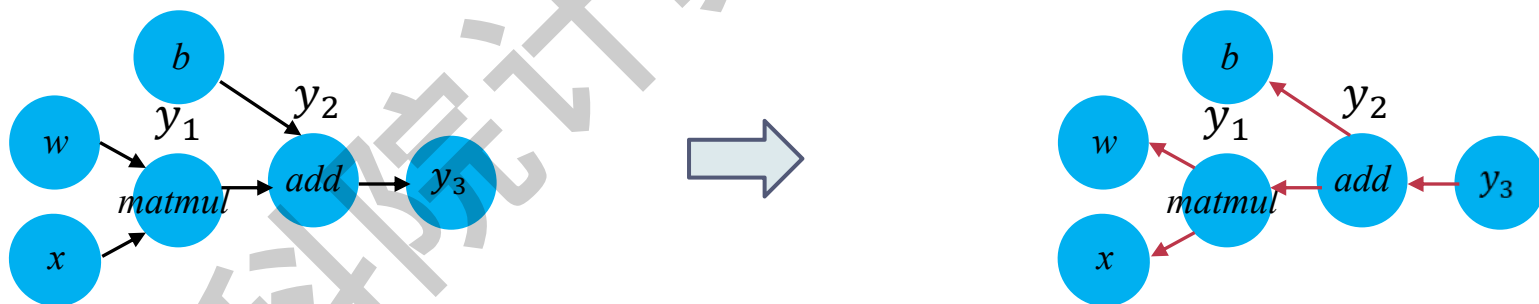


计算导数的方法

- ▶ 自动微分是一种计算导数的方法
- ▶ 常见的求导方式
 - ▶ **手动求导**：用链式法则求解出梯度公式，然后根据公式编写代码、代入数值计算得到梯度结果
 - ▶ **数值求导**：直接代入数值近似求解
 - ▶ **符号求导**：直接对代数表达式求解，最后才代入问题数字，出现表达式膨胀问题
 - ▶ **自动求导**：用户只需描述前向计算的过程，由编程框架自动推导反向计算图，先建立表达式，再代入数值计算

手动求解法

- ▶ 手动用链式法则求解出梯度公式，代入数值，得到最终梯度值
- ▶ 缺点：
 - ▶ 对于大规模的深度学习算法，手动用链式法则进行梯度计算并转换成计算机程序非常困难
 - ▶ 需要手动编写梯度求解代码，且模型变化，算法也需要修改



前向传播 → 反向传播

数值求导法

- ▶ 利用导数的原始定义求解

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

- ▶ 优点：
 - ▶ 易操作
 - ▶ 可对用户隐藏求解过程
- ▶ 缺点：
 - ▶ 计算量大，求解速度慢
 - ▶ 可能引起舍入误差和截断误差

符号求导法

- ▶ 利用求导规则来对表达式进行自动操作，从而获得导数
- ▶ 常见求导规则：

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$

$$\frac{d}{dx}f(x)g(x) = \left(\frac{d}{dx}f(x)\right)g(x) + f(x)\left(\frac{d}{dx}g(x)\right)$$

$$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

- ▶ 缺点：表达式膨胀问题

n	l_n	$\frac{d}{dx}l_n$ 符号求导结果	$\frac{d}{dx}l_n$ 手动求导结果
1	x	1	1
2	$4x(1-x)$	$4(1-x) - 4x$	$4 - 8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$	$16(1 - 10x + 24x^2 - 16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1 - 42x + 504x^2 - 2640x^3 + 7040x^4 - 9984x^5 + 7168x^6 - 2048x^7)$

表达式膨胀示例

自动求导法

- ▶ 数值求导法：直接代入数值近似求解
- ▶ 符号求导法：直接对代数表达式求解，最后才代入问题数字
- ▶ 自动求导法：介于数值求导和符号求导的方法

对基本算子应用符号求导法

代入数值，保留中间结果

应用于整个函数

▶ 计算分两步执行：

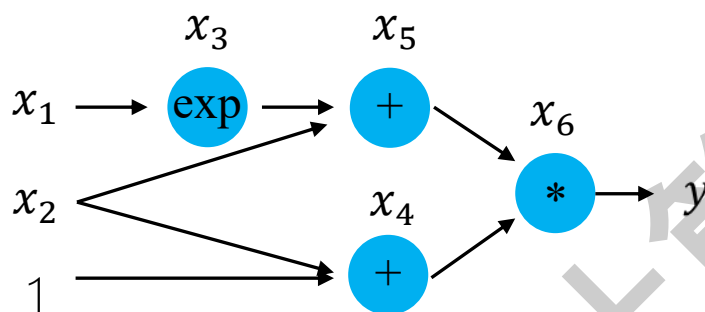
- ▶ 1) 原始函数建立计算图，数据正向传播，计算出中间节点 x_i ，并记录计算图中的节点依赖关系
- ▶ 2) 反向遍历计算图，计算输出对于每个节点的导数

$$\bar{x}_i = \frac{\partial y_j}{\partial x_i}$$

- ▶ 对于前向计算中一个数据(x_i)连接多个输出数据(y_j 、 y_k)的情况，自动求导中，将这些输出数据相对于该数据的导数累加

$$\bar{x}_i = \bar{y}_j \frac{\partial y_j}{\partial x_i} + \bar{y}_k \frac{\partial y_k}{\partial x_i}$$

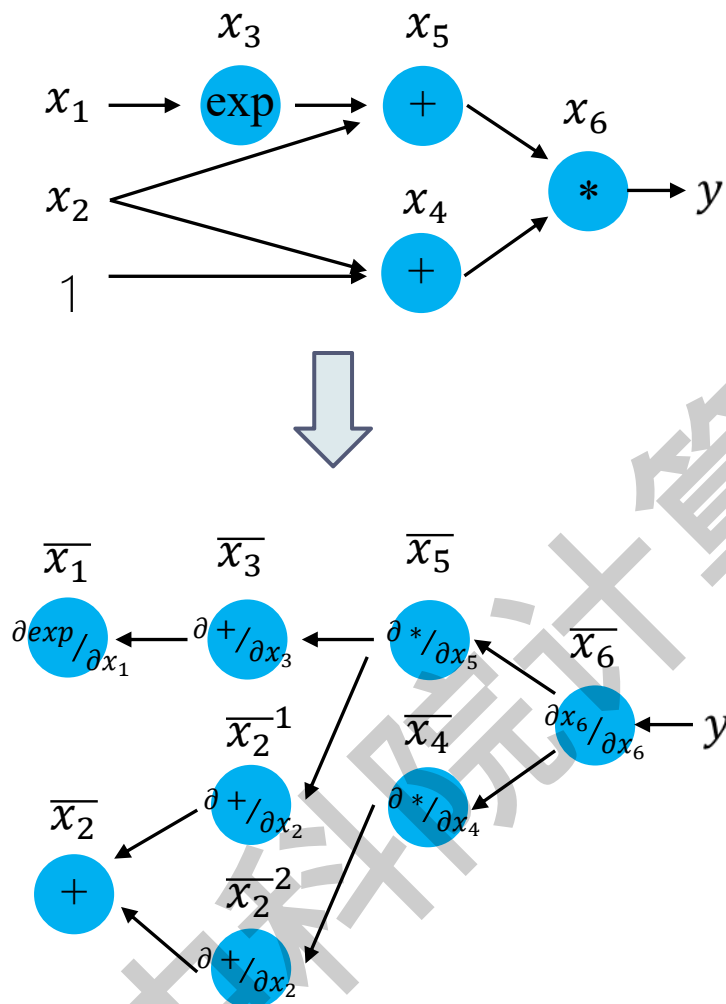
示例--- $f(x_1, x_2) = (e^{x_1} + x_2) (x_2 + 1)$



$$\begin{aligned}x_1 &= 3 \\x_2 &= 2 \\x_3 &= e^{x_1} = 20.086 \\x_5 &= x_3 + x_2 = 22.086 \\x_4 &= x_2 + 1 = 3 \\x_6 &= x_4 * x_5 = 66.258 \\y &= x_6 = 66.258\end{aligned}$$

前向计算

示例--- $f(x_1, x_2) = (e^{x_1} + x_2)(x_2 + 1)$



$$\bar{x}_2 = \bar{x}_2^1 + \bar{x}_2^2 = 25.086$$

$$\bar{x}_1 = \bar{x}_3 * \frac{\partial x_3}{\partial x_1} = \bar{x}_3 * x_3 = 60.258$$

$$\bar{x}_2^2 = \bar{x}_4 * \frac{\partial x_4}{\partial x_2} = \bar{x}_4 * 1 = 22.086$$

$$\bar{x}_2^1 = \bar{x}_5 * \frac{\partial x_5}{\partial x_2} = \bar{x}_5 * 1 = 3$$

$$\bar{x}_3 = \bar{x}_5 * \frac{\partial x_5}{\partial x_3} = \bar{x}_5 * 1 = 3$$

$$\bar{x}_4 = \bar{x}_6 * \frac{\partial x_6}{\partial x_4} = \bar{x}_6 * x_5 = 22.086$$

$$\bar{x}_5 = \bar{x}_6 * \frac{\partial x_6}{\partial x_5} = \bar{x}_6 * x_4 = 3$$

$$\bar{x}_6 = \frac{\partial y}{\partial x_6} = 1$$

反向计算

求导方式对比

方法	对图的遍历次数	精度	备注
手动求解法	NA	高	实现复杂
数值求导法	n_I+1	低	计算量大, 速度慢
符号求导法	NA	高	表达式膨胀
自动求导法	n_O+1	高	对输入维度较大的情况优势明显

其中:

n_I : 要求导的神经网络层的输入变量数, 包括 w 、 x 、 b

n_O : 神经网络层的输出个数

PyTorch中的自动求导

- ▶ AutoGrad是PyTorch的自动微分引擎，用户只需要一行代码`tensor.backward()`，即可调用其自动计算梯度并反向传播
- ▶ AutoGrad模块的`backward`函数实现
 - ▶ 1) 正向图解析
 - ▶ 2) 构建反向计算图的节点
 - ▶ 3) 进行反向梯度传播

PyTorch中的自动求导

- ▶ AutoGrad模块的backward函数实现
 - ▶ 1) 正向图解析

```
// 创建根节点和梯度的列表，并预分配num_tensors大小的空间
std::vector<Edge> roots; // 反向传播根节点集合
variable_list grads; // 反向传播的梯度集合

for (int i = 0; i < num_tensors; i++) {
    // 获取正向图的输出张量
    at::Tensor tensor = py::handle(PyTuple_GET_ITEM(tensors, i)).cast<at::Tensor>();
    // 获取从梯度函数指向输出结果的边
    auto gradient_edge = torch::autograd::impl::gradient_edge(tensor);
    roots.push_back(std::move(gradient_edge)); //将获取的边加入到根节点集合中
    at::Tensor grad_tensor = py::handle(PyTuple_GET_ITEM(grad_tensors, i)).cast<at::Tensor>();
    auto grad_var = torch::autograd::make_variable(grad_tensor);
    grads.push_back(grad_var); // 将梯度变量加入到梯度集合中
}
```

PyTorch中的自动求导

- ▶ AutoGrad模块的backward函数实现
 - ▶ 2) 构建反向计算图的节点

```
std::vector<Edge> output_edges;//反向计算图中所有边的集合
if (inputs != nullptr) {
    for (int i = 0; i < num_inputs; ++i) { // 初始化列表
        ...
        const auto output_nr = tensor.output_nr();
        auto grad_fn = tensor.grad_fn();
        if (!grad_fn) { // 没梯度函数, 则标记是叶子节点
            output_edges.emplace_back(std::make_shared<Identity>(), 0);
        } else { // 有梯度函数, 创建梯度函数指向该节点的边 (构造反向计算图)
            output_edges.emplace_back(grad_fn, output_nr);
        }
    }
}
```

PyTorch中的自动求导

- ▶ AutoGrad模块的backward函数实现
 - ▶ 3) 进行反向梯度传播

```
// 反向计算图已经构建完成，可进行执行
// roots中包含了反向传播根节点
// grads中包含了反向传播产生的梯度，output_edges中是构建的反向计算图的边
variable_list outputs;
{
    pybind11::gil_scoped_release no_gil;
    auto& engine = python::PythonEngine::get_python_engine();
    // 进入引擎执行
    outputs = engine.execute(roots, grads, keep_graph, create_graph, accumulate_grad,
                            output_edges);
}
```

提纲

- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练
- ▶ 本章小结

计算图执行

- ▶ 将计算图中的张量和操作（本节又称算子）映射到给定设备上具体执行
- ▶ 设备管理
- ▶ 张量实现
- ▶ 算子执行
 - ▶ 获取算子执行序列
 - ▶ 实现算子：前端定义、后端实现、前后端绑定
 - ▶ 查找并调用算子

1、设备管理

- ▶ 设备是编程框架中计算图执行时的硬件实体，每个设备都具体负责计算子图中的张量存放和算子运算
- ▶ 常见设备包括通用处理器（如CPU）和领域专用处理器（如GPU和DLP等）
- ▶ 添加对领域专用处理器的设备管理支持（三个模块）
 - ▶ 设备操作
 - ▶ 执行流管理
 - ▶ 事件管理

PyTorch中的设备类型

- ▶ PyTorch中的设备被直接按照类型分类，例如CPU, CUDA, DLP等
- ▶ 通过索引表示特定设备，设备索引唯一，在有多个特定类型的设备时标识特定的计算设备

```
DeviceType parse_type(const std::string& device_string) {  
    static const std::array<  
        std::pair<const char*, DeviceType>,  
        static_cast<size_t>(DeviceType::COMPILE_TIME_MAX_DEVICE_TYPES)>  
        types = {{  
            {"cpu", DeviceType::CPU},  
            {"cuda", DeviceType::CUDA},  
            {"ipu", DeviceType::IPU},  
            {"xpu", DeviceType::XPU},  
            {"mkldnn", DeviceType::MKLDNN},  
            {"opengl", DeviceType::OPENGL},  
            {"opencl", DeviceType::OPENCL},  
            {"ideep", DeviceType::IDEEP},  
            {"hip", DeviceType::HIP},  
            {"ve", DeviceType::VE},  
            {"fpga", DeviceType::FPGA},  
            {"ort", DeviceType::ORT},  
            {"xla", DeviceType::XLA},  
            {"lazy", DeviceType::Lazy},  
            {"vulkan", DeviceType::Vulkan},  
            {"mps", DeviceType::MPS},  
            {"meta", DeviceType::Meta},  
            {"hpu", DeviceType::HPU},  
            {"mtia", DeviceType::MTIA},  
            {"privateuseone", DeviceType::PrivateUse1},  
        }};  
}
```

设备管理

- ▶ 在pytorch/c10/core/impl/DeviceGuardImplInterface.h中定义了抽象的设备管理类DeviceGuardImplInterface
- ▶ 设备操作
 - ▶ 初始化设备运行环境、获取设备句柄和关闭并释放设备等
- ▶ 执行流管理：设备上抽象出来的管理计算任务的软件概念
 - ▶ 在异构编程模型下，完成设备上任务执行的下发和同步操作
 - ▶ 执行流创建、执行流同步和执行流销毁等
- ▶ 事件管理
 - ▶ 表示设备上任务运行的状态和进展
 - ▶ 事件创建、事件记录和事件销毁等基本操作

设备管理

```
struct C10_API DeviceGuardImplInterface {
    ...
    // 设备相关函数（设备是计算资源的抽象，可以是CPU、GPU、DLP等）
    virtual DeviceType type() const = 0; // 设备类型定义
    virtual Device exchangeDevice(Device) const = 0; // 将当前设备设置为指定设备，并返回之前的设备
    virtual Device getDevice() const = 0; // 获得当前设备标识符
    virtual void setDevice(Device) const = 0; // 设置当前上下文所使用的设备，所有被调用设备接口开始之前需要调用本接口
    virtual DeviceIndex deviceCount() const noexcept = 0; // 获取系统内的设备数量

    // 执行流相关函数（执行流是设备用于执行任务的队列，用于设备上任务执行的下发和同步操作）
    virtual Stream getStream(Device) const noexcept = 0; // 得到当前设备的执行流，
    virtual Stream getDefaultStream(Device) const {...} // 得到当前设备的默认执行流
    virtual bool queryStream(const Stream& /*stream*/) const {...} // 执行流查询操作，如果异步执行流的任务全部执行完成则返回为真
    virtual void synchronizeStream(const Stream& /*stream*/) {...} // 执行流同步操作，以等到执行流中的计算任务全部结束

    // 事件相关函数（事件是设备在软件层面抽象出的概念，用来监控设备上任务运行的状态和进展）
    void record(void** /*event*/, const Stream& /*stream*/, // 在给定设备上，把事件加入到执行流中
               const DeviceIndex /*device_index*/, const EventFlag /*flag*/) const override {...}
    void block(void* /*event*/, const Stream& /*stream*/) const override {...} // 事件阻塞操作，用于阻塞主机端线程直到事件被完成
    bool queryEvent(void* /*event*/) const override {...} // 事件查询操作，用于在执行流中查询事件是否已执行完成
    void destroyEvent(void* /*event*/, const DeviceIndex /*device_index*/) const noexcept override {} // 事件销毁操作，回收事件资源

    virtual ~DeviceGuardImplInterface() = default;
};
```

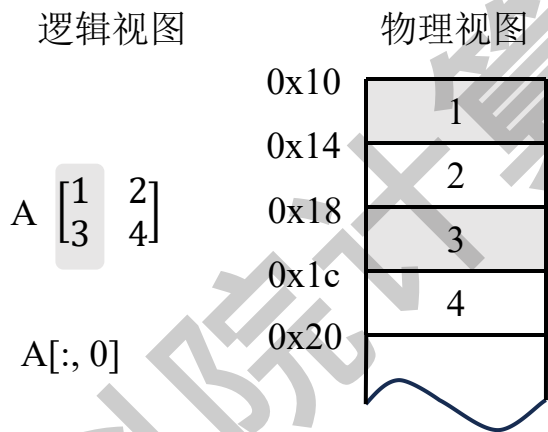
2、张量实现

- ▶ 逻辑视图：形状、布局、步长、偏移量、数据类型和设备等。是框架使用者能直接控制和表达的基本属性
- ▶ 物理视图：设备上的物理地址空间大小、指针、数据类型等。对框架使用者不可见

	属性名	示例	备注
逻辑视图	size	(D, H, W)	维度
	stride	$(1, D, D * H)$	步长
	offset	0	存储位置的偏移
	datatype	float	数据类型
	device	cpu	设备类型
	layout	"NHWC"	布局信息
物理视图	data_ptr	(cpu, 0x1234, ...)	存储在设备内存上的地址
	size	$D * H * W$	存储长度
	datatype	float	在设备上的实际存储类型

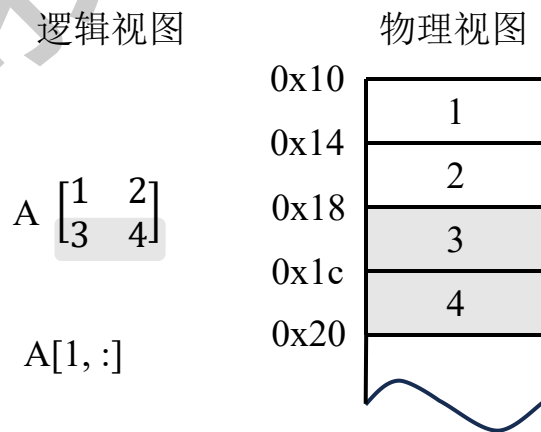
张量数据结构

- ▶ A的逻辑视图是一个形状为[2,2]的张量，物理视图是物理地址空间中从0x10位置开始连续存储的一块数据
- ▶ 逻辑视图通过偏移量和步长来确定物理视图中物理地址空间的寻址空间
- ▶ 一个物理视图可以对应多个逻辑视图：切片的结果不是新的物理视图，而是原本物理视图下的一个新的逻辑视图



大小: 2
步长: 2
偏移量: 无

a) 列切片

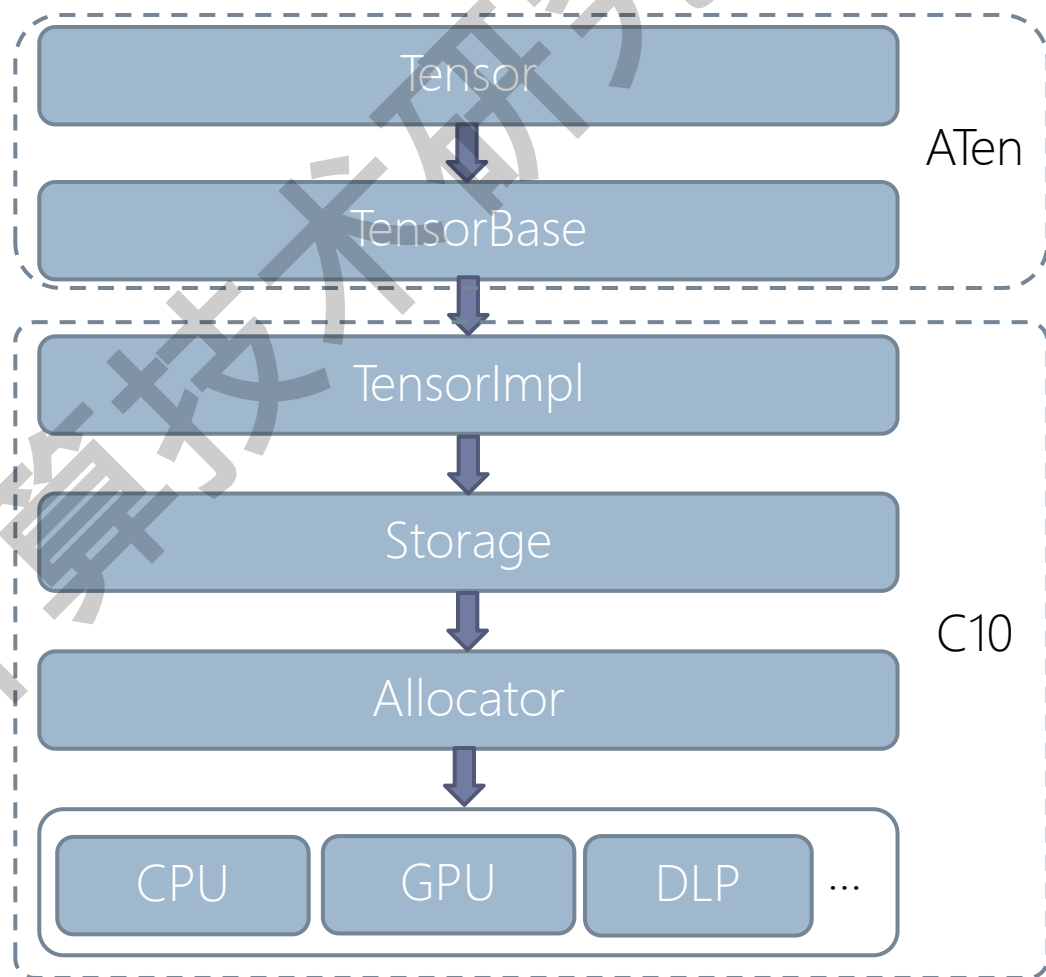


大小: 2
步长: 1
偏移量: 2

b) 行切片

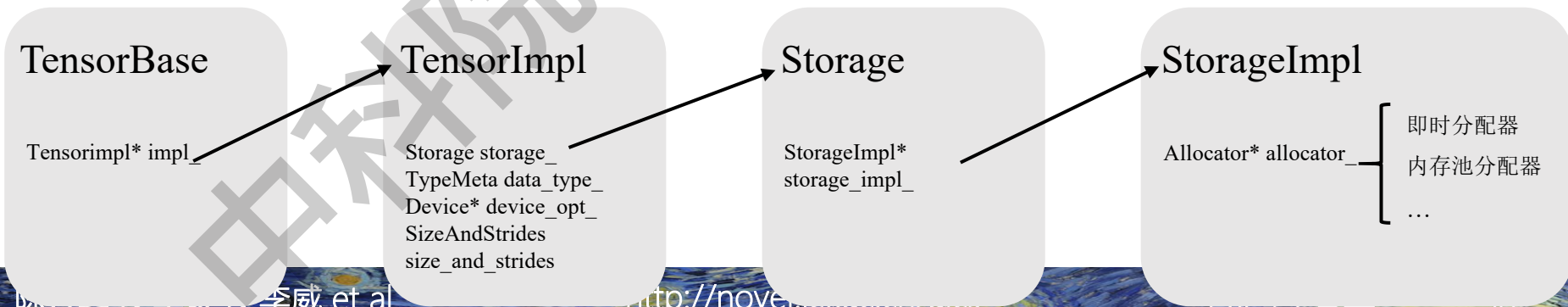
PyTorch中的张量抽象

- ▶ PyTorch中存在与张量对应的类Tensor
- ▶ 持有一个指向底层TensorImpl对象的指针



PyTorch中的张量抽象

- ▶ 通过张量 (Tensor) 抽象类和存储 (Storage) 抽象类来分别表示张量数据结构中的逻辑视图和物理视图
 - ▶ TensorImpl类：张量抽象的实现，包含了维度信息，步长信息，数据类型，设备，布局等**逻辑视角**的张量信息
 - ▶ StorageImpl类：张量的存储实现，包含了内存指针、数据总数等**物理视角**的张量信息，调用结构体Allocator进行张量数据空间的分配



张量内存分配

- ▶ 从逻辑视图到物理视图的转换需要完成对张量的内存分配，即对张量进行内存管理
- ▶ 根据设备的类型不同，张量管理的方式不同
 - ▶ 即时分配---CPU
 - ▶ 内存池分配---GPU

张量内存分配--即时分配

- ▶ 每当需要分配张量的内存时，就立即从系统中申请一块合适大小的内存空间
- ▶ 代码核心部分：malloc()和free()函数

```
struct C10_API DefaultCPUAllocator final : at::Allocator {
    DefaultCPUAllocator() = default;

    // 覆盖基类的分配函数，用于分配n字节的内存空间
    at::DataPtr allocate(size_t nbytes) const override {
        void* data = nullptr;
        ...
        data = malloc(nbytes); // 封装的alloc_cpu()函数进行内存申请，我们在这里进行了简化
        ...
        return {data, data, &ReportAndDelete, at::Device(at::DeviceType::CPU)};
    }

    // 内部实现的上报信息及释放内存函数
    static void ReportAndDelete(void* ptr) {
        ...
        free(ptr); // 释放此前分配的内存
    }
    ...
}
```

张量内存分配--内存池分配

- ▶ 预先分配一块固定大小的内存池，然后在需要时从内存池中分配内存
- ▶ 自我维护：内存块的拆分和合并
- ▶ 优点：节约设备内存使用，减少设备内存碎片化

```
class NativeCachingAllocator : public CUDAAllocator {
private:
    // 互斥锁，用于多线程时锁定哈希表防止竞争
    std::mutex mutex;

    // 重要：哈希表，用于记录分配的指针
    ska::flat_hash_map<void*, Block*> allocated_blocks;

    // 添加分配的内存块，添加时需要上锁
    void add_allocated_block(Block* block) {
        std::lock_guard<std::mutex> lock(mutex);
        allocated_blocks[block->ptr] = block;
    }

public:
    // 获取一个已经分配的内存块的指针，可附带执行删除操作
    Block* get_allocated_block(void* ptr, bool remove = false) {
        std::lock_guard<std::mutex> lock(mutex);
        auto it = allocated_blocks.find(ptr);
        ...
        Block* block = it->second;
        if (remove) {
            allocated_blocks.erase(it);
        }
        return block;
    }
    ...
};
```

张量初始化

- ▶ 在CPU上创建一个空张量
 - ▶ 选择分配器 → 创建StorageImpl类 → 创建TensorImpl类

```
TensorBase empty_cpu(IntArrayRef size, ScalarType dtype, bool pin_memory, c10::optional<c10::MemoryFormat> memory_format_opt) {
    auto allocator = GetCPUAllocatorMaybePinned(pin_memory);
    constexpr c10::DispatchKeySet cpu_ks(c10::DispatchKey::CPU);
    return empty_generic(size, allocator, cpu_ks, dtype, memory_format_opt);
}

TensorBase empty_generic(IntArrayRef size, c10::Allocator* allocator, c10::DispatchKeySet ks, ScalarType scalar_type,
    c10::optional<c10::MemoryFormat> memory_format_opt) {
    return _empty_generic(size, allocator, ks, scalar_type, memory_format_opt);
}

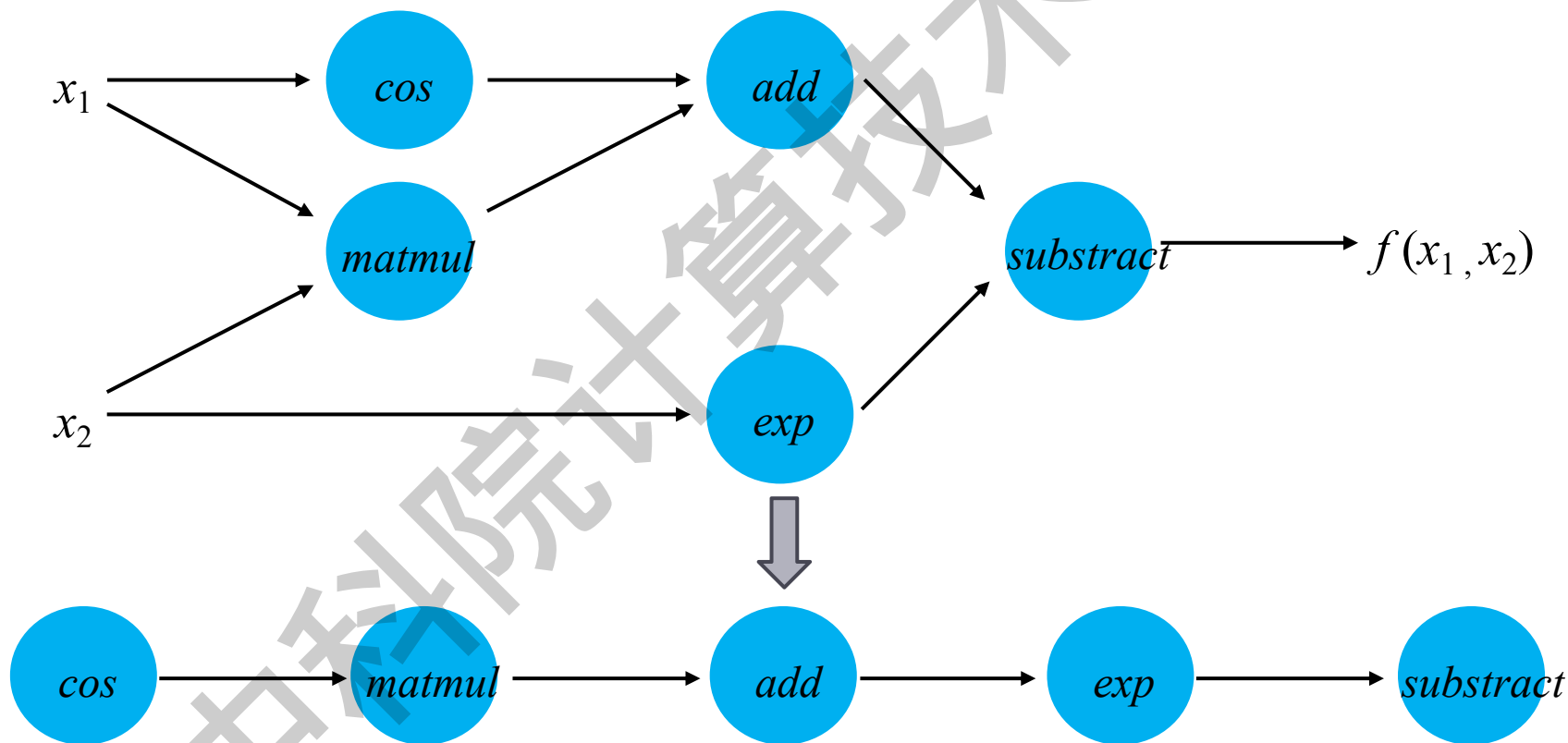
template <typename T>
// 通用分配函数，通过传入的分配器和分派键集合为不同平台分配
TensorBase _empty_generic(ArrayRef<T> size, c10::Allocator* allocator, c10::DispatchKeySet ks, ScalarType scalar_type,
    c10::optional<c10::MemoryFormat> memory_format_opt) {
    ...
    // 创建StorageImpl实例
    auto storage_impl = c10::make_intrusive<StorageImpl>(c10::StorageImpl::use_byte_size_t(), size_bytes,
        allocator, /*resizeable=*/true);
    // 将StorageImpl的所有权交给新创建的TensorImpl
    auto tensor = detail::make_tensor_base<TensorImpl>(std::move(storage_impl), ks, dtype);
    ... // 设置相关信息
    return tensor;
}
```

3、算子执行

- ▶ 计算图的执行过程 = 每个算子独立执行的过程
- ▶ 计算图 → 执行序列（确保正确的数据流和依赖关系）
- ▶ 针对每个算子进行算子实现：前端定义、后端实现和前后端绑定
- ▶ 分派执行：查找适合给定输入的算子实现，并调用相应的实现来执行具体的计算任务

执行序列

- ▶ 分析计算图节点之间的依赖关系 → 执行序列
- ▶ 拓扑排序算法（可有多种可行的结果）



算子实现

- ▶ 正向传播实现和反向传播实现分离
- ▶ 用户接口（前端）和具体实现（后端）分离
- ▶ 算子实现流程
 - ▶ 前端定义：在编程框架中配置算子信息，包含算子的输入、输出以及相关的接口定义，最后生成前端接口（如Python API）
 - ▶ 后端实现：使用C++或其他高级的编程语言，编写算子的底层实现代码，完成算子的计算逻辑部分实现
 - ▶ 前后端绑定：编程框架将前端定义的算子与后端的具体实现进行绑定

native_function模式

- ▶ PyTorch用于管理整个算子实现模块
- ▶ 在使用该模式进行算子实现时，需要修改配置文件 `native_functions.yaml` 以添加算子配置信息
- ▶ native函数格式（位于 `native_functions.yaml`）
 - ▶ `func`字段：定义了算子名称和输入输出的参数类型
 - ▶ `variants`字段：表示需要自动生成的高级方法
 - ▶ `dispatch`字段：表示该算子所支持的后端类型和对应的实现函数

```
- func: func_name(ArgType arg0[=default], ArgType arg1[=default], ...) -> Return
  variants: function, method
  dispatch:
    CPU: func_cpu
    CUDA: func_cuda
```

PReLU算子的native函数

- ▶ PReLU算子实现、PReLU正向传播函数实现和PReLU反向传播函数实现

```
- func: prelu(Tensor self, Tensor weight) -> Tensor  
variants: function, method  
autogen: prelu.out
```

```
- func: _prelu_kernel(Tensor self, Tensor weight) -> Tensor  
dispatch:  
CPU, CUDA: _prelu_kernel  
QuantizedCPU: _prelu_kernel_quantized_cpu  
MkldnnCPU: mkldnn_prelu  
MPS: prelu_mps
```

```
- func: _prelu_kernel_backward(Tensor grad_output, Tensor self, Tensor weight) -> (Tensor,  
Tensor)  
dispatch:  
CPU, CUDA: _prelu_kernel_backward  
MkldnnCPU: mkldnn_prelu_backward  
MPS: prelu_backward_mps
```

前端定义

▶ 前端实现代码

```
class PReLU(Module):
    __constants__ = ['num_parameters']
    num_parameters: int

    def __init__(self, num_parameters: int = 1, init: float = 0.25, device=None, dtype=None) -> None:
        factory_kwargs = {'device': device, 'dtype': dtype}
        self.num_parameters = num_parameters
        super().__init__()
        self.weight = Parameter(torch.empty(num_parameters, **factory_kwargs).fill_(init))

    def forward(self, input: Tensor) -> Tensor:
        return F.prelu(input, self.weight)

    def extra_repr(self) -> str:
        return 'num_parameters={}'.format(self.num_parameters)
```

▶ 在配置文件中添加算子正向传播函数和反向传播函数的对应

```
- name _prelu_kernel (Tensor self, Tensor weight) -> Tensor
  self, weight: "grad.defined() ? _prelu_kernel_backward (grad, self, weight) :
std::tuple<Tensor, Tensor>()"
  result: at::where(self_p >= 0, self_t, weight_p * self_t + weight_t * self_p)
```

后端实现

- ▶ 表层实现：不同设备之间的抽象函数接口
 - ▶ `_prelu_kernel()`和`_prelu_kernel_backward()`
 - ▶ `iter`提供了统一的计算抽象，其封装了前向计算的输入`input`、权重`weight`，以及反向计算的梯度`grad`
 - ▶ 调用`stub`函数进行具体的实现

```
Tensor prelu(const Tensor& self, const Tensor& weight) {  
    ...  
}  
  
Tensor _prelu_kernel(const Tensor& self, const Tensor& weight) {  
    // weight 在 self 上进行广播，并且它们具有相同的数据类型  
    auto result = at::empty_like(self);  
    auto iter = TensorIteratorConfig().add_output(result).add_input(self).add_input(weight).build();  
    prelu_stub(iter.device_type(), iter);  
    return result;  
}  
  
std::tuple<Tensor, Tensor> _prelu_kernel_backward(const Tensor& grad_out, const Tensor& self, const Tensor& weight) {  
    Tensor grad_self = at::empty({0}, self.options());  
    Tensor grad_weight = at::empty({0}, weight.options());  
    auto iter = TensorIteratorConfig().add_output(grad_self).add_output(grad_weight).add_input(self).add_input(weight).add_input(grad_out).build();  
    prelu_backward_stub(iter.device_type(), iter);  
    return {grad_self, grad_weight};  
}
```

后端实现

- ▶ **底层实现**：具体到某个设备上的实际代码实现
 - ▶ 正向prelu_kernel() 反向prelu_backward_kernel()
 - ▶ 这两个函数都利用了SIMD指令实现向量优化
 - ▶ 底层实现中的prelu_kernel和表层实现中的prelu_stub会在前后端绑定中完成对应

```
void prelu_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES_AND2(kBFloat16, kHalf, iter.dtype(), "prelu_cpu", [&]() {
    using Vec = Vectorized<scalar_t>;
    cpu_kernel_vec(iter,
      [](scalar_t input, scalar_t weight) {return (input > scalar_t(0)) ? input : weight * input;},
      [](Vec input, Vec weight) {return Vec::blendv(weight * input, input, input > Vec(0));});
  });
}

void prelu_backward_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES_AND2(kBFloat16, kHalf, iter.dtype(), "prelu_backward_cpu", [&]() {
    cpu_kernel_multiple_outputs(iter, [](scalar_t input, scalar_t weight, scalar_t grad) -> std::tuple<scalar_t, scalar_t> {
      auto mask = input > scalar_t{0};
      auto grad_input = mask ? grad : weight * grad;
      auto grad_weight = mask ? scalar_t{0} : input * grad;
      return {grad_input, grad_weight};
    });
  });
}
```

前后端绑定

- ▶ 同一个算子可能会有多个后端实现的代码
 - ▶ 多种后端 & 多种输入，根据不同情况调用相应的后端实现
 - ▶ PyTorch使用分派机制来管理前后端对应关系，由Dispatcher管理分派表
 - ▶ 分派表的表项记录着算子到具体的后端实现对应关系，纵轴表示PyTorch所支持的算子，横轴表示支持的分派键（与后端相关的标识符）

```
TORCH_LIBRARY_IMPL(aten, CPU, m)  
{  
  m.impl("prelu", cpu_prelu);  
}
```

	CPU	GPU	DLP	...
add				
mul				
prelu	cpu_prelu			
...				

分派执行

- ▶ 获得算子执行序列 → 实现对应算子 → 对算子分派执行
- ▶ 分派执行：在运行时根据输入张量的类型和设备类型查找并调用合适的算子实现方法
- ▶ Dispatcher计算分派键，并由此找到对应的内核函数
 - ▶ 算子：Dispatcher的调度对象，代表了具体的计算任务
 - ▶ 分派键：根据输入张量和其他信息计算，可简单地理解为与硬件平台相关联的标识符
 - ▶ 内核函数：特定硬件平台上实现算子功能的具体代码

提纲

- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练
- ▶ 本章小结

1、为什么需要深度学习编译

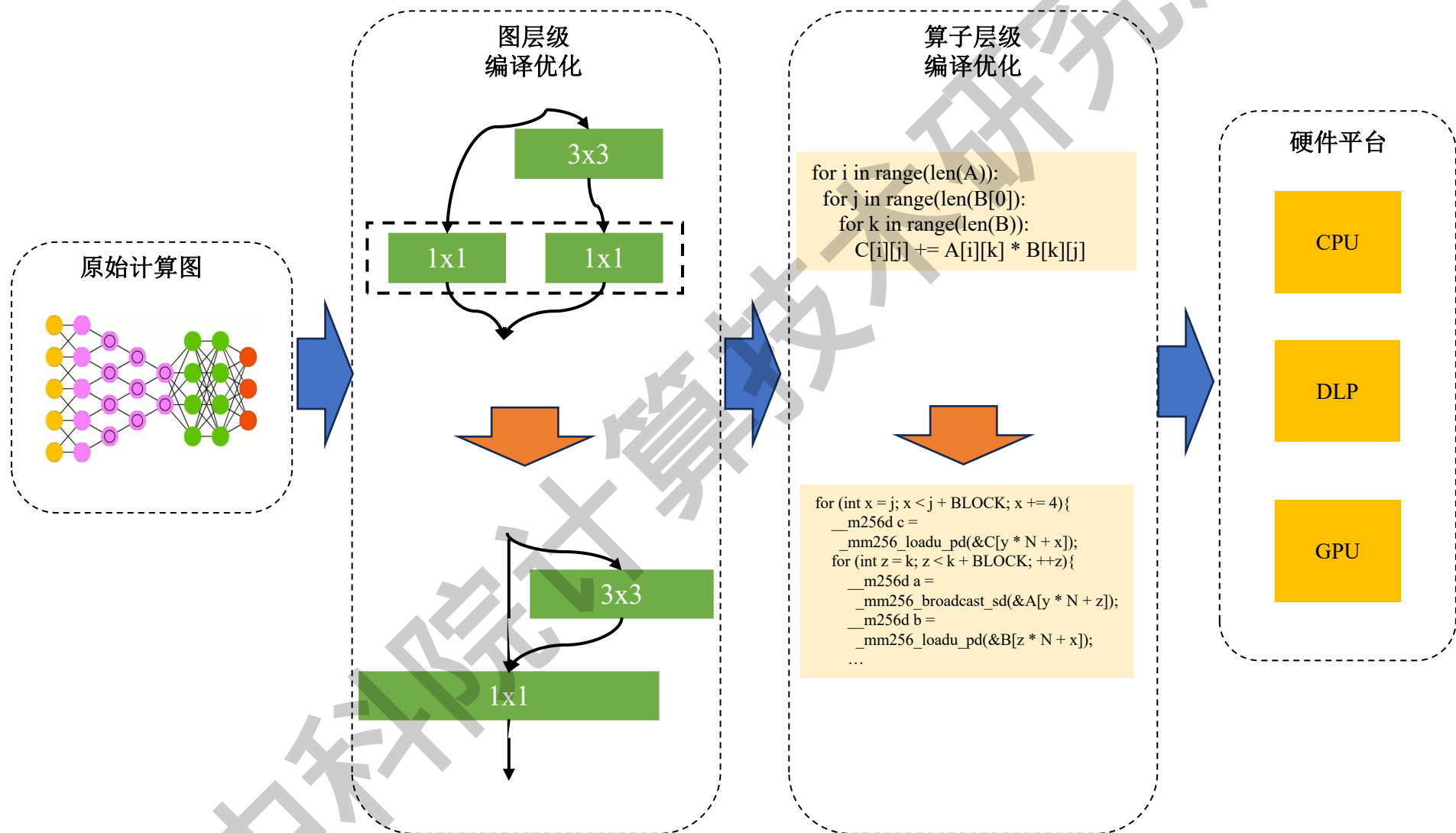
▶ 编程框架中早期优化方式存在的问题

- ▶ 框架维护成本高：对于新硬件和新算子，都需要程序员手动进行算子开发，开发数量呈平方级增长
- ▶ 性能受限：性能受限于程序员人工优化算子的能力，且没有充分探索计算图的优化空间
- ▶ 在深度学习编程框架中引入深度学习编译机制
 - ▶ 减少人工开发工作量：可针对不同硬件平台进行代码生成
 - ▶ 便于性能优化：（图层级）对完整的计算图进行静态分析和全局优化；（算子层级）利用自动调优技术优化算子，最大限度提升硬件利用率

什么是深度学习编译器

- ▶ 接收以计算图形式表示的深度学习任务，并在指定硬件平台上生成高性能代码
- ▶ 多个层级中间表示 & 多个层级优化
 - ▶ 图层级优化：子图替换、常量折叠、公共子表达式删除、布局优化以及算子融合等
 - ▶ 算子层级优化：自动调优，基于搜索的方法和基于多面体模型的方法
- ▶ 常见深度学习框架中所采用的编译技术和深度学习编译器
 - ▶ TVM、TC (Tensor Comprehensions)、XLA、MLIR ...

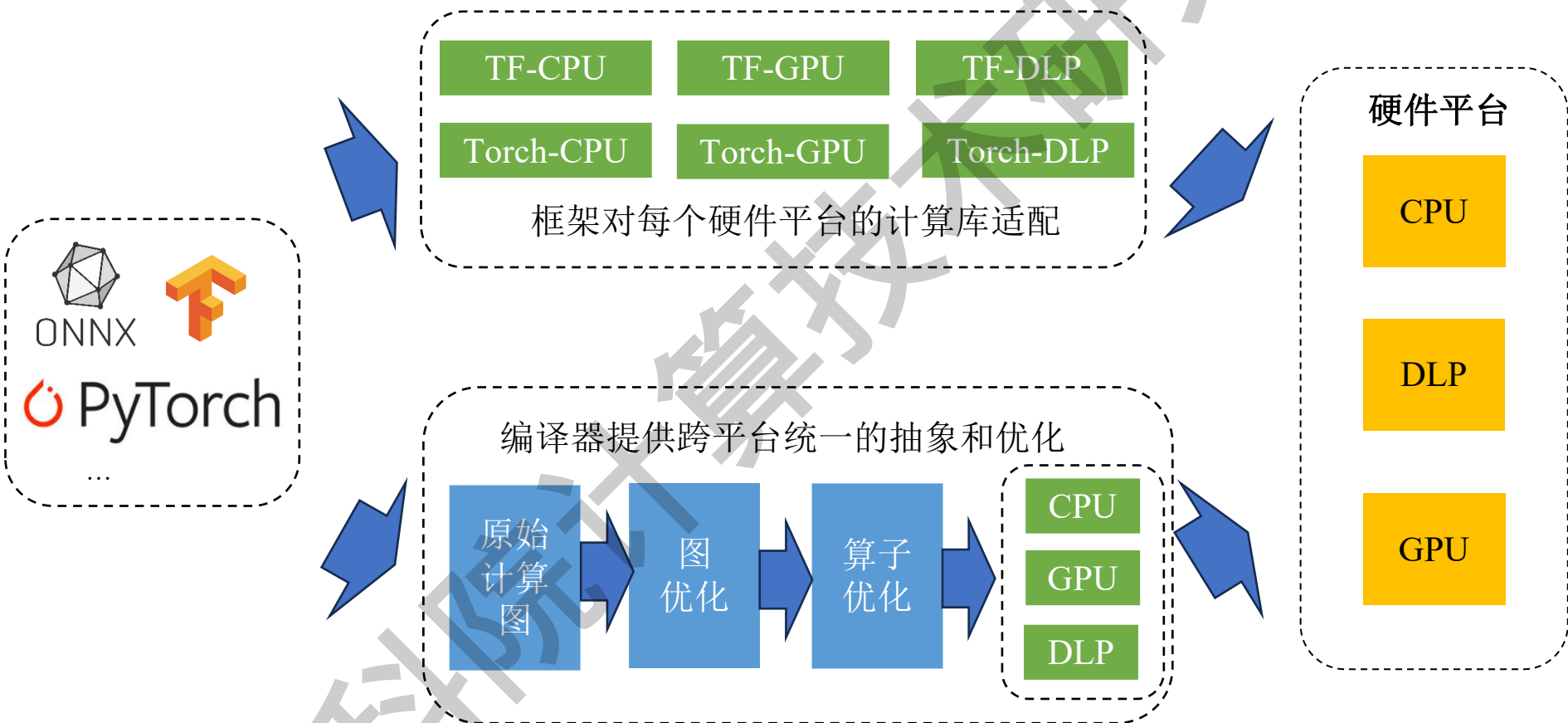
什么是深度学习编译器



深度学习编译器跟编程框架的关系

- ▶ 深度学习编程框架
 - ▶ 自行适配厂商提供的计算库或者手写算子来支持不同硬件，这带来了极高的框架维护成本
- ▶ 深度学习编译器
 - ▶ 提供跨平台统一的抽象和优化，较为灵活的适配不同的上层编程框架和底层硬件平台
 - ▶ 经过图层级优化和算子层级优化后，自动生成在目标硬件平台上的高性能算子

深度学习编译器跟编程框架的关系



2、图层级编译优化

- ▶ 不关心特定算子的具体执行过程，而关心数据在图中的流动过程
- ▶ 图优化方法
 - ▶ 子图替换、常量折叠、公共子表达式删除、布局优化、算子融合...

图优化方法

- ▶ 子图替换：将原计算图中的节点（计算操作）替换为功能等价但运算逻辑更优的形式
- ▶ TensorFlow中人为设定的替换规则

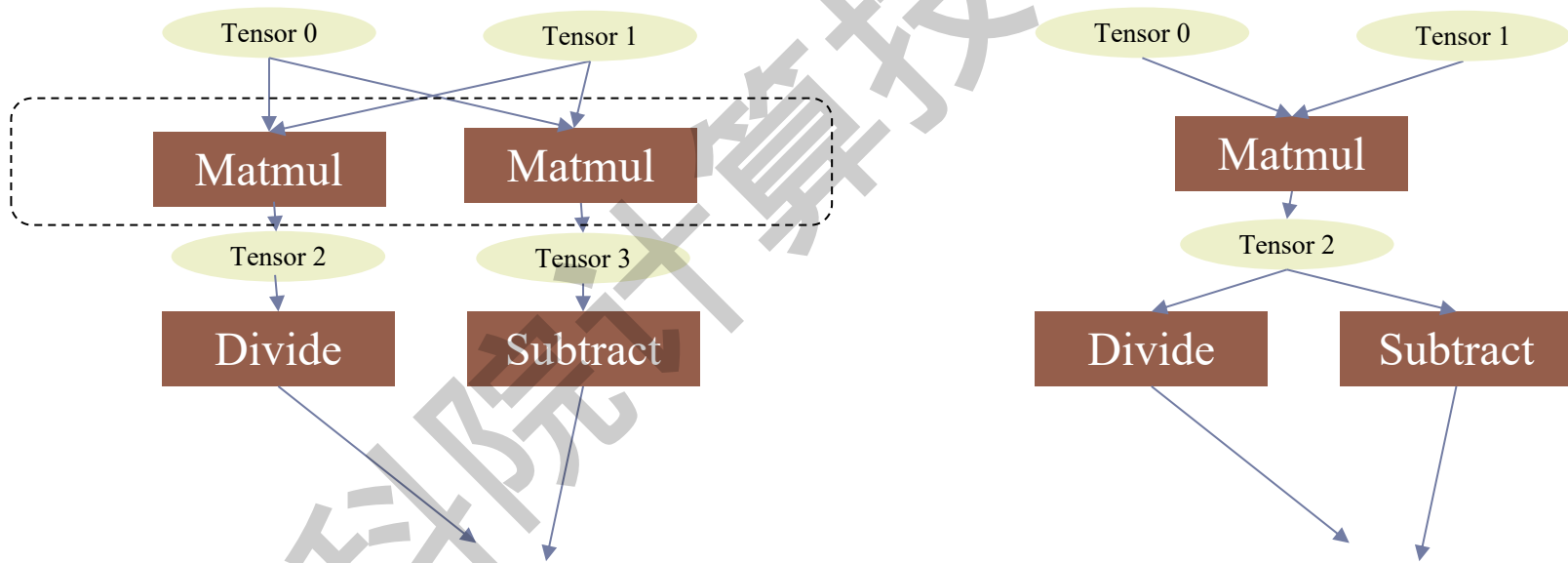
子图替换前	子图替换后
Add(const1, Add(x, const2))	Add(x, const1 + const2)
Conv2D(const1 * x, const2)	Conv2D(x, const1 * const2)
Concat([x, const1, const2, y])	Concat([x, concat([const1, const2]), y])
Matmul(Transpose(x), y)	Matmul(x, y, transpose_x = true)
Cast(Cast(x, dtype1), dtype2)	Cast(x, dtype2)
Reshape(Reshape(x, shape1), shape2)	Reshape(x, shape2)
AddN(x * a, b * x, c * x)	x * AddN(a + b + c)
(matrix1 + scalar1) + (matrix2 + scalar2)	(matrix1 + matrix2) + (scalar1 + scalar2)

图优化方法

▶ 常量折叠:

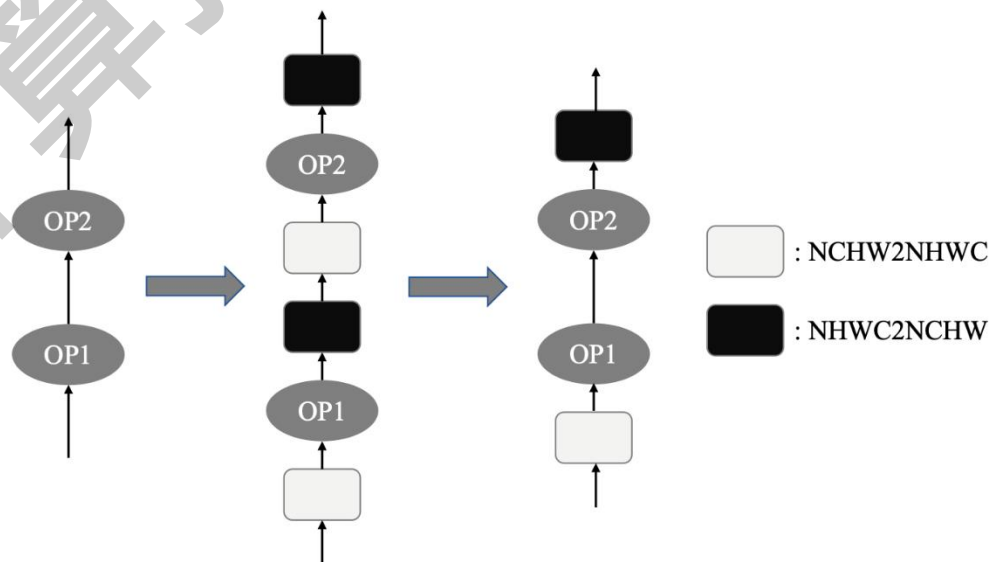
▶ 如 $16 * 16 * 224$ 结果为定值, 则计算其值后带入此定值

▶ 公共子表达式消除



图优化方法

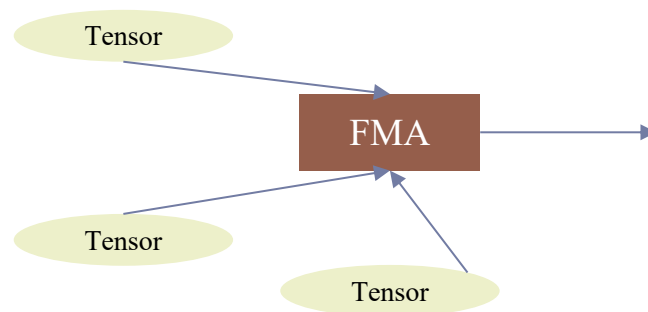
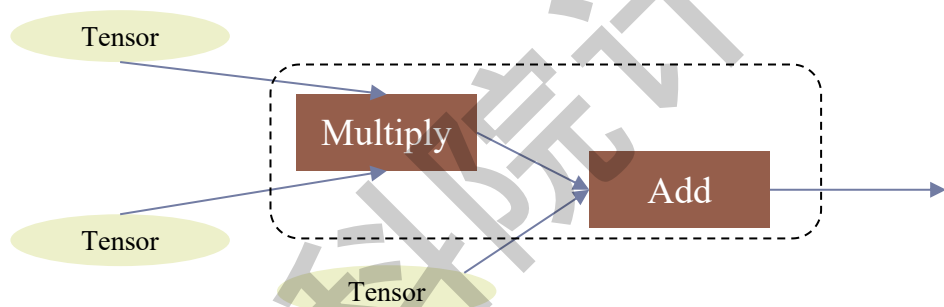
- ▶ 代数化简：将代价高的计算换为等价代价低的运算
 - ▶ 若：表达式中出现乘0，此表达式的结果直接为0
- ▶ 布局优化：输入布局影响执行性能
 - ▶ 使用Tensor Core计算相同输入数据，采用NHWC格式的性能普遍优于NCHW格式



图优化方法

▶ 算子融合：纵向

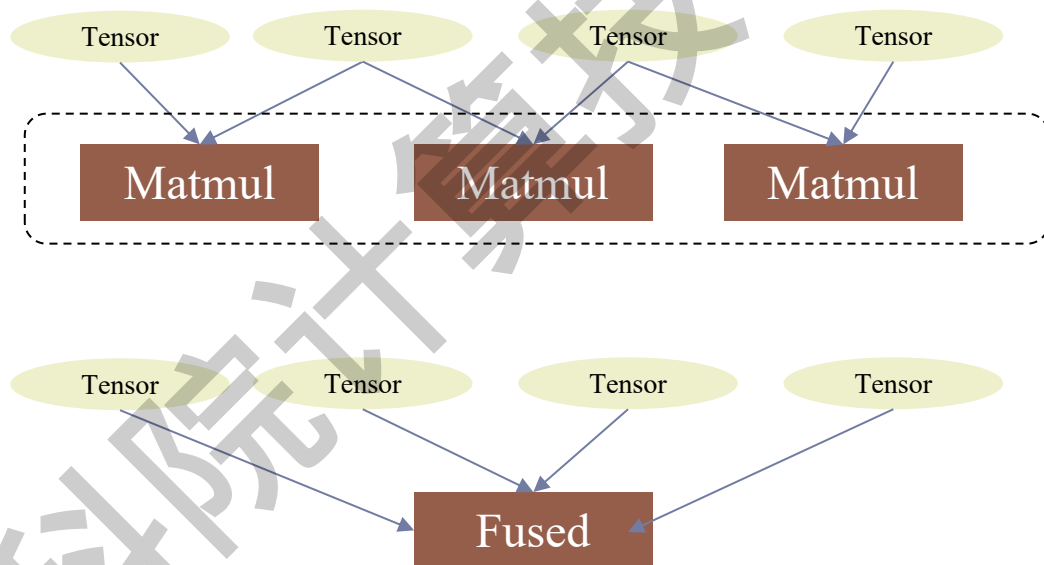
- ▶ 函数调用有开销，外设的函数调用（Kernel Launch）开销巨大
- ▶ 将多个小算子融合为一个大算子进行执行
- ▶ 常见：FMA（Fused Multiply-Add）



图优化方法

▶ 算子融合：横向

- ▶ 多个小的矩阵乘可以合并为一个大的矩阵乘



3、算子层级编译优化

- ▶ 接收图层级编译优化后的计算图节点作为输入，将其下降到算子层级中间表示上，最终生成目标硬件后端上的代码
- ▶ 算子层级中间表示：抽象建模一个计算及其在设备上的具体执行流程
- ▶ 算子调度：针对目标硬件后端上的计算特性和访存特性进行优化
- ▶ 自动调优：自动确定最优的调度配置

算子层级中间表示

- ▶ 计算与调度分离
- ▶ 计算表示涵盖了算子的计算定义信息，但不包括具体的实现信息。包括对张量和对计算本身的描述。

```
A = placeholder((N, L), name = "A", dtype = "float")
B = placeholder((L, M), name = "B", dtype = "float")
k = reduce_axis((0, L), name = "k")
C = compute((N, M), lambda i, j: sum(A[i, k] * B[k, j]), axis = k, name = "C")
```

- ▶ 计算加上调度的表示确定了算子的实现，可以使用嵌套循环程序对其进行表示。

```
for (i: int32, 0, 1024) "parallel" :
  for (j: int32, 0, 1024) :
    C[((i*1024) + j)] = 0f32
    for (k: int32, 0, 1024) :
      let cse_var_2: int32 = (i*1024)
      let cse_var_1: int32 = (cse_var_2 + j)
      C[cse_var_1] = (C[cse_var_1] + (A[(cse_var_2 + k)]*B[((k*1024) + j)]))
```

算子调度

- ▶ 通过循环变换来匹配目标平台的体系结构特性（包括计算特性和访存特性）
- ▶ 算子调度
 - ▶ 算子的具体实现通常表现为嵌套循环程序

```
for (i: int32, 0, 1024) :  
  for (j: int32, 0, 1024) :  
    C[...] = 0f32  
    for (k: int32, 0, 1024) :  
      C[...] = C[...] + A[...] * B[...]
```

- ▶ 循环分块 (tiling) 优化、循环向量化 (Vectorize) 等
- ▶ 优点：提升缓存命中率和（在CPU平台上）使用向量化加速

算子调度

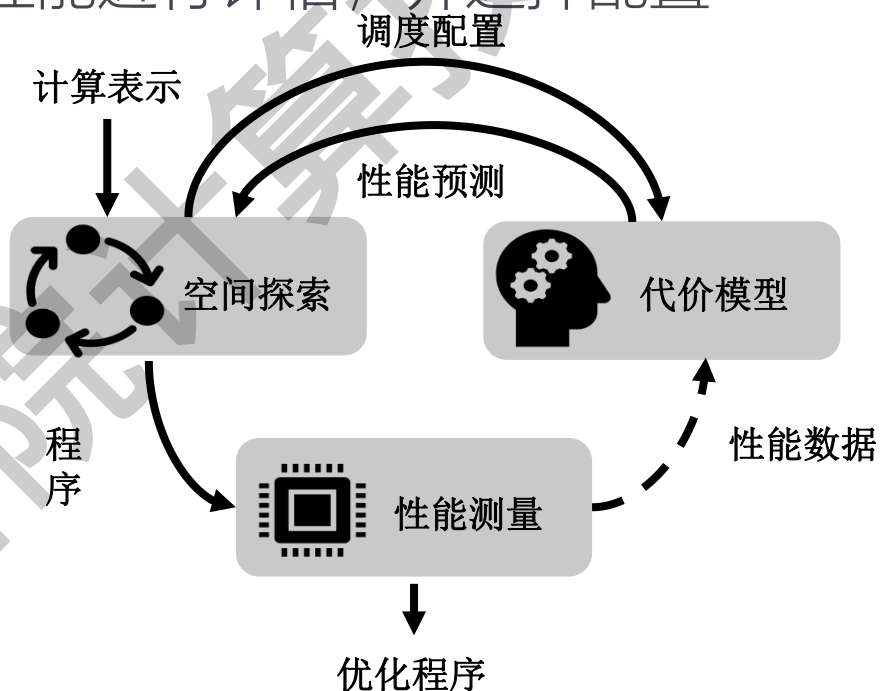
- ▶ 一个完整的调度是由多个调度原语构成的
- ▶ 常见的算子调度原语

名称	描述	形式
split	循环切分	$loop_1, loop_2 = stage.split(loop, factor)$
reorder	循环排序	$stage.reorder([loop_1, \dots, loop_n])$
cache	加入缓存阶段	$cache_read(stage, "shared")$
unroll	循环展开	$stage.unroll(loop, length)$
compute_at	阶段融合	$stage_2.compute_at(stage_1, location)$
vectorize	循环向量化	$stage.vectorize(loop)$
parallel	循环并行	$stage.parallel(loop)$
tensorize	循环张量化	$stage.tensorize(loop, intrin_gemm(m, n, k))$

- ▶ 通过不同调度原语和调度参数的组合，编译器可以构建一个包含海量不同程序重写的优化空间

自动调优

- ▶ 通过**搜索**的方式确定合适的调度配置
 - ▶ 空间探索：一个点代表一种配置
 - ▶ 性能测量：测试某配置下的程序性能
 - ▶ 代价模型：对性能进行评估，并选择配置



自动调优

- ▶ 自动调优的核心是空间搜索
 - ▶ 空间和搜索算法的设计
- ▶ 常见的空间搜索方式
 - ▶ 基于手工模板的搜索
 - ▶ 基于序列构建的搜索
 - ▶ 层次化的搜索方法

基于手工模板的搜索方法

- ▶ 依赖于给定的调度模板
 - ▶ 该模板包括手工设计的原语序列
 - ▶ 该序列通常只有调度参数没有确定
- ▶ 空间较为简单，可用多种搜索算法
 - ▶ 随机搜索、网格搜索、遗传算法...
- ▶ 手工模板的设计需要领域专家进行



参数搜索

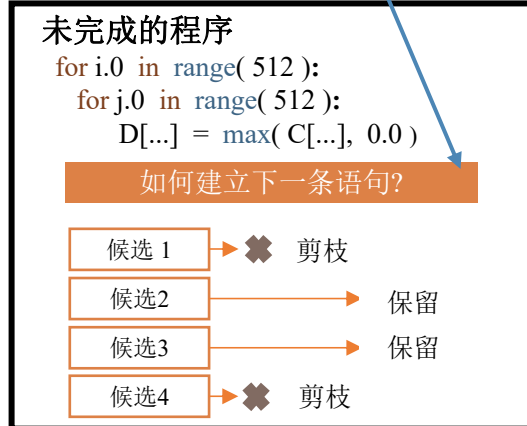
固定人工模板

```
for i.0 in range( ? ):
  for j.0 in range( ? ):
    for k.0 in range( ? ):
      for i.1 in range( ? ):
        for j.1 in range( ? ):
          C[...] += A[...] * B[...]
        for i.1 in range( ? ):
          for j.1 in range( ? ):
            D[...] = max(C[...], 0.0)
```

基于序列构建的搜索方法

- ▶ 逐条循环语句地构建优化程序
 - ▶ 编译器选择合适的调度原语以及调度参数
 - ▶ 使用代价模型进行性能评估
- ▶ 搜索算法存在一定的限制
 - ▶ 随机搜索、集束搜索、蒙特卡洛树搜索...
- ▶ 耗时且低效
 - ▶ 缺少程序优化的先验知识对空间进行有效剪枝

提前终止的集束搜索



层次化构建的搜索方法

▶ 从粗到细的粒度构建优化程序

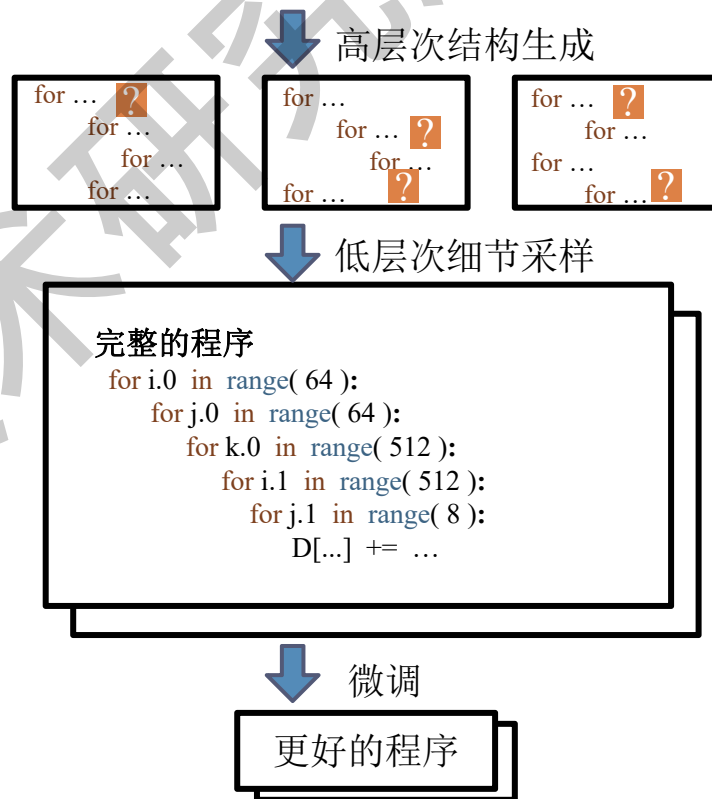
- ▶ 粗粒度：程序所要采用的循环结构
- ▶ 细粒度：具体的调度参数

▶ 搜索算法存在一定的限制

- ▶ 随机搜索、遗传算法...

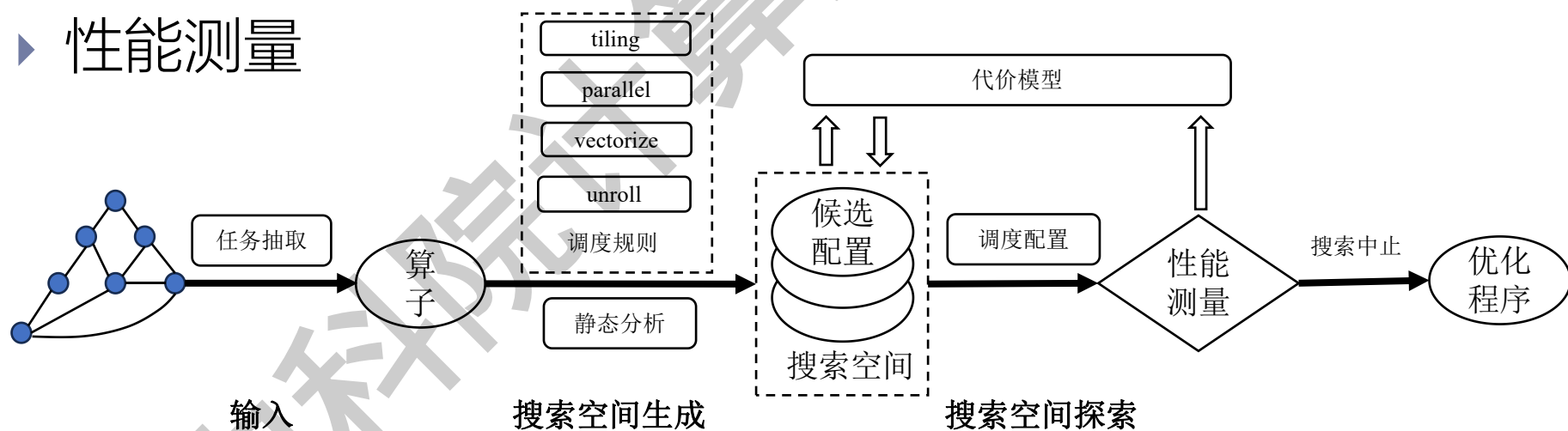
▶ 优缺点

- ▶ 更大的搜索空间
- ▶ 在粗粒度的结构选择策略中引入优化的先验知识
- ▶ 需要领域专家设计



算子层级编译优化的实现

- ▶ 模型输入：将调度该算子的任务分发到搜索空间生成器
- ▶ 搜索空间生成：为算子产生调度序列，及该调度序列所需的参数取值范围
- ▶ 搜索空间探索：通过特定搜索策略选取高性能的算子
- ▶ 性能测量

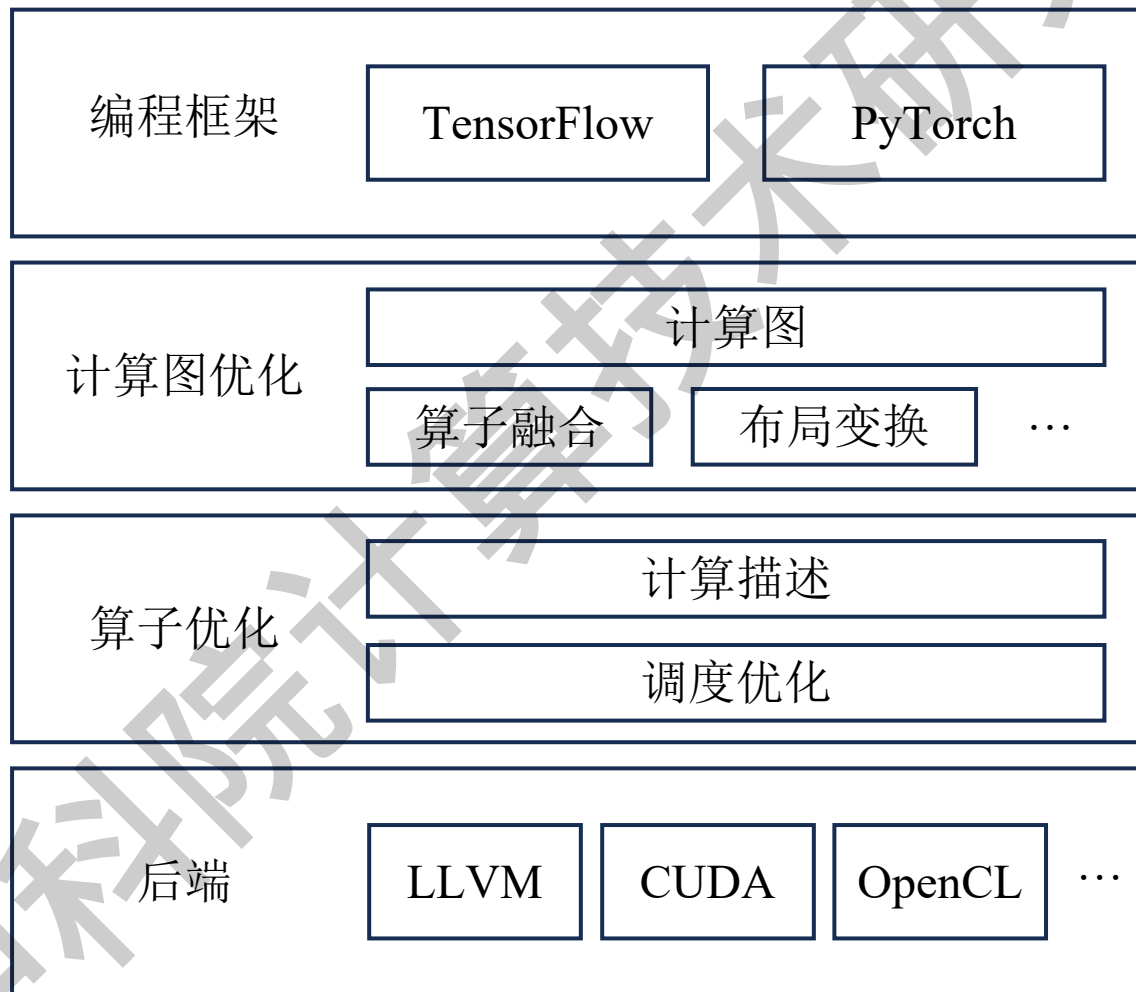


4、常见深度学习编译器介绍

- ▶ TVM
- ▶ Tensor Comprehensions
- ▶ XLA
- ▶ MLIR
- ▶ TorchDynamo和TorchInductor
- ▶ ...

TVM (Tensor Virtual Machine)

- ▶ TVM结构示意图：两个层级的编程抽象



TVM

- ▶ TVM的核心思想：计算与调度分离

- ▶ 计算：定义元素之间的运算关系

- ▶ 调度：规划具体计算执行的运算顺序、数量

```
C = te.compute((M, N), lambda m, n:  
    te.sum(A[m, k] * B[k, n], axis=k,  
    name="C")
```

计算

```
s = te.create_schedule(C.op)  
mo, no, mi, ni = s[C].tile(C.op.axis[0],  
    C.op.axis[1], bn, bn)  
(kaxis,) = s[C].op.reduce_axis  
ko, ki = s[C].split(kaxis, factor=kfactor)
```

调度

- ▶ 两个层级的编程抽象

- ▶ 图层级中间表示

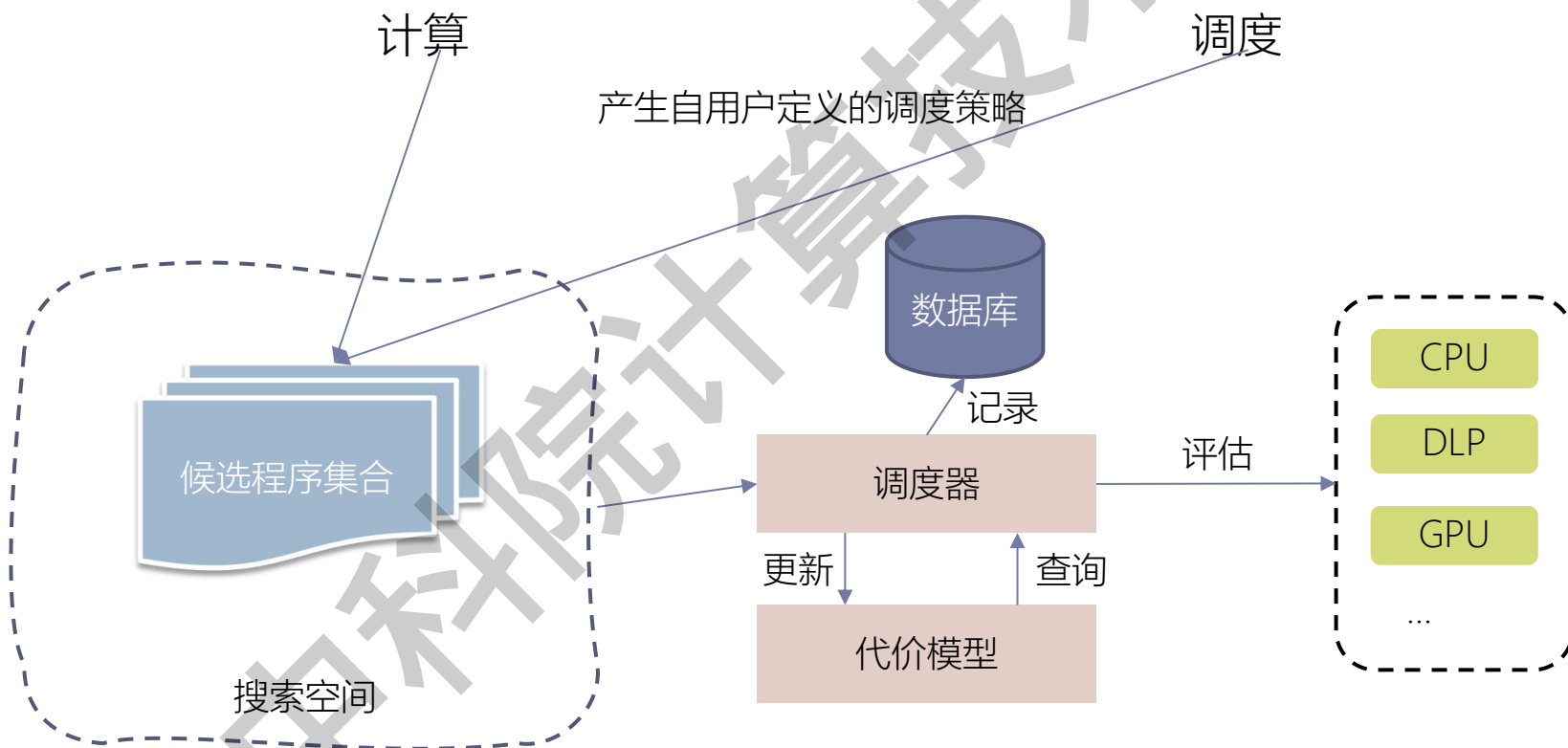
- ▶ 算子层级中间表示

TVM

▶ TVM的自动调优实现

```
C = te.compute((M, N), lambda m, n:  
  te.sum(A[m, k] * B[k, n], axis=k),  
  name="C")
```

```
s = te.create_schedule(C.op)  
mo, no, mi, ni = s[C].tile(C.op.axis[0],  
  C.op.axis[1], bn, bn)  
(kaxis,) = s[C].op.reduce_axis  
ko, ki = s[C].split(kaxis, factor=kfactor)
```



Tensor Comprehensions

- ▶ 自动生成高性能代码Tensor Comprehensions (TC)
 - ▶ 基于多面体模型 (Polyhedral Model) 的即时 (JIT) 编译器
 - ▶ 将DAG转换为具有内存管理和同步的CUDA内核函数
- ▶ 多面体模型技术
 - ▶ 使用一个结构化的方式来捕获和表示循环代码的结构和语义，并可以在这个表示的基础上应用各种优化和变换手段，在保持代码语义不变的基础上提高性能

XLA (Accelerated Linear Algebra)

- ▶ XLA由谷歌开发的，并作为TensorFlow的一部分提供
 - ▶ 对计算图进行优化和编译
 - ▶ 核心：HLO IR（提供细粒度的算子抽象），组合成任意的算子
 - ▶ 提供多种与硬件架构无关的优化：公共子表达式消除、算子融合等
 - ▶ BatchNorm算子，通过一系列的Broadcast、Reduce和Element_wise操作组成
 - ▶ 使用LLVM IR表示CPU和GPU后端

XLA (Accelerated Linear Algebra)

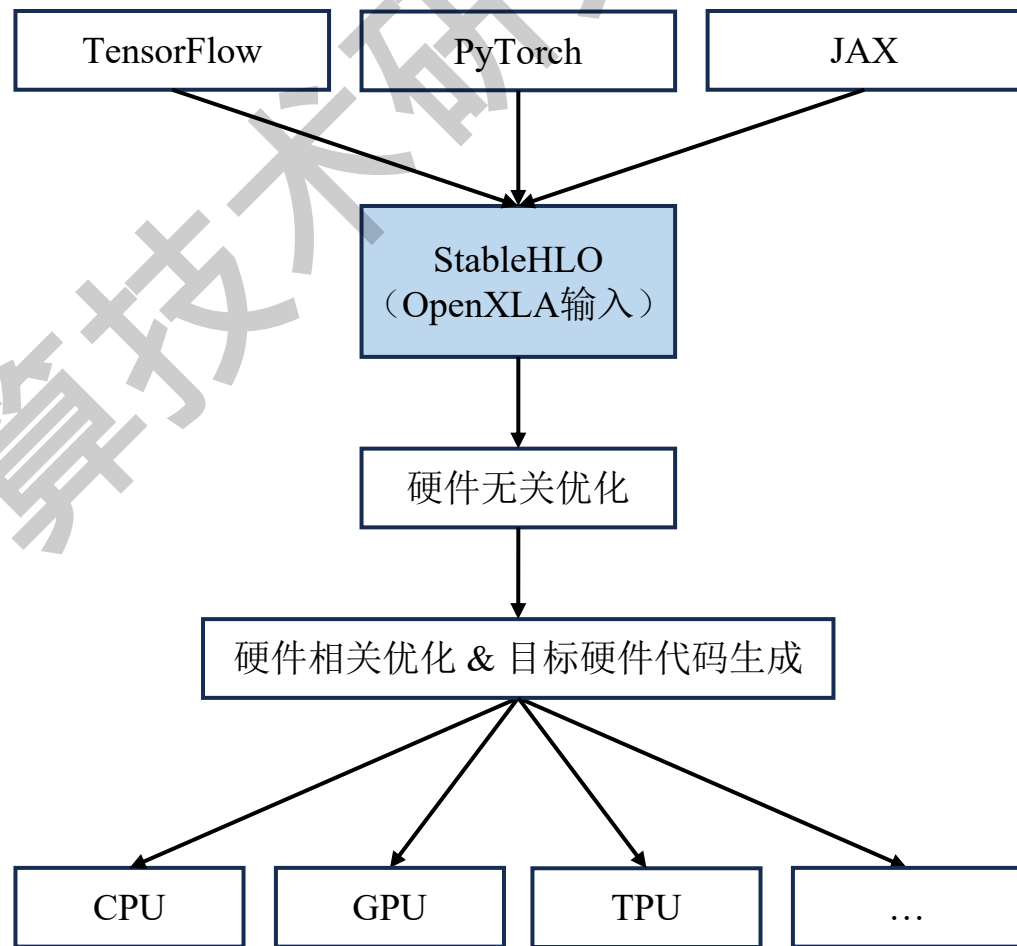
- ▶ OpenXLA项目

- ▶ 源于将编译器相关技术从 TensorFlow 独立

- ▶ 编程框架的输入 →

StableHLO → 硬件无关优化 →

硬件相关优化 → 代码生成



MLIR (Multi-Level Intermediate Representation)

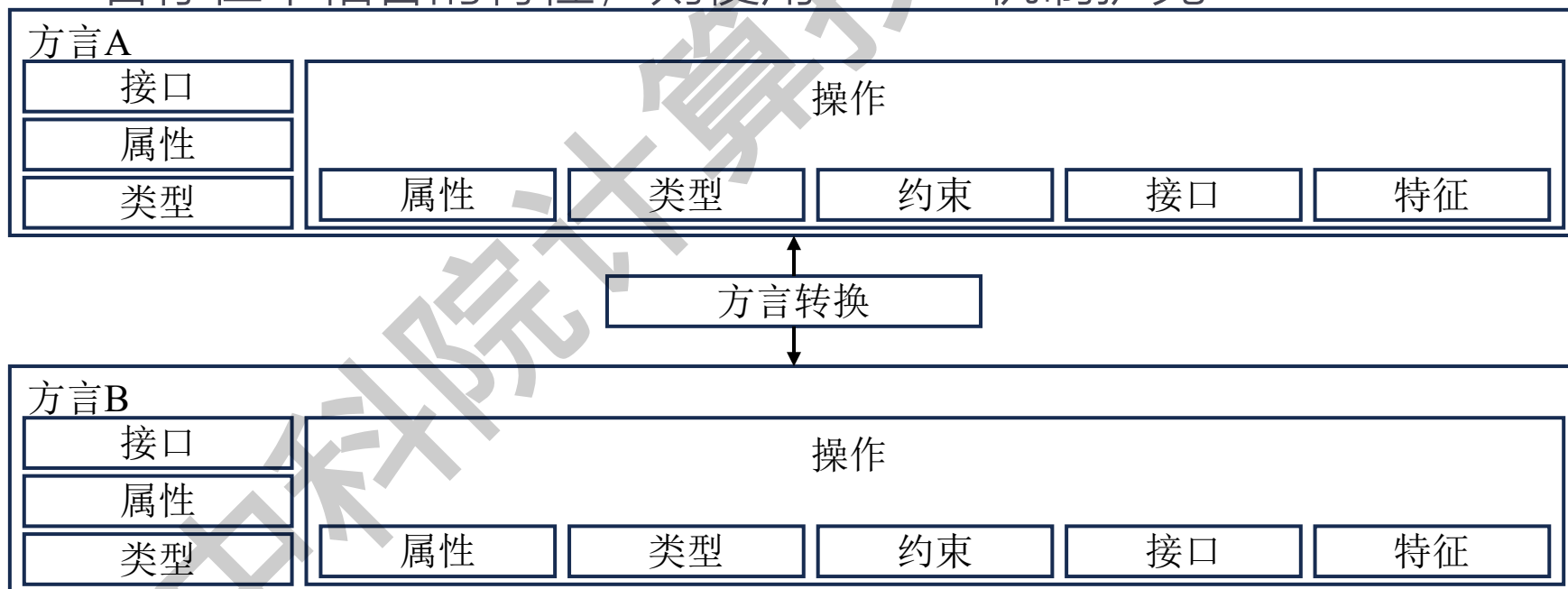
- ▶ 神经网络、学习框架、硬件平台数量蓬勃发展
- ▶ 为减少重复工作量，MLIR（多层次中间表示）的概念被提出



- ▶ 最差情况下，需要做 $N*M*P$ 的适配
- ▶ MLIR提供了一套**基础设施**为开发编译器提供便利

MLIR

- ▶ 提供一套基础设施为开发编译器提供便利
 - ▶ 使用混合的中间表示，解决当下的软件碎片化问题
- ▶ 设计方言“Dialect”机制，便于接入各式语言和中间表示
 - ▶ 若存在不相容的特性，则使用Dialect机制扩充



PyTorch 编译技术迭代

PyTorch中编译技术的发展历程

版本号	更新内容
/	使用Codegen而非C++模板，利用Structure Kernels对算子生成进行描述；使用Dispatcher分派
1.0	引入JIT编译器，使用jit.trace或jit.script
1.8	引入Torch.fx，使用中间表示FX IR实现Python到Python的代码转换
1.12	引入nvFuser，一个针对nVidia平台的编译器
2.0	引入torch.compile()，结合多种编译技术优化体验



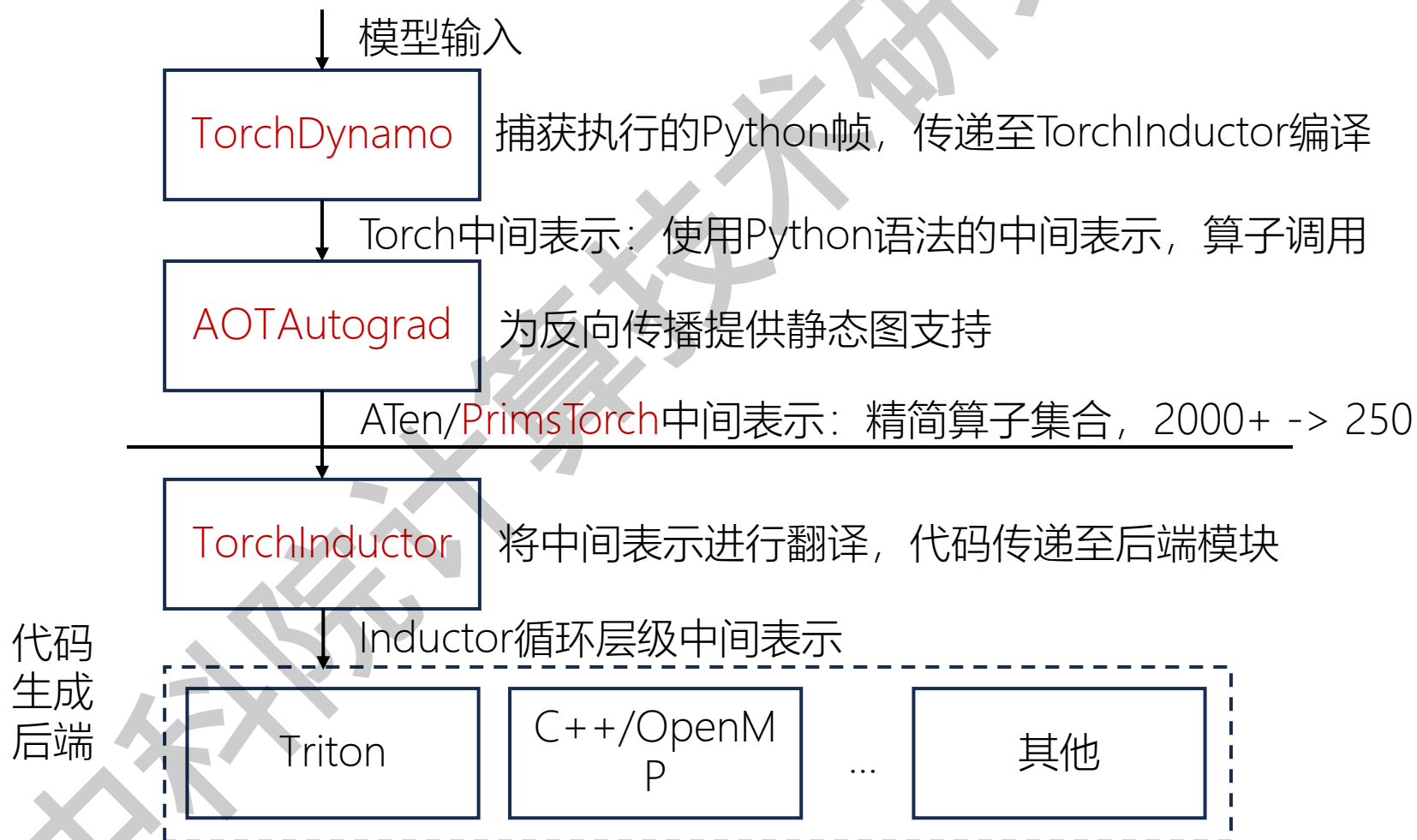
PyTorch 2.0

- ▶ 易用且高性能: `torch.compile()`
- ▶ PyTorch 2.0 引入的新编译相关技术

技术	用途
TorchDynamo	图捕获工具, 捕获执行的Python帧并将其给Inductor编译, 用户无感
TorchInductor	深度学习编译器后端, 将Dynamo捕获的结果编译到多个后端
AOTAutograd	重载了Pytorch的Autograd, 使用trace技术生成反向的trace
PrimTorch	将PyTorch的2000多个算子抽象缩减为了约250个算子的集合
Triton	TorchInductor中使用的GPU后端, 使用Python编写算子并自动调优

TorchDynamo & TorchInductor

核心技术的组织结构



常见深度学习编译器对比

编译器名称	主要维护团体	中间表示	典型特点
TVM	Apache	图层级: Relay/Relax中间表示 算子层级: Tensor中间表示	基于算子调度的自动调优
Tensor Comprehensions	官方不再维护	算子层级: 基于Halide的中间表示	基于多面体模型的自动优化
XLA	Google	图层级: HLO中间表示	细粒度的算子抽象、算子融合
MLIR	LLVM Developer Group	基于方言机制的多层级中间表示	编译器基础设施

提纲

- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练
- ▶ 本章小结

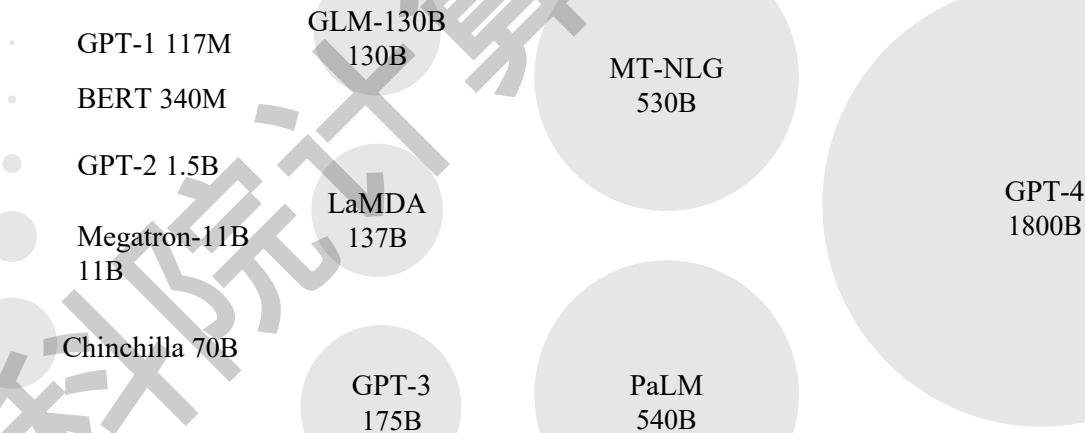
1、为什么需要分布式训练

- ▶ 大模型及其相关应用蓬勃发展
 - ▶ 参数数量的增加带来了模型的表达能力和拟合能力提高
 - ▶ 庞大的训练数据使得模型能够学习到更全面的知识和对数据分布的理解
 - 许多从前难以实现的任务变得可行
- ▶ 更大的参数量和更多的训练数据导致训练过程中的算力墙和存储墙

更大的参数量

- ▶ 大模型通常拥有更长更深的网络结构
 - ▶ 网络中有更多层次的神经元结构→更大的参数量
 - ▶ 从 GPT-1 到 GPT-3，模型的参数量从1.2亿增长到了1750亿

常见大模型参数量（截至2023年7月）



更多的训练数据

- ▶ 大模型的训练都需要海量的数据
 - ▶ 数据的多样性可以帮助模型更好地理解 and 捕捉数据中的模式
 - ▶ 从GPT-1到GPT-3，训练模型使用的数据集大小从5GB增长到了753GB； GPT-3训练使用的数据集包含维基百科、书本、会议期刊、源代码等；

数据集大小/GB	GPT-1	GPT-2	GPT-3	GPT-J/GPT-NeoX-20B	Megatron-11B	MT-NLG	Gopher
维基百科	\	\	11	6	11	6	12
书籍	5	\	21	118	5	118	2100
学术杂志	0	\	101	244	\	77	\
Reddit链接	\	40	50	63	38	63	\
Common crawl数据集	\	\	570	227	107	983	3450
其它	\	\	\	167	\	127	4823
总计	5	40	753	825	161	1374	10385

模型很大 & 数据很多

- ▶ 在训练过程中产生了计算墙和存储墙
 - ▶ 计算墙：GPT-3模型的训练若使用8张V100显卡，训练预计耗时36年，而使用1024张A100可以将训练耗时减少到1个月
 - ▶ 存储墙：千亿级别大模型的存储(包括参数、训练中间值等)需要2TB存储空间，而单卡的显存目前最大只有80GB
- ▶ **单个计算设备的资源有限**，无法存储整个模型的参数或者计算全部的数据集，并且**提升单个设备性能成本远远高于使用多个设备**
- ▶ 分布式训练技术：拆分任务并由多个设备共同协作完成计算
 - ▶ 拆分训练数据
 - ▶ 拆分模型（计算图）

2、分布式训练基础

- ▶ 分布式架构和分布式同步策略是分布式训练的基础
 - ▶ 分布式架构：组织和管理分布式训练任务的方式，以最大程度地利用计算资源和提高训练效率
 - ▶ 分布式同步策略：在分布式环境中，为了保证计算节点之间的一致性和正确性，对不同计算节点之间的操作进行协调和同步的策略

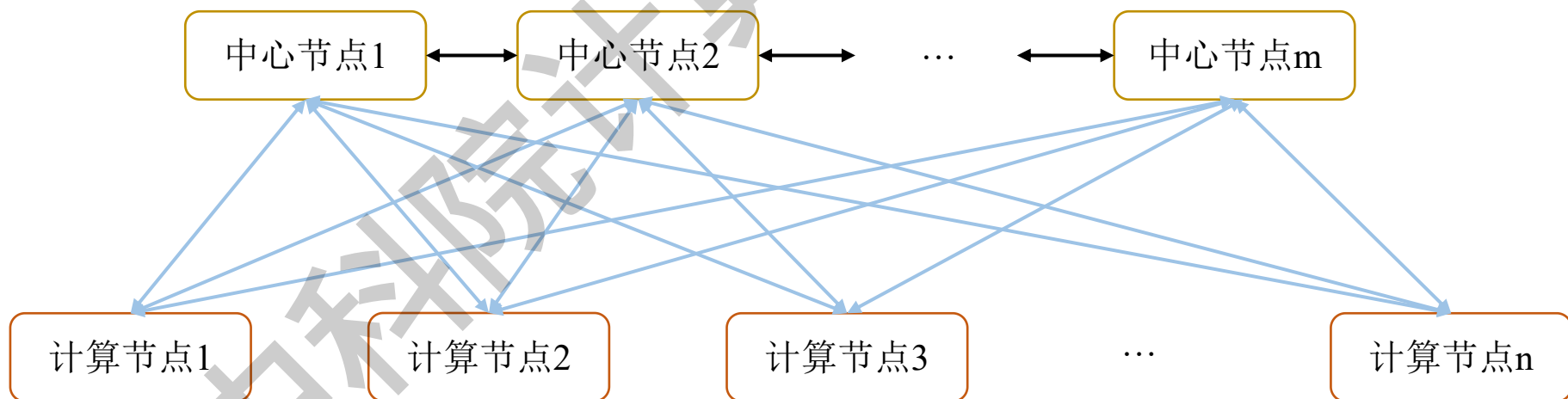
分布式架构

常使用以下两种架构实现分布式训练:

- ▶ 参数服务器(Parameter Servers)
 - ▶ 中心化的Parameter Servers架构由李沐于2014年提出,“中心化”是指将模型参数进行中心化管理,以此实现模型参数的同步
- ▶ 集合通信(Collective Communication)
 - ▶ 去中心化的Collective Communication架构中,每个训练节点都有当前全局最新参数,节点间的参数同步通常采用多次设备之间的点对点通信完成的

参数服务器

- ▶ 参数服务器将所有节点分成中心节点(Server节点) 和计算节点(Worker节点)两类
 - ▶ 中心节点用于存储参数和梯度更新
 - ▶ 计算节点用于完成中心节点下发的实际计算任务, 仅与中心节点通信以更新和检索共享参数

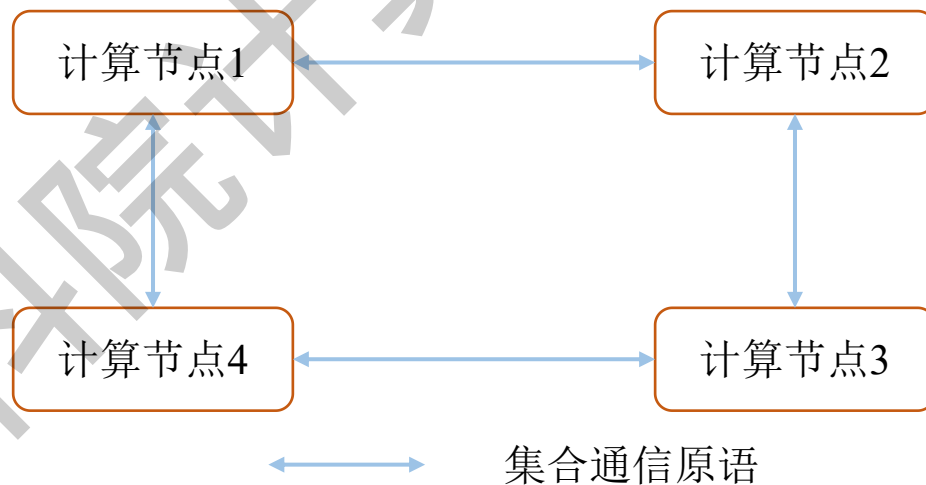


参数服务器

- ▶ 参数服务器架构的计算和存储分离
- ▶ 优点
 - ▶ 灵活：通过改变中心节点数量适应不同的负载和数据规模
 - ▶ 高效地参数共享：由中心节点统一管理模型参数
- ▶ 缺点
 - ▶ 单点故障：单个中心节点故障会影响整个系统
 - ▶ 数据一致性的问题：多个计算节点可能同时读取和更新模型参数
 - ▶ 网络通信开销：受通信带宽的限制，中心节点成为系统的瓶颈

集合通信

- ▶ 集合通信是指一个进程组的所有进程都参与全局通信操作
- ▶ 集合通信中没有中心节点（也被称为去中心化的架构）
 - ▶ 每个计算节点都有当前全局最新参数
 - ▶ 节点间的参数同步通常采用多次设备之间的点对点通信完成的
 - ▶ 对芯片的算力和芯片之间的网络通信要求较高



集合通信

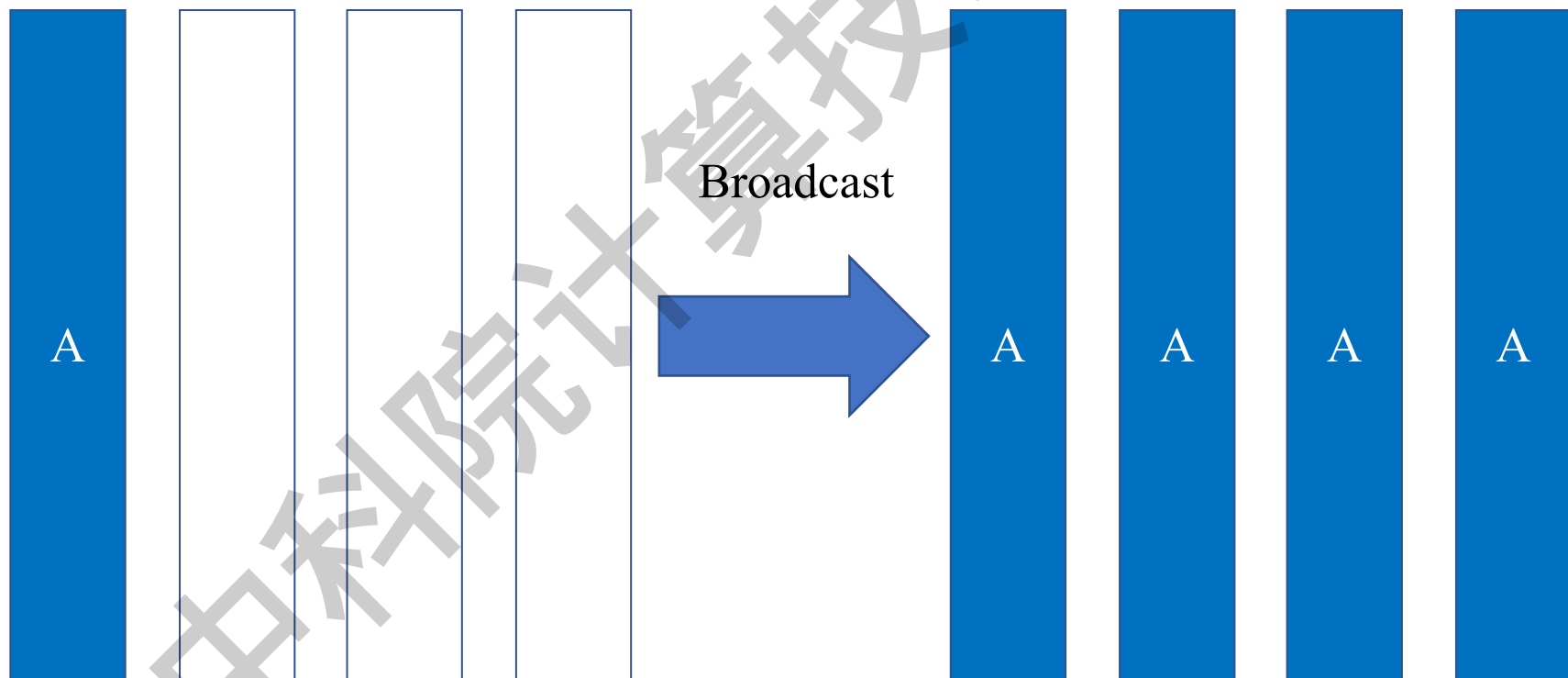
- ▶ 集合通信的基础操作：发送 (send)、接收 (receive)、复制 (copy)、组内进程障碍同步 (barrier)以及节点间进程同步 (signal+wait)
- ▶ 基础操作**组合**后可以得到集合通信中常用的通信原语
- ▶ 通信原语
 - ▶ 一对多通信原语：Broadcast、Scatter
 - ▶ 多对一通信原语：Gather、Reduce
 - ▶ 多对多通信原语：All-to-All、All-Gather、All-Reduce、Reduce-Scatter

集合通信原语

- 一对多广播 (Broadcast) : 将一个进程的数据广播到所有进程, 常用于分享模型参数

DLP 0 DLP 1 DLP 2 DLP 3

DLP 0 DLP 1 DLP 2 DLP 3

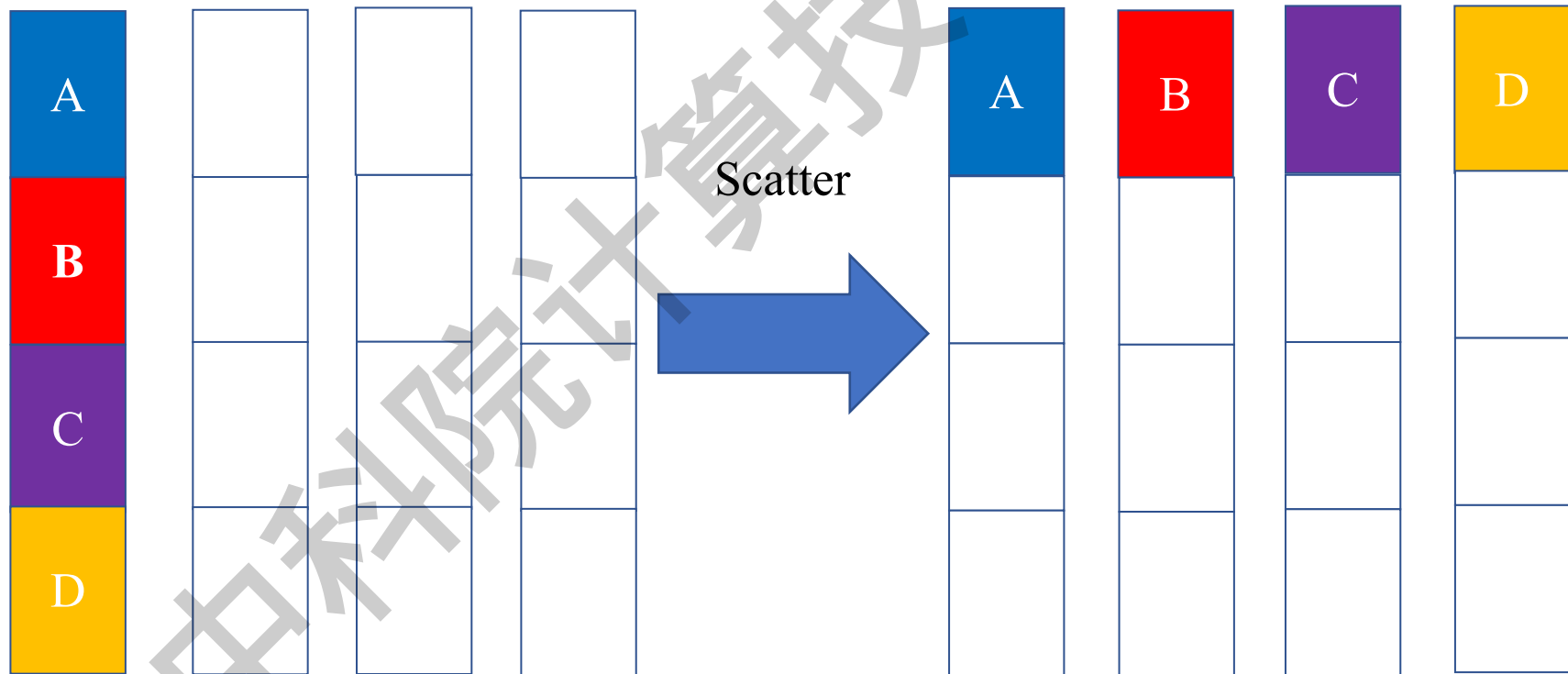


集合通信原语

- 一对多散射 (Scatter) : 将一个进程中的数据按索引散射到多个进程, 常用于更新权重

DLP 0 DLP 1 DLP 2 DLP 3

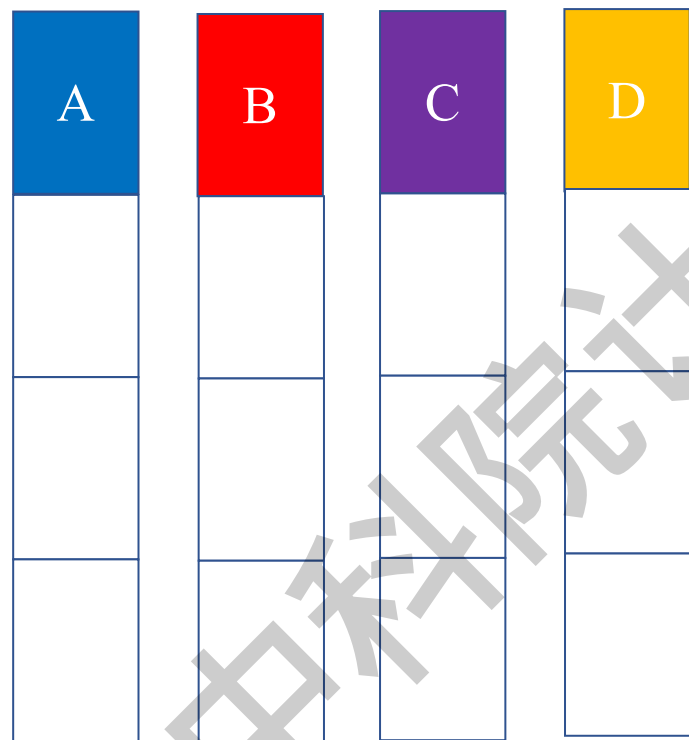
DLP 0 DLP 1 DLP 2 DLP 3



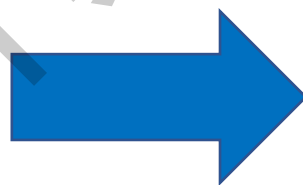
集合通信原语

- 多对一收集 (Gather) : 从多个进程收集数据到一个进程, 常用于收集梯度

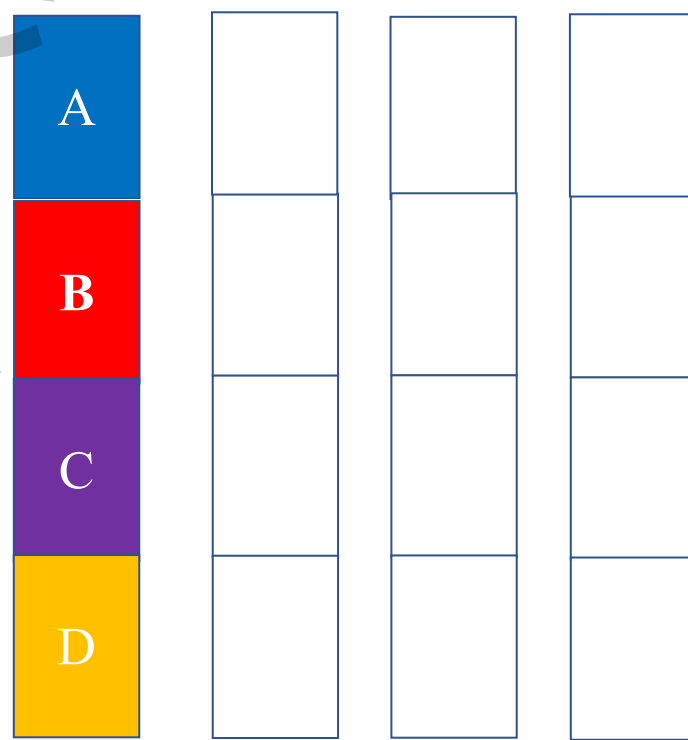
DLP 0 DLP 1 DLP 2 DLP 3



Gather



DLP 0 DLP 1 DLP 2 DLP 3

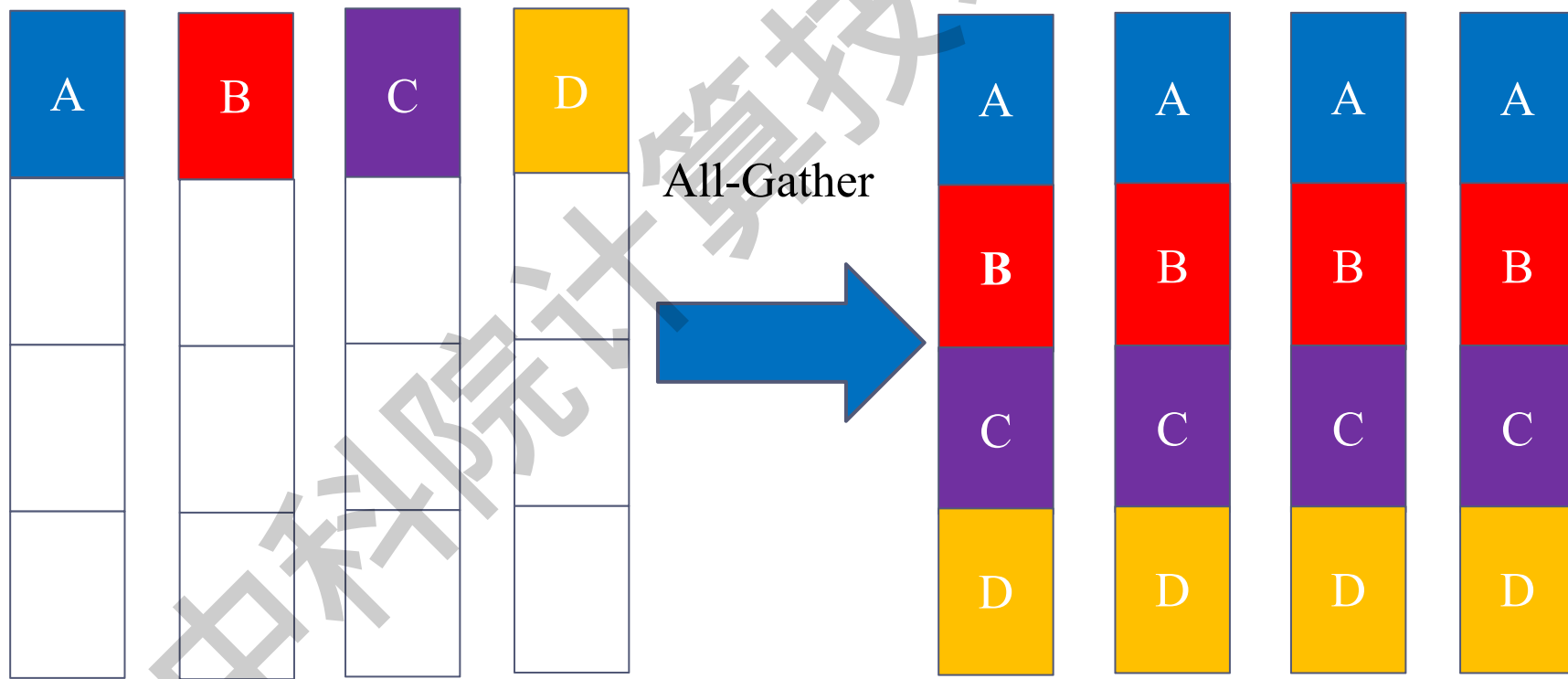


集合通信原语

- 多对多收集 (All-Gather) : 从多个进程收集数据, 并广播到所有进程, 常用于数据同步

DLP 0 DLP 1 DLP 2 DLP 3

DLP 0 DLP 1 DLP 2 DLP 3

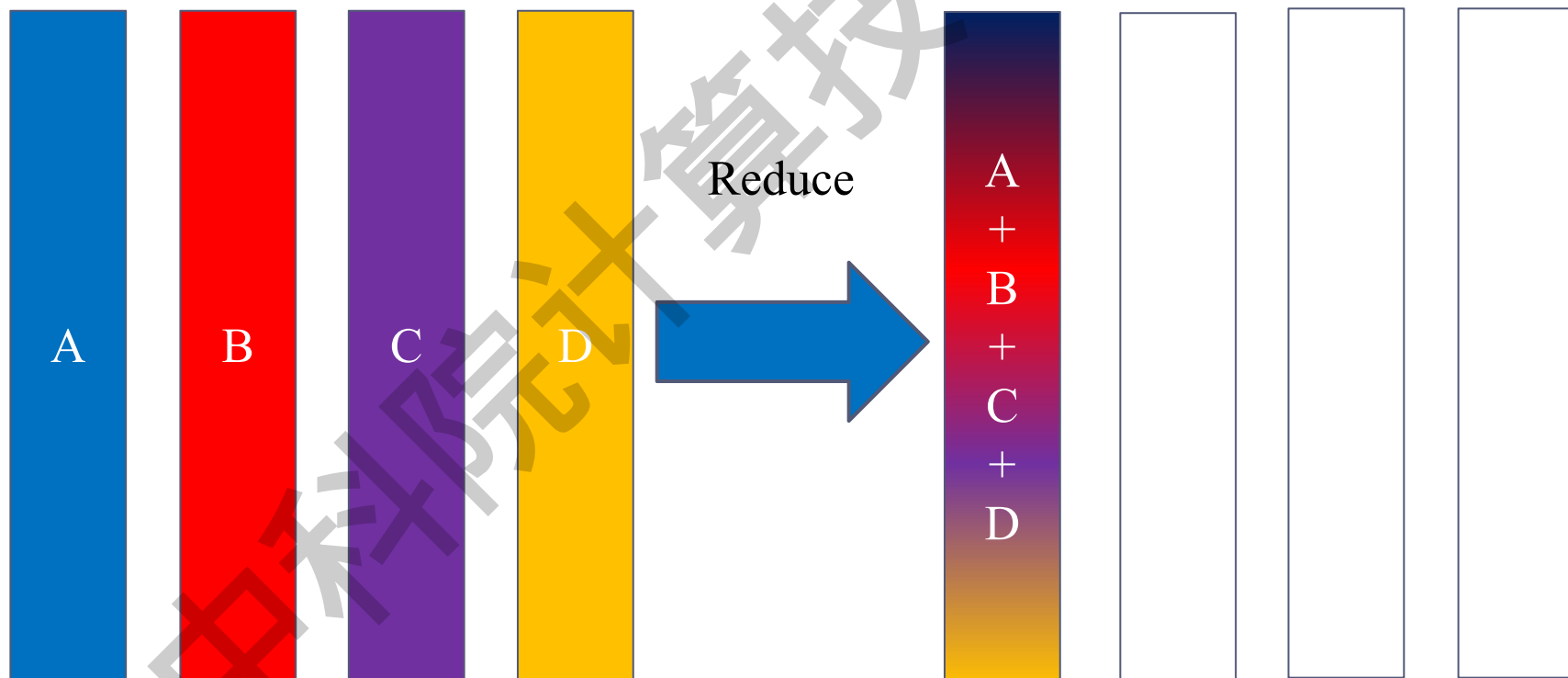


集合通信原语

- 多对一归约 (Reduce) : 从多个进程收集数据, 并按某种运算 (如求和运算) 归约到一个进程, 常用于梯度累加

DLP 0 DLP 1 DLP 2 DLP 3

DLP 0 DLP 1 DLP 2 DLP 3

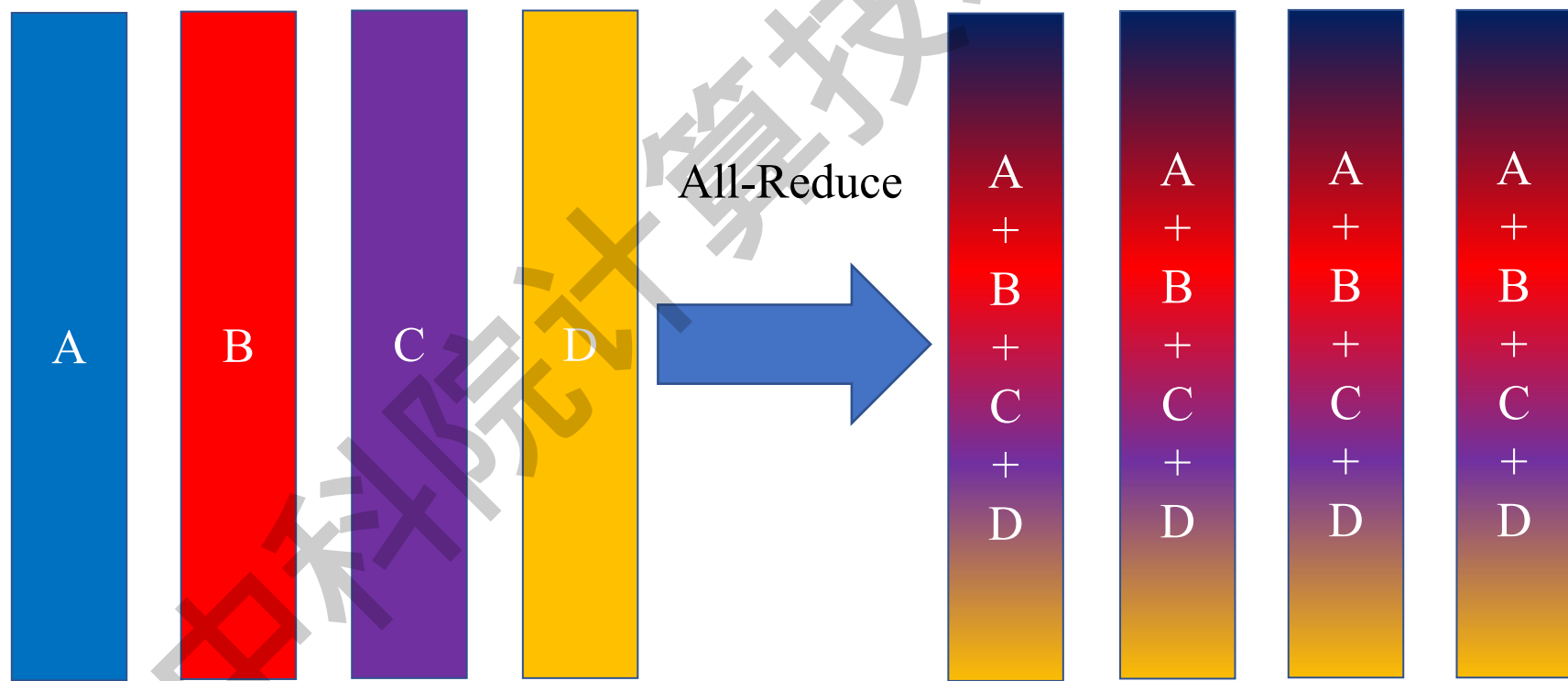


集合通信原语

- 多对多归约 (All-Reduce) : 从多个进程收集数据, 并按某运算归约, 再广播到所有进程, 常用于数据同步和梯度累加

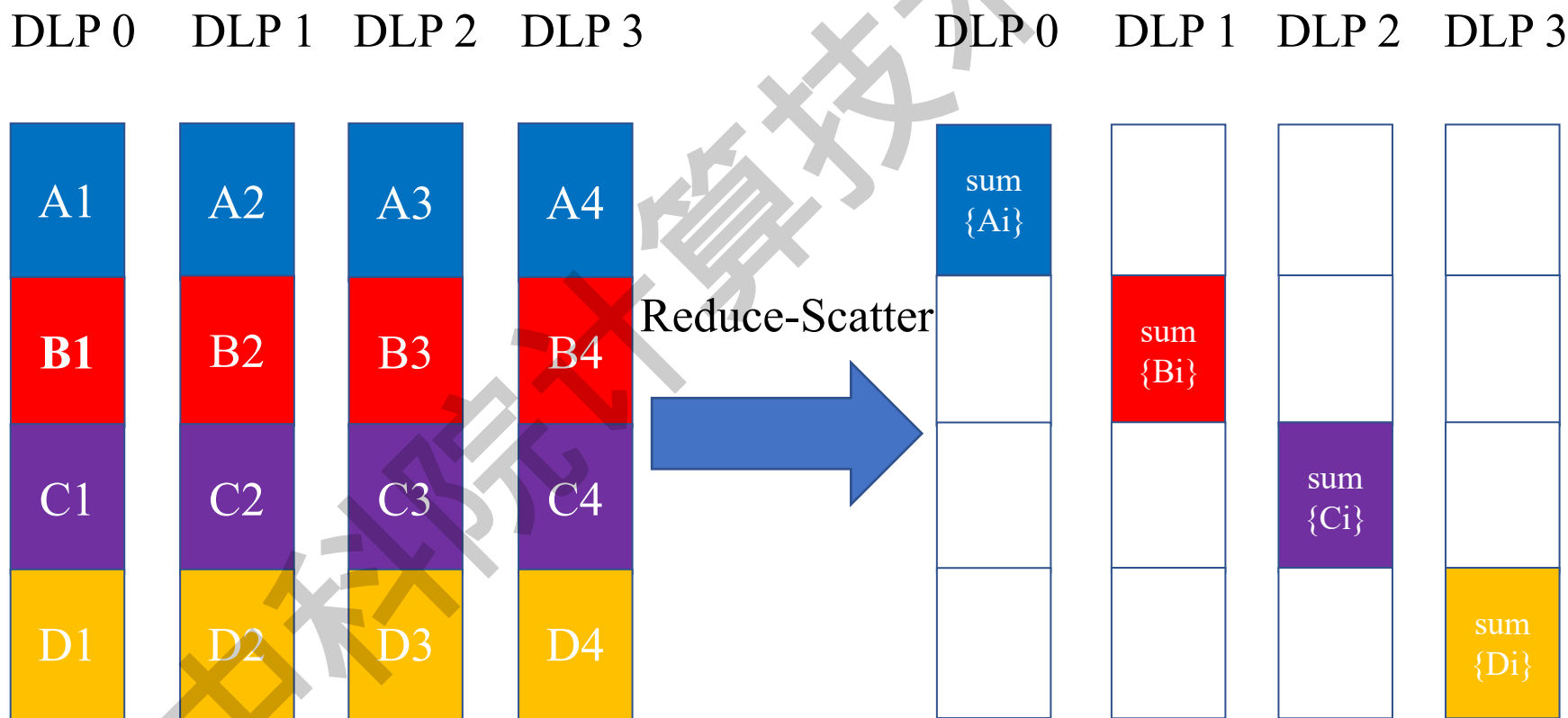
DLP 0 DLP 1 DLP 2 DLP 3

DLP 0 DLP 1 DLP 2 DLP 3



集合通信原语

- 多对多归约散射 (Reduce-Scatter) : 从多个进程收集数据, 并按某种运算归约到一个进程, 将该进程中的数据按索引散射到对应进程上, 常用于更新权重



分布式同步策略

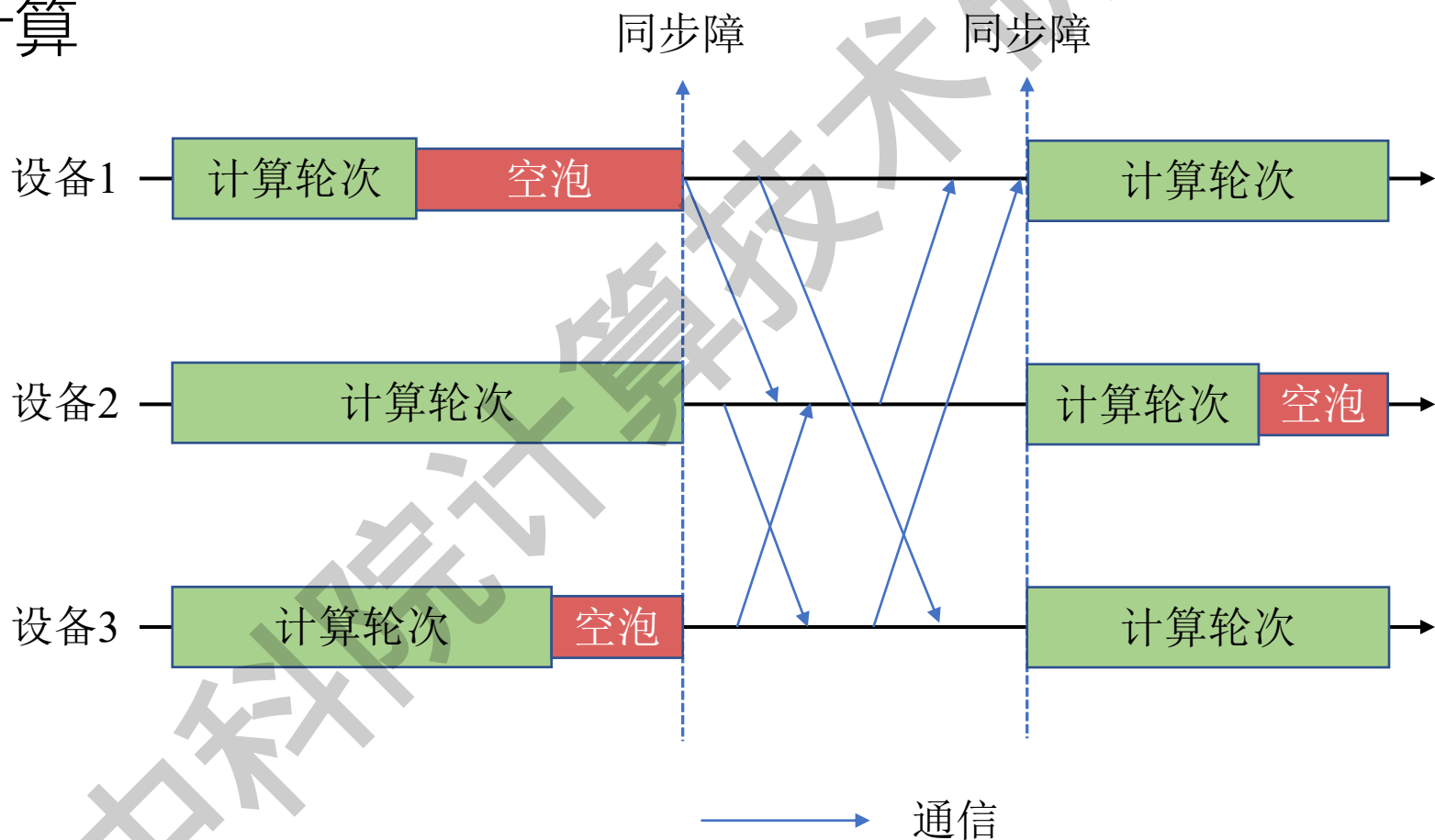
- ▶ 设备之间的通信可以采用不同的同步策略
 - ▶ 同步通信
 - ▶ 异步通信
- ▶ 选择合适的分布式同步策略对于保证分布式系统的正确性、性能和可扩展性至关重要

同步通信

- ▶ 采用同步通信作为分布式同步策略
 - ▶ 需要等待全部计算节点完成本轮计算后才进行通信
- ▶ 时序性和顺序性
 - ▶ 使用同步障确保计算节点之间的数据一致性
 - ▶ 可能会导致较大的延迟和通信开销

同步通信

- 同步障的存在确保全部设备完成通信后才可开始下一轮计算

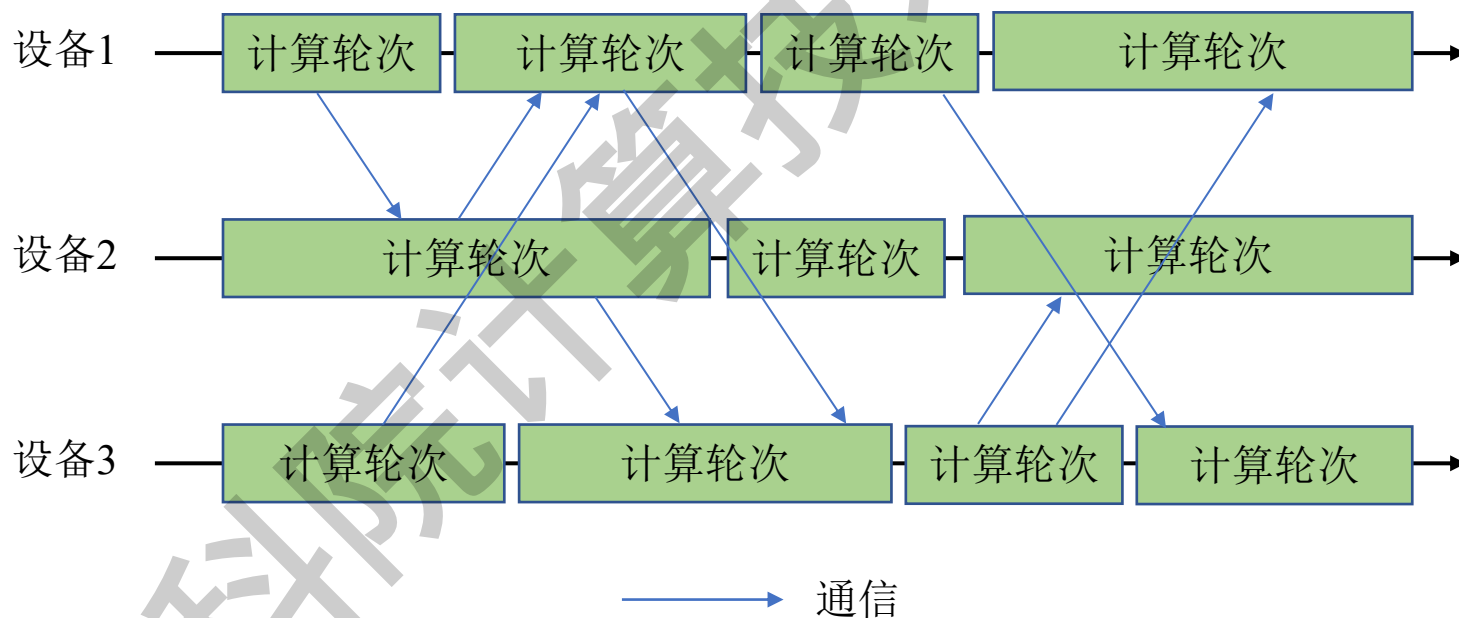


异步通信

- ▶ 采用异步通信作为分布式同步策略
 - ▶ 每个计算节点可以随时和其他设备进行通信
- ▶ 更加灵活
 - ▶ 提高整个分布式训练系统的计算利用率
 - ▶ 但不能保证数据的一致性

异步通信

- ▶ 每个设备可以随时处理自己收到的信息，不会因为同步障碍而带来互相等待的开销



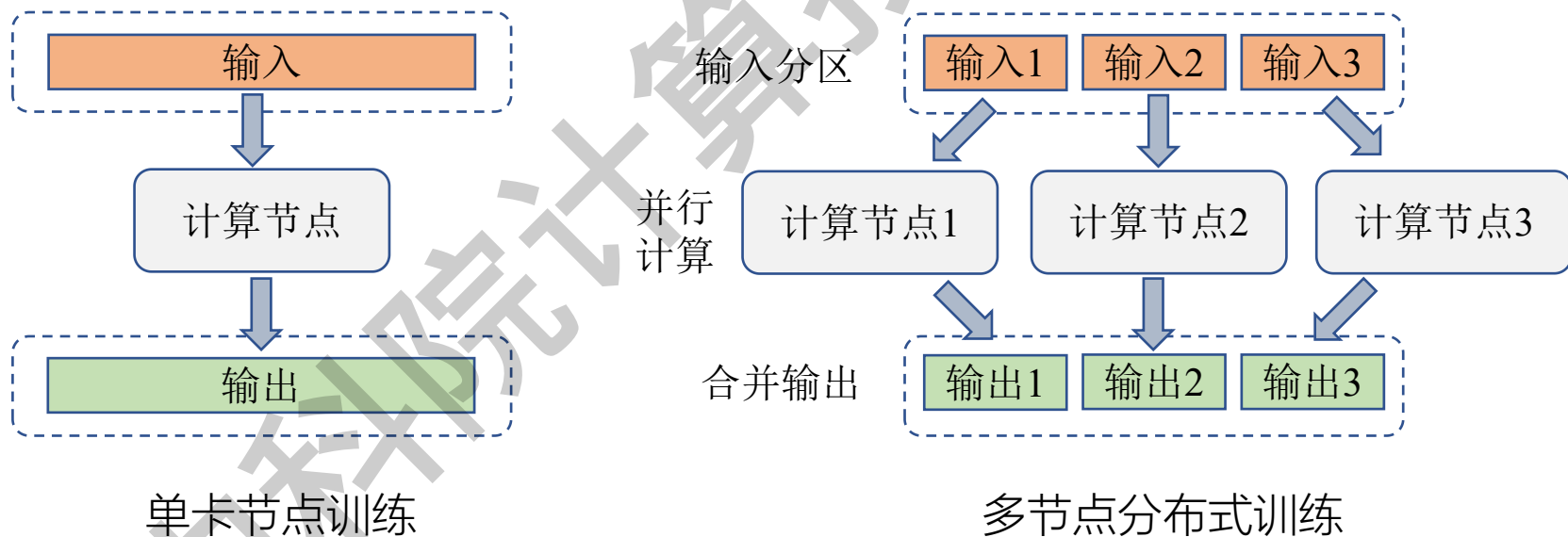
3、分布式训练方法概述

根据分布式计算中的分区情况，可以划分出不同的分布式计算方法：

- ▶ 数据并行：对输入数据进行分区
- ▶ 模型并行：对模型参数进行分区
- ▶ 混合并行：同时对输入数据和模型参数进行分区

分布式计算步骤

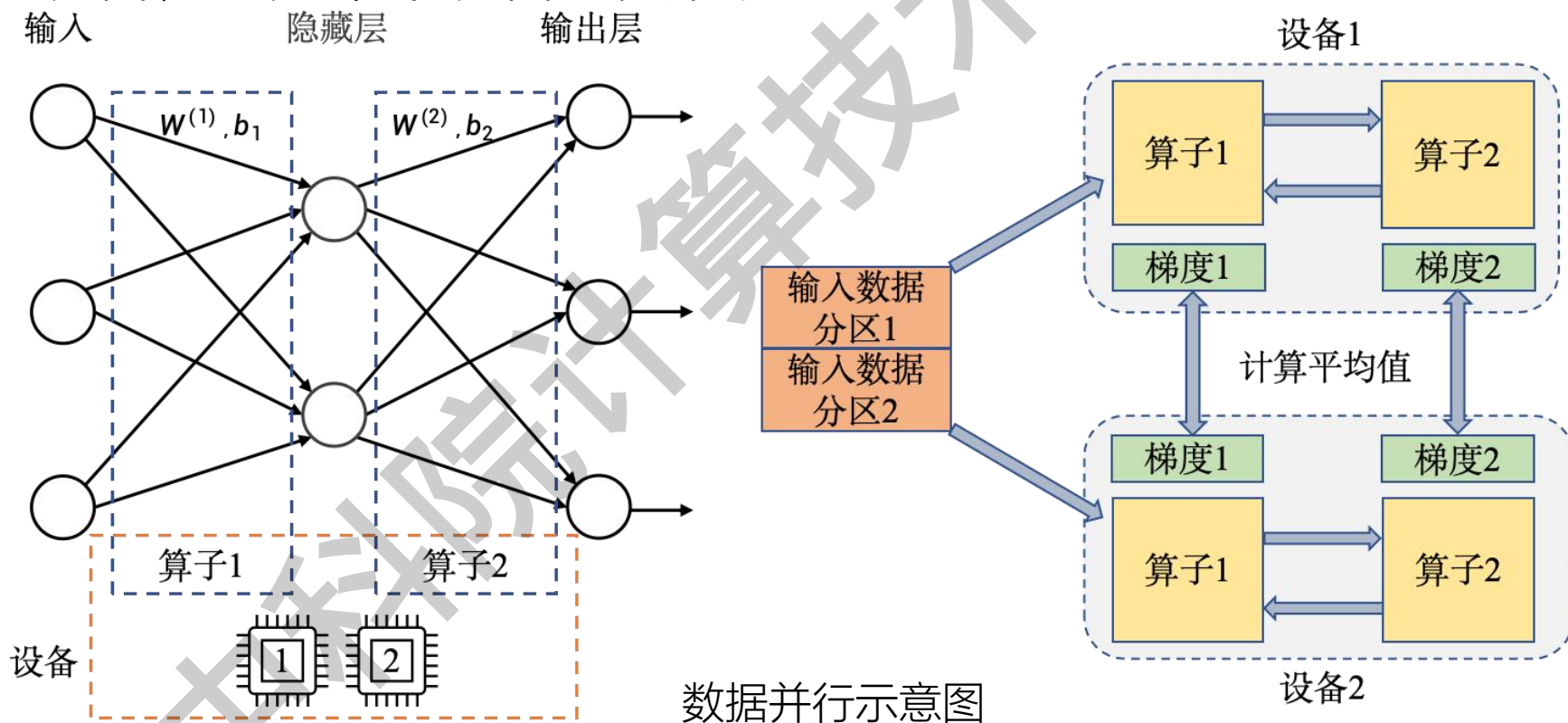
- ▶ 分布式计算一般包含三个步骤
 - ▶ (1) 将输入进行 **分区**
 - ▶ (2) 将每个分区发给不同的计算节点，实现 **并行** 计算
 - ▶ (3) **合并** 每个计算节点的输出，得到和单节点等价的计算结果



数据并行

数据并行(Data Parallelism)往往用于解决单节点算力不足的问题。

其中，每个设备共享完整的模型副本，输入数据会被分发给这些设备，减少单个设备的负载。



模型并行

模型并行(Model Parallelism)往往用于解决单节点内存不足的问题。一般将模型并行分为算子内并行和算子间并行。

▶ 算子内并行

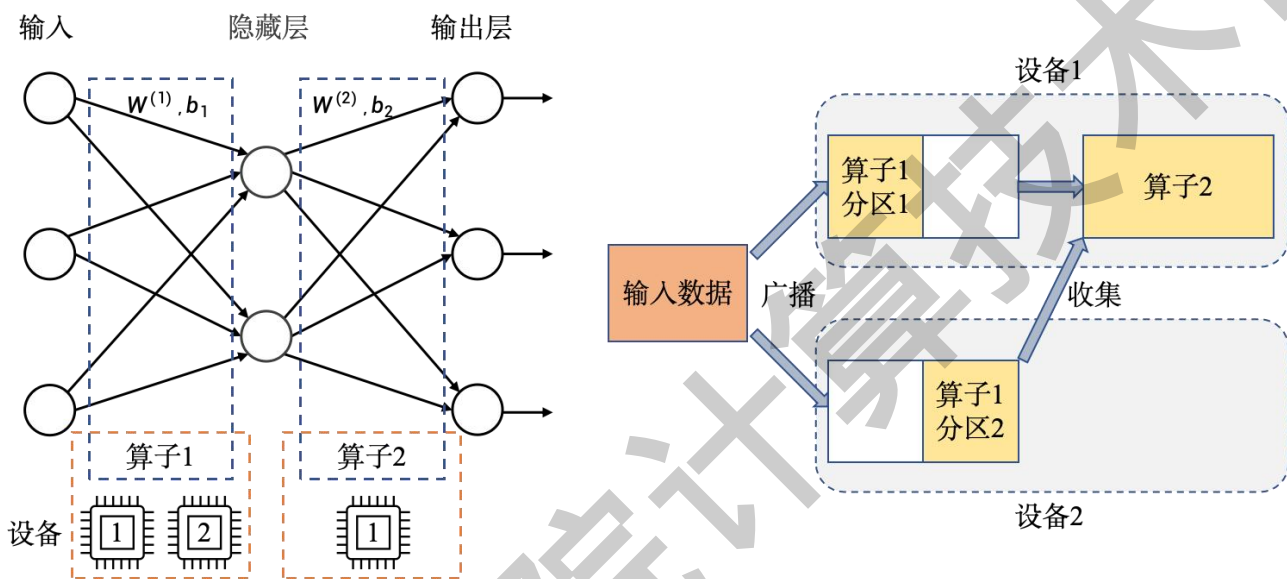
- ▶ 大型算子计算所需内存超过单设备内存容量，对单个算子进行切分
- ▶ 按行切分和按列切分

▶ 算子间并行

- ▶ 模型的总内存需求超过单设备的内存容量，在算子间进行切分

算子内并行

模型中单个算子本身计算所需的内存已经超过单设备的内存容量，就需要对这些大型算子进行切分。



正向：输入被广播给设备1和设备2，计算结果合并后传给下游算子2



反向：算子2的数据被广播给设备1和设备2，两设备根据本地的参数分区完成局部的反向计算

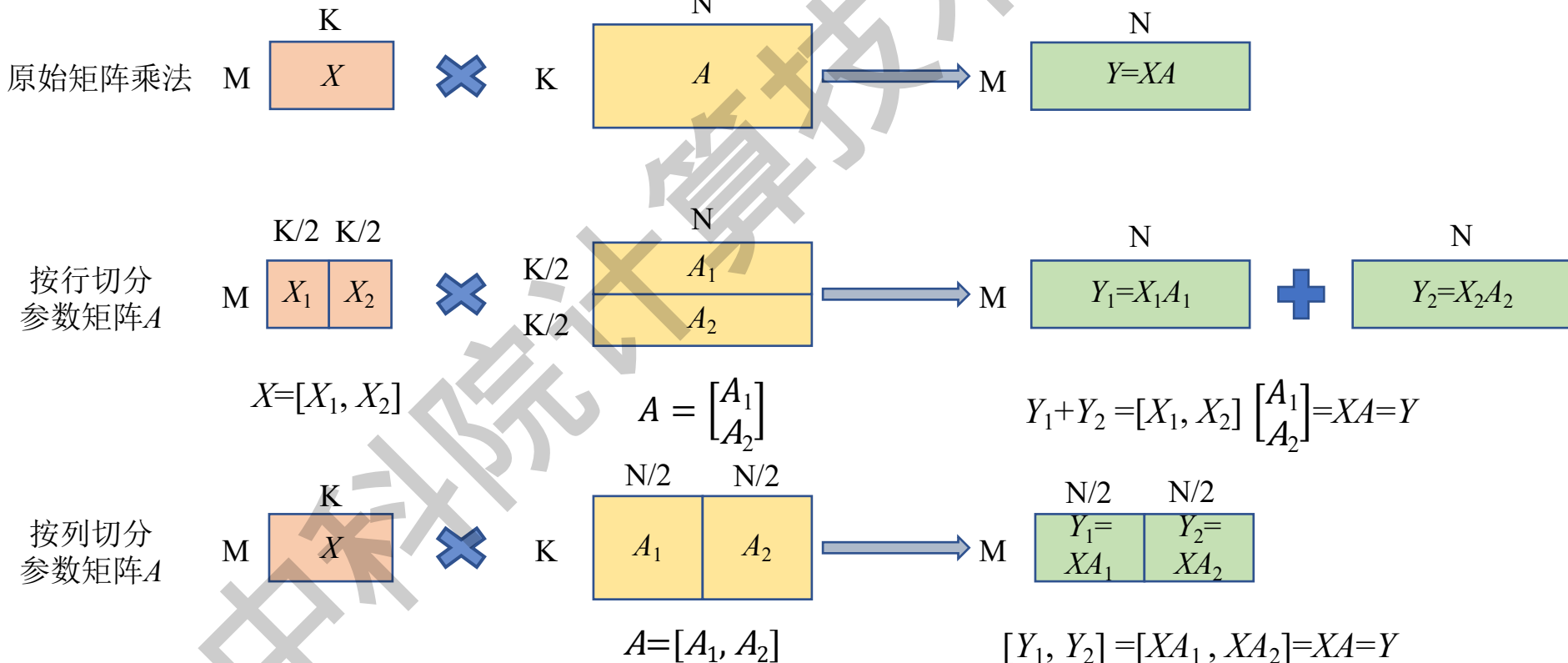
模型并行示意图——算子内并行

对单个算子的切分一般可以分为按行切分和按列切分

按行切分和按列切分

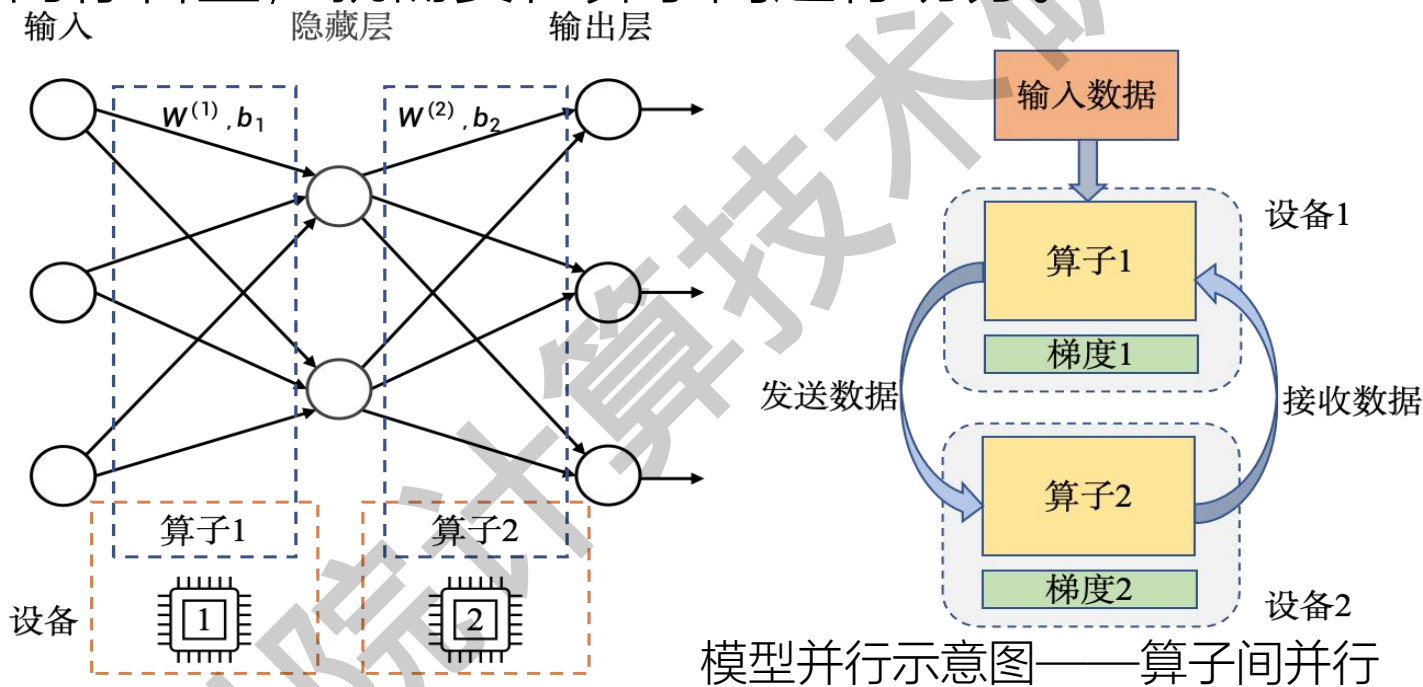
算子内并行可以采用不同的模型参数分区方式:

- 按行切分: 将参数矩阵 A 切分为 $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$
- 按列切分: 将参数矩阵 A 切分为 $A = [A_1, A_2]$



算子间并行

模型中单个算子参数量较少，但整个模型的总内存需求超过单设备的内存容量，就需要在算子间进行切分。



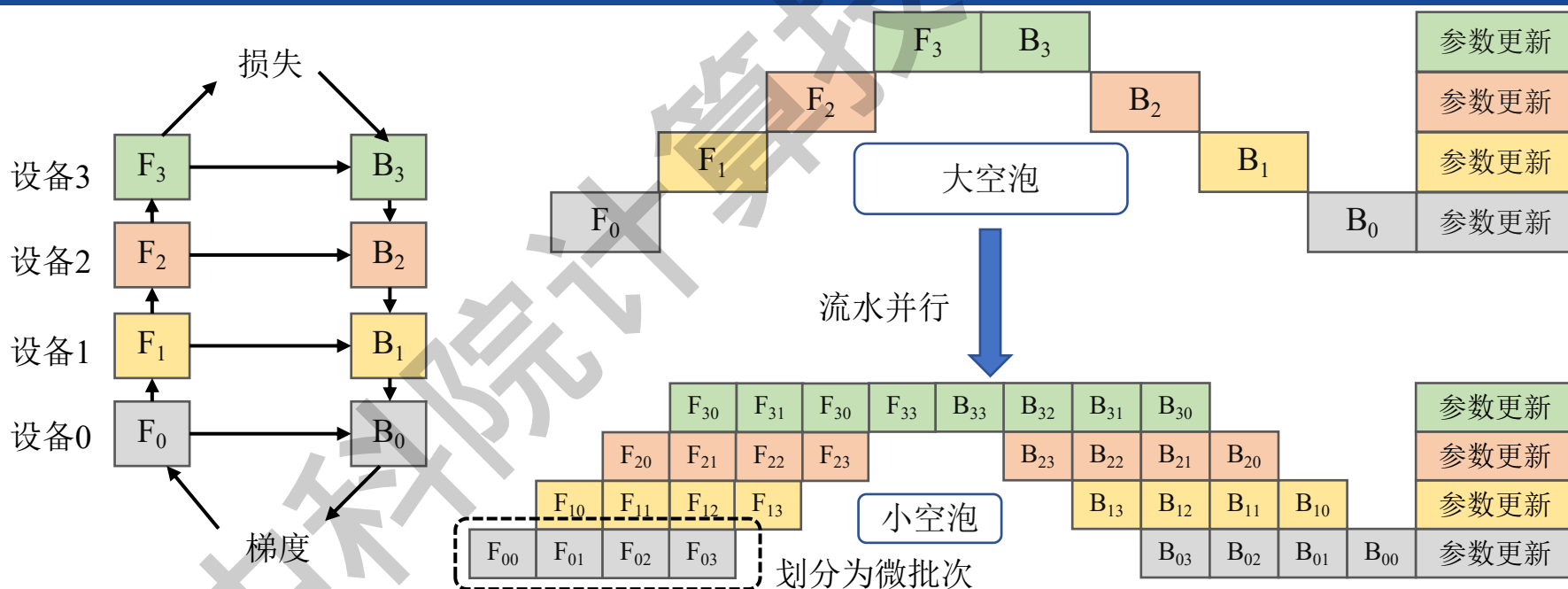
模型并行空泡(Model Parallelism Bubble)现象：算子间并行中，下游设备需要等待上游设备计算完成，因此下游设备容易长期处于空闲状态。

→ 利用流水线技术缓解空泡现象（流水并行）

流水并行

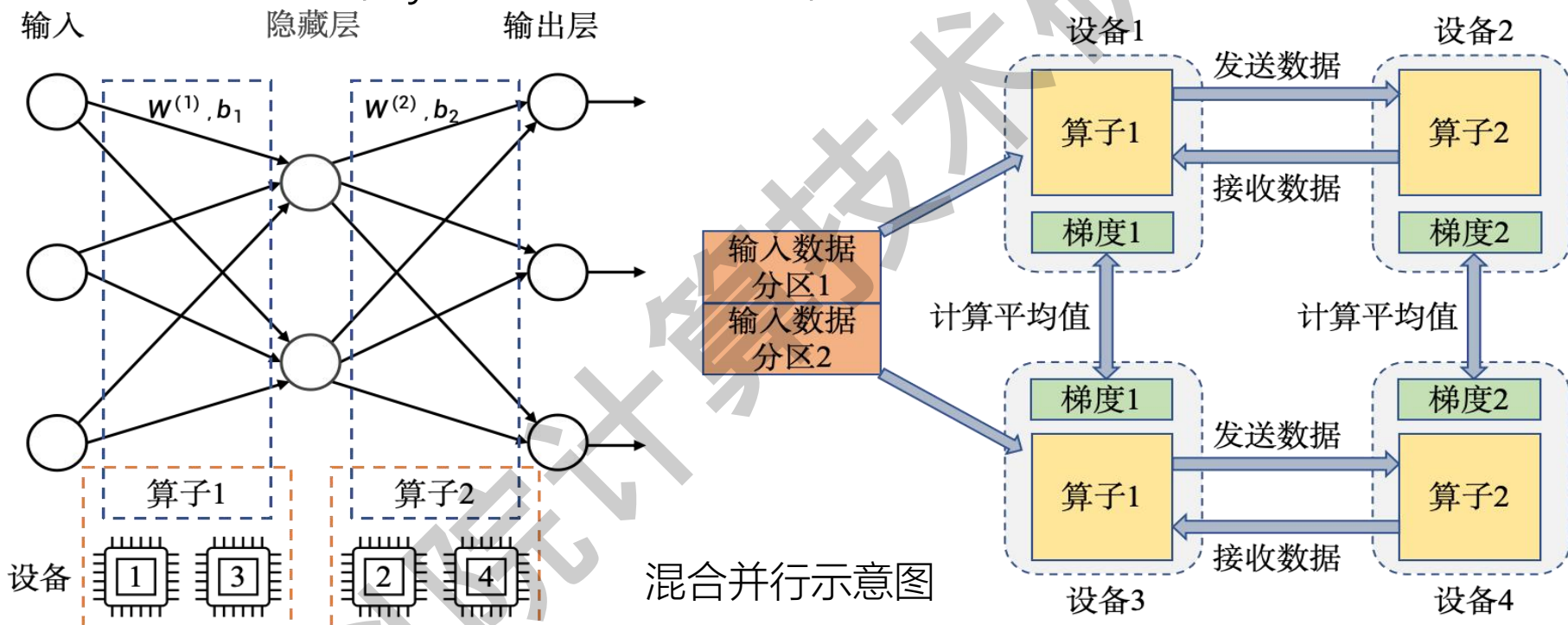
流水并行(Pipeline Parallelism)是在模型并行中构建流水线，并利用流水线调度。该训练系统中，模型的上下游算子会被分配到不同的流水阶段，每个设备负责一个流水阶段。

Batch划分成Micro-Batch，输入Micro-Batch完成前向和反向传播，利用平均梯度更新参数。



混合并行

数据并行和模型并行常在大模型的训练过程中同时使用，这也就是混合并行(Hybrid Parallelism)。



模型并行：
算子1和算子2被分配给设备1和设备2
→ 解决内存不足



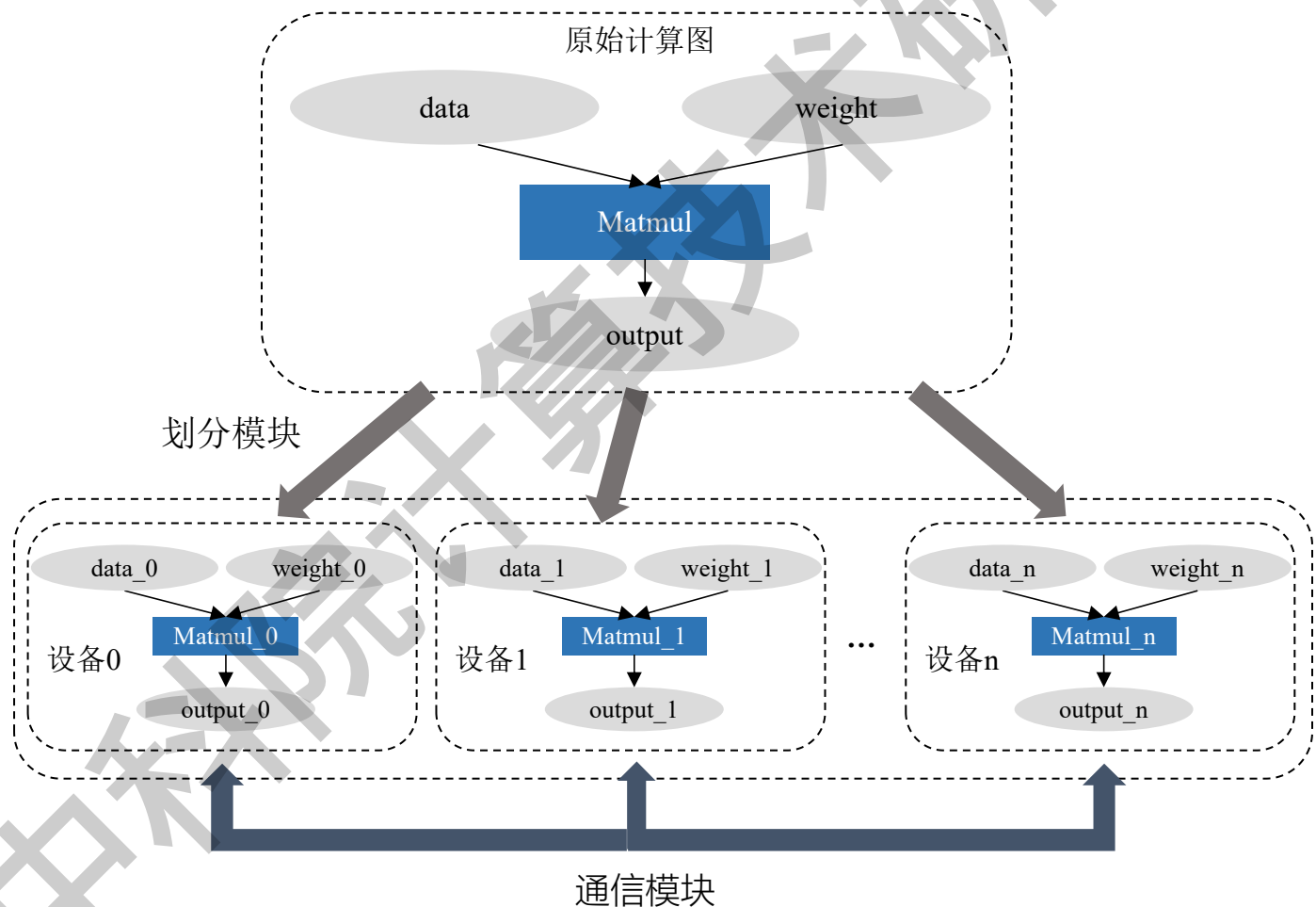
数据并行：
引入设备3和设备4，将输入数据分区
→ 提升系统总算力

4、分布式训练框架实现

- ▶ 实现一个分布式训练框架
 - ▶ 利用分布式架构和分布式同步策略
 - ▶ 支持常见的分布式训练方法
 - ▶ 达到高效利用设备资源的目的
- ▶ 分布式训练框架中最主要的两个模块
 - ▶ 划分模块：划分训练任务
 - ▶ 通信模块：管理节点之间的通信

分布式训练框架中的划分和通信模块

- ▶ 划分模块：原始计算图可以被拆分为多个子计算图，
- ▶ 通信模块：在设备之间进行通信，实现模型参数初始化和同步



划分模块

- ▶ 任务划分方法
 - ▶ 数据并行划分：对输入数据进行划分
 - ▶ 模型并行划分：对模型参数进行划分
 - ▶ 混合划分：对输入数据和模型参数都进行划分

数据并行划分

- ▶ 模型会被复制为很多份并分发给各个设备
- ▶ 使用代码自动加载不重叠的训练数据
 - ▶ PyTorch的DDP库依靠DistributedSampler类对样本（即训练数据）进行采样
 - ▶ DistributedSampler类
 - ▶ `__init__()`: 计算单次采样数和总样本数
 - ▶ `__iter__()`: 返回一个采样迭代器，使用给定的种子和当前epoch生成样本的随机索引顺序
 - ▶ 最终，每个进程在每个epoch中都能获得相同数量且不重复的样本

模型并行划分

- ▶ 每个设备会获得一部分的模型
- ▶ 编程者手动划分模型
 - ▶ ToyModel的两个线性层被放置在两个不同的设备上计算
 - ▶ 算子间并行的思想 → 可以采用流水线并行进一步优化

```
class ToyModel(nn.Module):  
    def __init__(self):  
        super(ToyModel, self).__init__()  
        self.net1 = torch.nn.Linear(10, 10).to(torch.device('dlp:0')) # net1放在设备0  
        self.relu = torch.nn.ReLU()  
        self.net2 = torch.nn.Linear(10, 5).to(torch.device('dlp:1')) # net2放在设备1  
  
    def forward(self, x):  
        x = self.relu(self.net1(x.to(torch.device('dlp:0'))))  
        return self.net2(x.to(torch.device('dlp:1')))
```

通信模块

- ▶ 支持基础通信操作和常见的通信原语
 - ▶ 模型参数发送：初始化时需要将模型参数发送到各个设备数据
 - ▶ 参数梯度平均：计算时需要各个设备进行参数梯度平均
 - ▶ ...
- ▶ 用成熟的通信库作为通信模块的基础
 - ▶ CNCL (Cambricon Neuware Communication Library)
 - ▶ NCCL (NVIDIA Collective Communications Library)

模型参数发送

- ▶ 模型参数等初始化信息从主节点发送到其他节点
 - ▶ 确保所有进程上的模型状态保持一致
- ▶ PyTorch的DDP库使用`_sync_module_states()`函数
 - ▶ 收集模块中的参数和缓冲区信息，装入列表`module_states`
 - ▶ 使用broadcast通信原语，将参数和缓冲区信息广播到其他进程

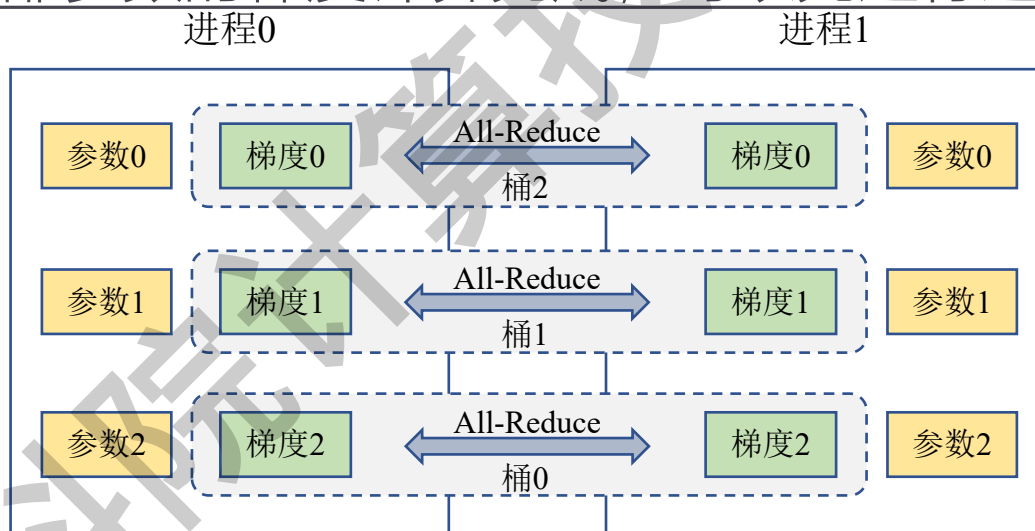
模型参数发送

- ▶ 将收集到的初始化信息广播到其他进程（使用broadcast）

```
def _sync_module_states(  
    module: nn.Module, process_group: dist.ProcessGroup, broadcast_bucket_size: int,  
    src: int, params_and_buffers_to_ignore: Container[str], broadcast_buffers: bool = True,) -> None:  
    module_states: List[torch.Tensor] = []  
    for name, param in module.named_parameters():  
        if name not in params_and_buffers_to_ignore:  
            module_states.append(param.detach())  
  
    if broadcast_buffers:  
        for name, buffer in module.named_buffers():  
            if name not in params_and_buffers_to_ignore:  
                module_states.append(buffer.detach())  
  
    _sync_params_and_buffers(process_group, module_states, broadcast_bucket_size, src)  
  
def _sync_params_and_buffers(  
    process_group: dist.ProcessGroup, module_states: List[torch.Tensor],  
    broadcast_bucket_size: int, src: int,) -> None:  
    # 将 module_states 在所有进程之间进行同步，通过从rank 0广播  
    if len(module_states) > 0:  
        dist.broadcast_coalesced(process_group, module_states, broadcast_bucket_size, src)
```

参数梯度平均

- ▶ DDP库的桶 (bucket) 机制
 - ▶ 将参数进行分组管理，每一组称为一个桶
 - ▶ 一般将相同类型的参数放在同一个桶之中
 - ▶ 当桶内全部参数的梯度计算完成，可以先进行通信操作



- ▶ 使用All-Reduce通信原语进行求和平均

本章小结

- ▶ 编程框架设计
- ▶ 计算图构建
- ▶ 计算图执行
- ▶ *深度学习编译
- ▶ *分布式训练



谢谢大家!

中科院计算技术研究所