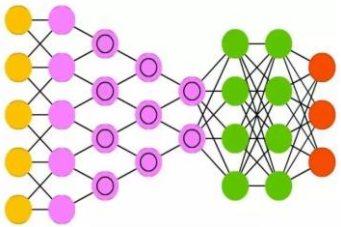
智能计算系统

第八章 智能编程语言

中国科学院计算技术研究所
陈云霁 研究员
cyj@ict.ac.cn

本章内容定位


输入 → 建模 → 实现 → 运行 → 输出

编程框架

学习了实现深度学习算法所使用的编程框架的基本原理、简单用法等

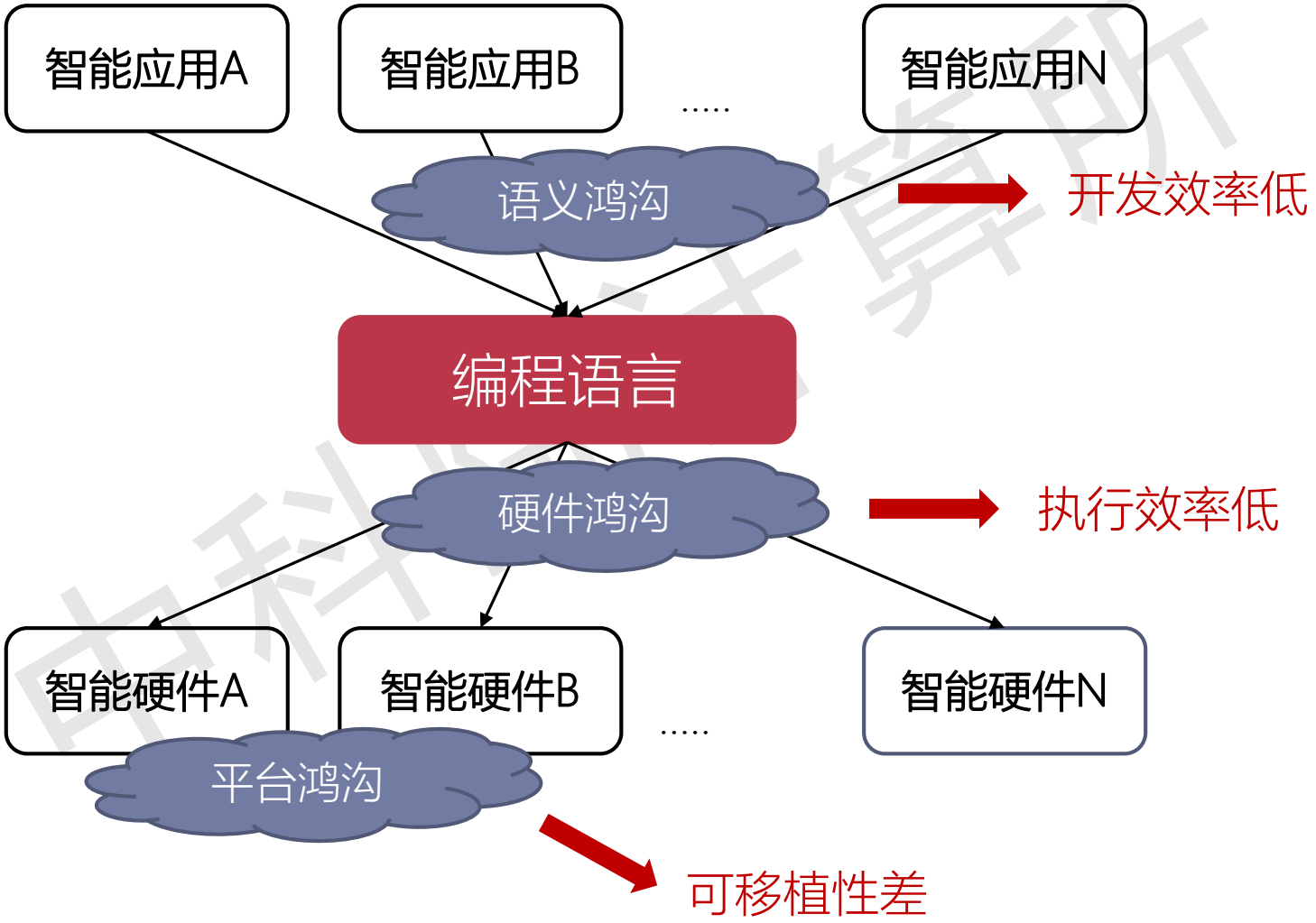
智能编程语言

本章将学习到智能编程语言的基础知识及相应的应用开发、调试和调优方法等

提纲

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

为什么需要智能编程语言



语义鸿沟

- 以深度学习中最为核心的卷积运算为例

C++：标量计算

```cpp
// 声明C++ array类型
T input = new T[ni * ci * (hi + 2 * pad) * (wi + 2 * pad)];
T filter = new T[co * ci * hk * wk];
T bias = new T[co];
int ho = (hi + 2 * pad - hk) / stride + 1;
int wo = (wi + 2 * pad - wk) / stride + 1;
T output = new T[ni * co * ho * wo];
// 计算
for (int ni_idx = 0; ni_idx < ni; ni_idx++) {
  for (int co_idx = 0; co_idx < co; co_idx++) {
    for (int ho_idx = 0; ho_idx < ho; ho_idx++) {
      for (int wo_idx = 0; wo_idx < wo; wo_idx++) {
        T sum = T(0);
        for (int ci_idx = 0; ci_idx < ci; ci_idx++) {
          for (int hk_idx = 0; hk_idx < hk; hk_idx++) {
            for (int wk_idx = 0; wk_idx < wk; wk_idx++) {
              int hi_idx = ho_idx * stride + hk_idx;
              int wi_idx = wo_idx * stride + wk_idx;
              sum += input[((ni_idx * ci + ci_idx) * (hi + 2 * pad) + hi_idx) * (wi + 2
    * pad) + wi_idx] * filter[((co_idx * ci + ci_idx) * hk + hk_idx) * wk + wk_idx];
            } } }
        output[((ni_idx * co + co_idx) * ho + ho_idx) * wo + wo_idx] = sum + bias[co_idx
    ];
    } } } }
```

7重循环

Python：向量语义（Array）

```python
# 声明numpy array类型
input = numpy.array(padded_input_data_list).reshape(ni, ci, hi+2*pad, wi+2*pad)
filter = numpy.array(filter_data_list).reshape(co, ci, hk, wk)
bias = numpy.array(bias_data_list).reshape(1, co, 1, 1);
ho = (hi + 2 * pad - hk) / stride + 1
wo = (wi + 2 * pad - wk) / stride + 1
output = numpy.array([0.]*(ni*co*ho*wo)).reshape(ni, co, ho, wo)
# 计算
for ni_idx in range(ni):
  for co_idx in range(co):
    for ho_idx in range(ho):
      for wo_idx in range(wo):
        hi_idx = ho_idx * stride
        wi_idx = wo_idx * stride
        output[ni_idx, co_idx, ho_idx, wo_idx] = np.sum(input[ni_idx, :, hi_idx:hi_idx+hk
    , wi_idx:wi_idx+wk] * filter[co_idx, :, :, :]) + bias[0, co_idx, 0, 0]
```

4重循环

- 有Conv语义和Tensor类型的编程语言

Tensor类型

```
// 声明tensors类型
Tensor input(ni, ci, hi, wi);
Tensor filter(co, ci, hk, wk);
Tensor bias(1, co, 1, 1);
Tensor output(ni, co, (hi+2*pad-hk)/stride+1, (wi+2*pad-wk)/stride+1)
// 计算
conv(input, filter, bias, output, pad, stride)
```

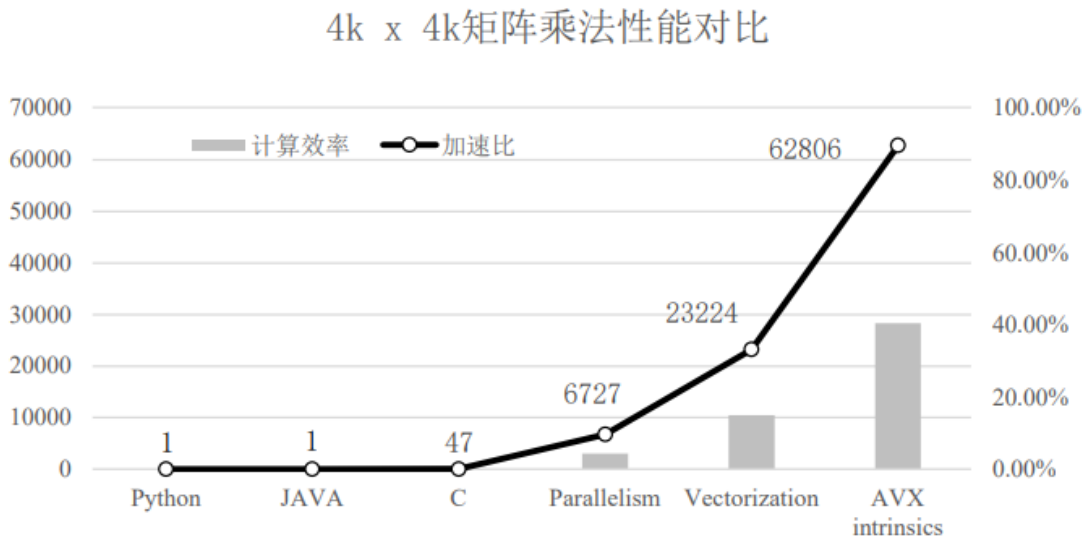
一条语句完成计算

硬件鸿沟

- 智能计算硬件在**控制**、**存储**和**计算**等方面有独特性
- 传统编程语言难以有效描述上述硬件特点
- 不同层次编程语言和硬件特性带来的性能影响

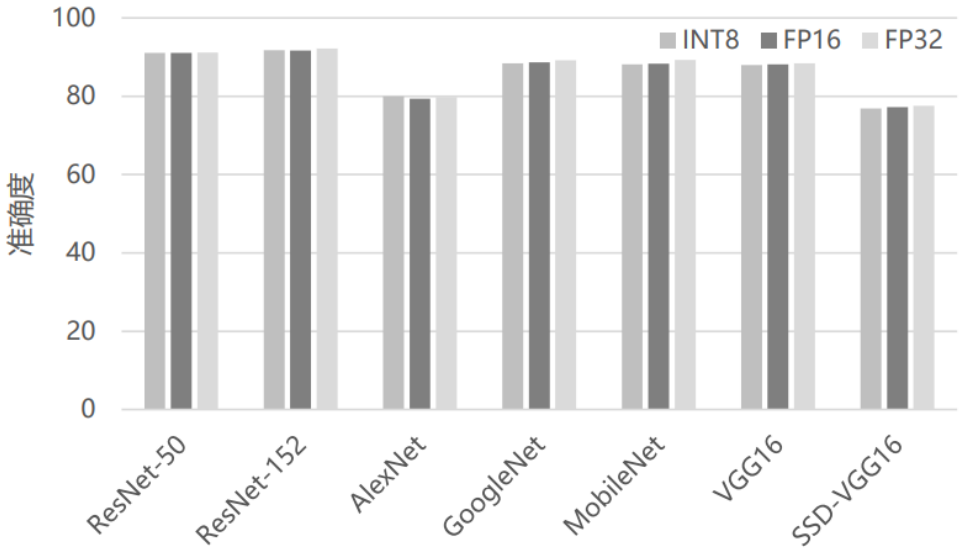
1、C语言实现相较Python/JAVA实现有47倍的性能提升

2、考虑了底层硬件提供的并行度、存储层次以及向量指令后，相较Python/JAVA有62,806倍的性能提升



- **存储逻辑**上一般采用程序员可见的Scratchpad Memory（SPM），而不是通用平台上程序员透明的Cache
- **计算逻辑**上提供了面向智能计算的定制运算单元，如16位浮点、Brain浮点等，其精度损失几乎可以忽略

传统编程语言中主要提供的是INT和FP32等数据类型，导致难以利用智能计算系统中更加丰富和高效的运算单元，如FP16和BF16等，甚至是INT4及Binary的数据类型。



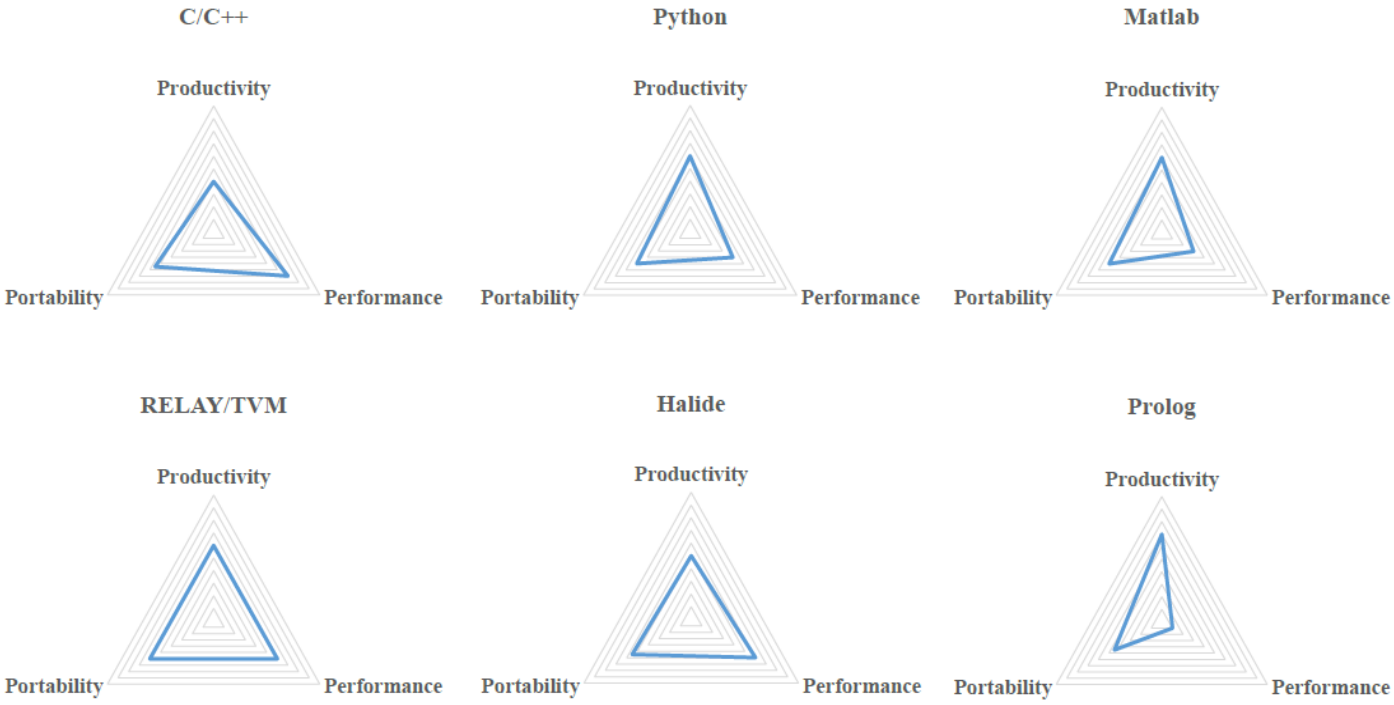
平台鸿沟

- **功能可移植性**：采用特定平台专用语言所编写的程序能够在别的平台上正常运行
  - 矩阵乘法的例子调用了AVX的intrinsic函数在ARM上无法运行
- **性能可移植性**：在特定平台上优化好的程序，在新的硬件平台上仍然保证有较高的执行效率

- 理想的编程语言需要**抽取不同硬件平台的共性特征**，并在此基础上提取**性能关键特征作为语言特性**提供给用户

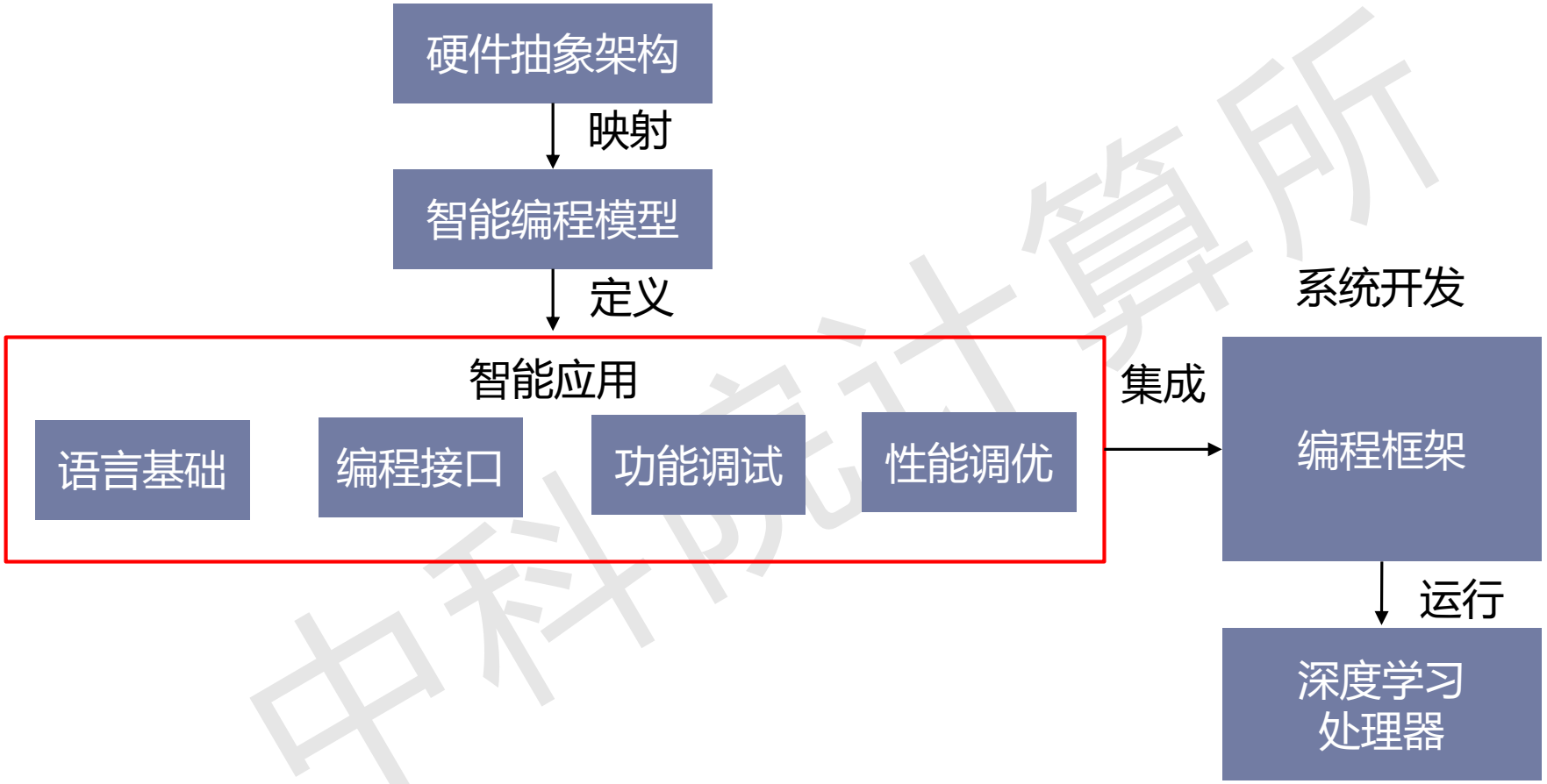
小结

- 面向**语义**、**硬件**和**平台**三大鸿沟，传统编程语言难以满足需求



- 领域专用编程语言是满足智能计算**高开发效率**、**高性能**和**高可移植性**的重要途径

内容组织



介绍智能编程语言原理基础上进一步介绍BCL语言的实例：
BANG语言

提纲

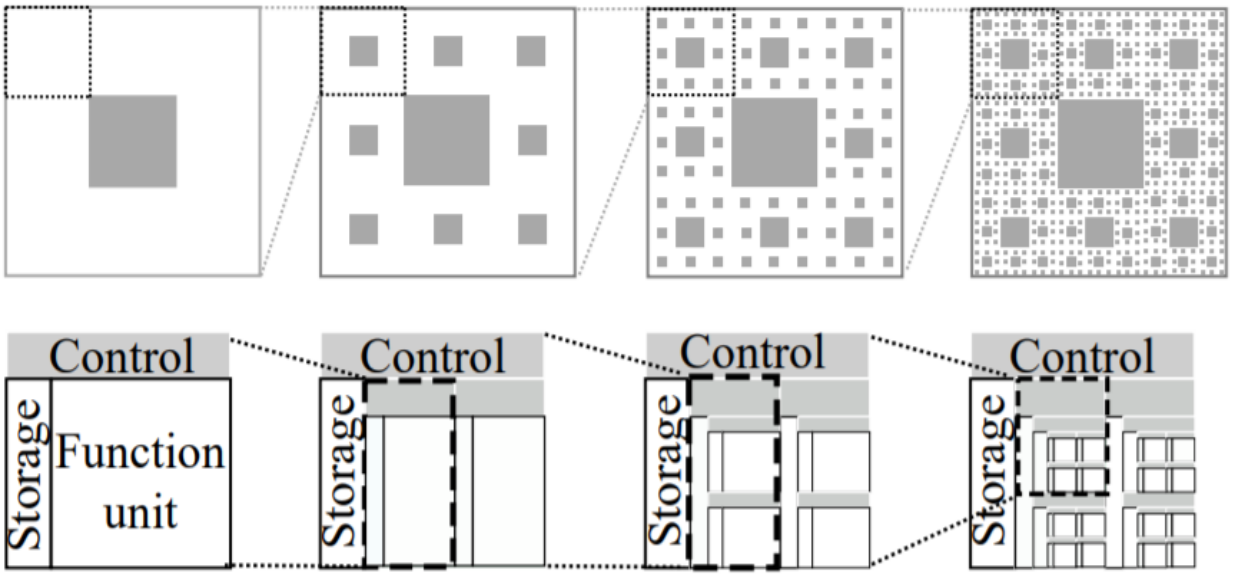
- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

智能计算系统抽象架构

- 抽象硬件架构
- 典型智能计算系统
- 控制模型
- 计算模型
  - 定制运算单元
  - 并行计算架构
- 存储模型

抽象硬件架构

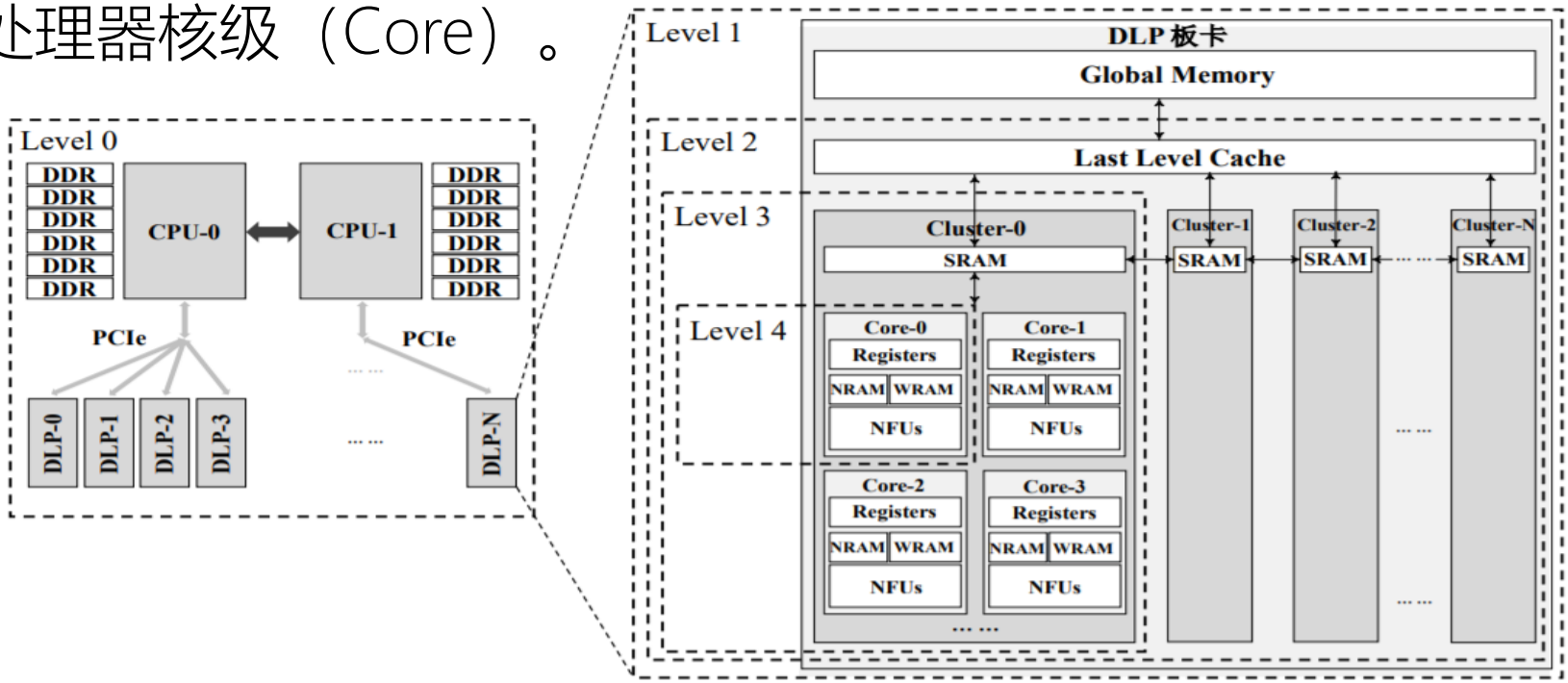
- 层次化的智能计算系统抽象硬件架构



- 智能计算系统中每一层都包含存储单元、控制单元和若干个计算单元
- 每个计算单元又进一步分解为子控制单元、子计算单元和子存储单元三部分，整个系统以这样的方式递归构成
- 在最底层，每个叶节点都是具体的加速器，用于完成最基本的计算任务。

典型智能计算系统

- 多卡的DLP服务器抽象为五个层次，即服务器级（Server）、板卡级（Card）、芯片级（Chip）、处理器簇级（Cluster）和处理器核级（Core）。



可以方便地通过增加各层次的规模来提升整个系统算力

控制模型

- 指令是实现对计算和存储进行控制的关键。为了设计高效的指令集、需要充分分析智能领域的典型计算模式，提炼最具代表性的操作，并进行针对性设计。
  - **对智能算法进行抽象**
    - 控制
    - 数据传输
    - 计算：标量、向量和矩阵运算等
    - 逻辑操作：标量和向量运算等
  - **关注计算与存储的交互**：尽可能将计算与存储并行，例如可以将控制计算和访存的指令分开在不同的队列中发射执行，以提高并行度

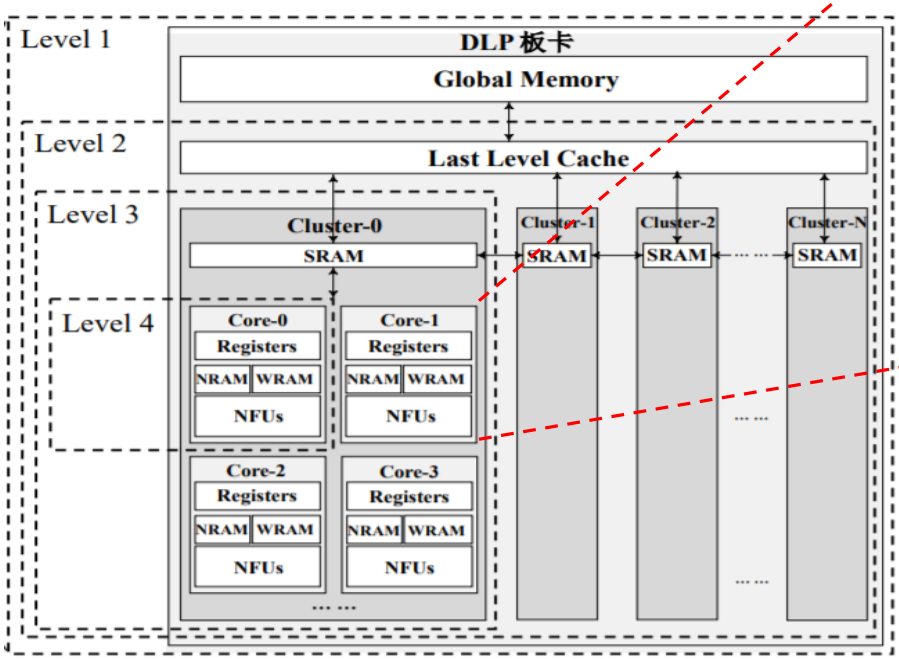
计算模型

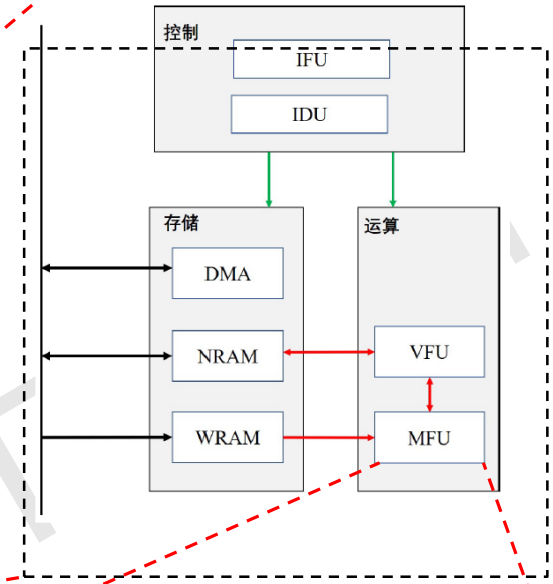
- 程序员可见的主要包括定制运算单元和并行计算架构
- 定制运算单元
  - 智能应用具有一定误差容忍度。通过利用智能应用误差容忍的特性，一般在智能计算系统中会采用定制的低位宽运算单元（如FP16、INT8、BF16甚至是INT4等）以提升处理能效。
  - 由于智能应用的多样性和复杂性，目前对于哪种低位宽最为合适并未形成统一结论。例如推断和训练对于精度的要求可能不一样，图像/视频类应用和语音类应用对于精度要求可能也不一样。

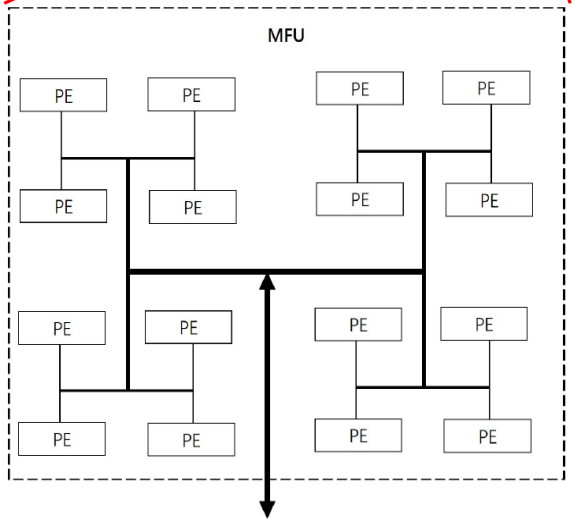
智能编程语言中需要有和各种类型运算单元对应的数据类型

并行计算架构

任务切分与同步

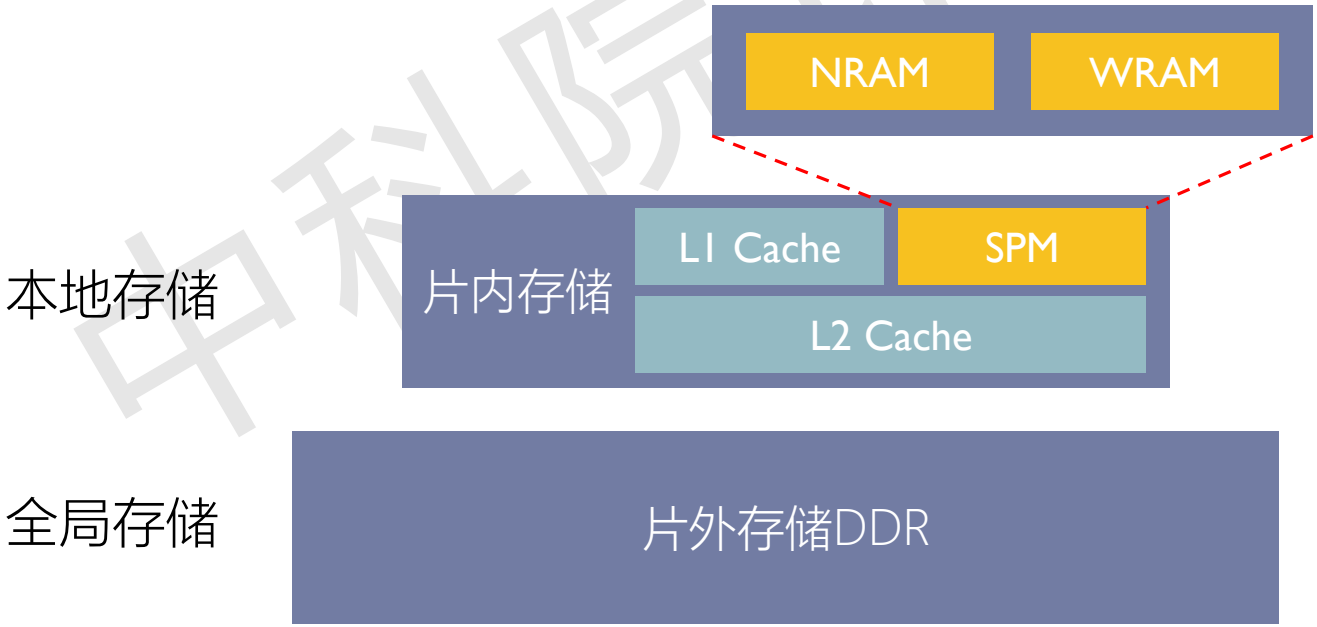






存储模型

- 智能应用中存在大量数据密集的内存访问，因此合理地组织存储层次和计算单元同样重要，需要两者协同设计以平衡计算与访存，实现高效的智能计算
  - 分为**全局存储**和**本地存储**



提纲

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

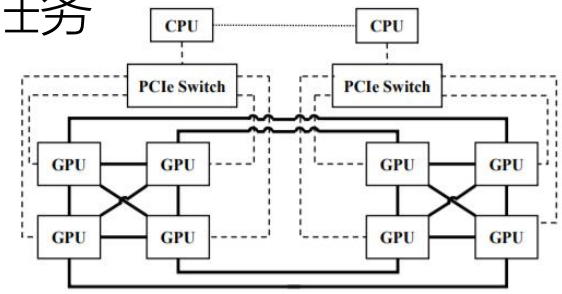
智能编程模型

- 异构编程模型
  - 分类及流程
  - 编译器支持
  - 运行时支持
- 通用智能编程模型
  - Kernel定义
  - 编译器支持
  - 运行时支持
- 智能编程模型实例：BANG异构编程

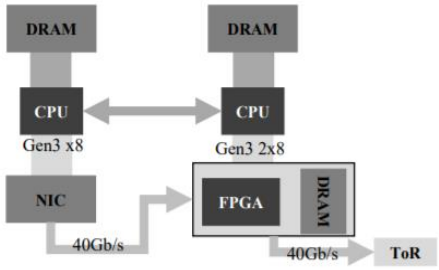
异构编程模型

- 异构计算系统组成
  - 通用处理器：控制设备（简称主机端），负责控制和调度等工作
  - 领域处理器：从设备（简称设备端），负责大规模的并行计算或领域专用计算任务
  - 二者协同完成完整计算任务

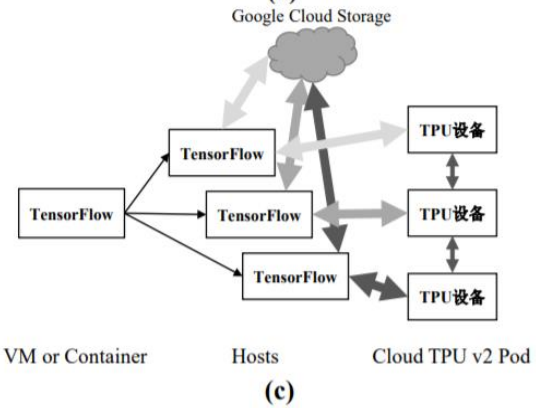
典型异构计算系统
（a）GPU为核心
（b）FPGA为核心
（c）TPU为核心
（d）DLP为核心



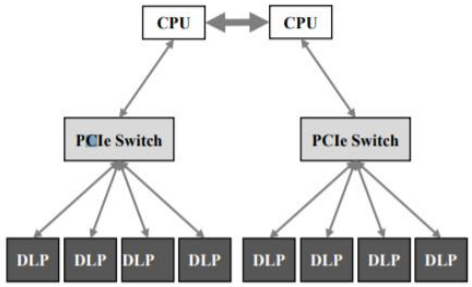
(a)



(b)



(c)



(d)

异构编程模型：分类

- 异构并行编程模型从用户接口角度大致可分为两类：
  - 构建全新的异构并行编程
  - 对现有编程语言进行异构并行扩展

	类别	主要编程考量
OpenCL	语言扩展	任务划分+数据分布、通信、同步
CUDA	语言扩展	任务划分+数据分布、通信、同步
Copperhead	新语言	任务划分为主
Merge	新语言	任务划分为主

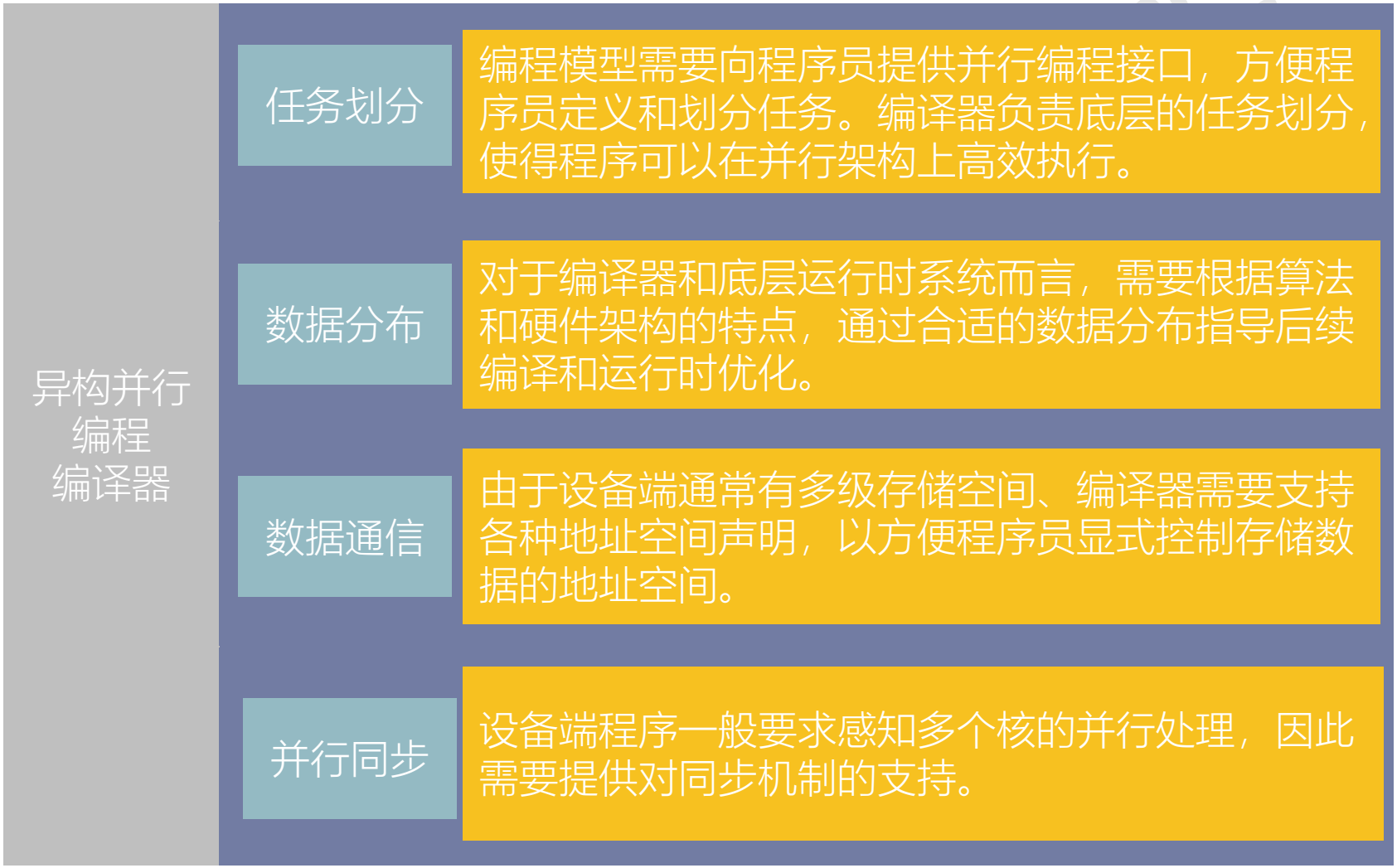
典型异构并行编程模型对比

异构编程模型：流程

- 异构编程模型的编译和链接流程
  - 整体采用分离式编程方式：主机端代码和设备端代码

异构编程模型：编译器支持

- 编译器支持是异构并行编程模型的核心



异构编程模型：运行时支持

- 完成任务映射及调度，即指定任务具体在哪个设备或计算单元上以何种顺序执行
- 分为主机端和设备端
  - 主机端：控制部分和串行任务在主机端执行
  - 设备端：计算部分和并行任务在设备端执行

通用智能编程模型

- 通用智能编程模型以前述异构编程模型为基础
- Kernel函数定义
  - 定义在设备端DLP上的核心计算任务
- 编译器支持
  - 指定DLP上的核心计算任务如何高效地翻译成目标代码
- 运行时支持
  - 指定DLP上的核心计算任务以何种方式映射到计算单元

通用智能编程模型：Kernel定义

- 与异构编程模型中的概念一致，DLP上执行的任务叫Kernel，资源允许情况下DLP可以同时执行多个并行的Kernel
  - 每个Kernel有一个入口函数，BCL中用__dlp_entry__来指定

```
__dlp_entry__ void L2LossKernel(half* input, half* output) {
  ...
}
```

  - Kernel的启动需调用运行时API：InvokeKernel函数

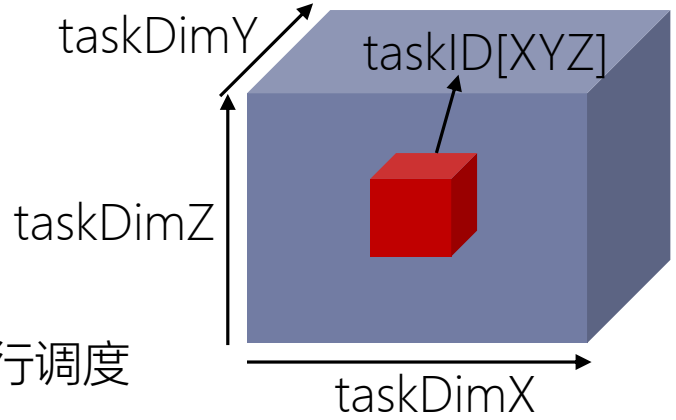
```
ret = InvokeKernel((void *)(&L2LossKernel), dim,
                   params, ft, pQueue);
```

  - 设备端程序默认的函数类型：Device函数，以__dlp_device__来修饰

```
__dlp_device__ void CreateBox(half* box, half* anchor_,
                              half* delt_, int A, int W,
                              int H, half im_w, half im_h)
```

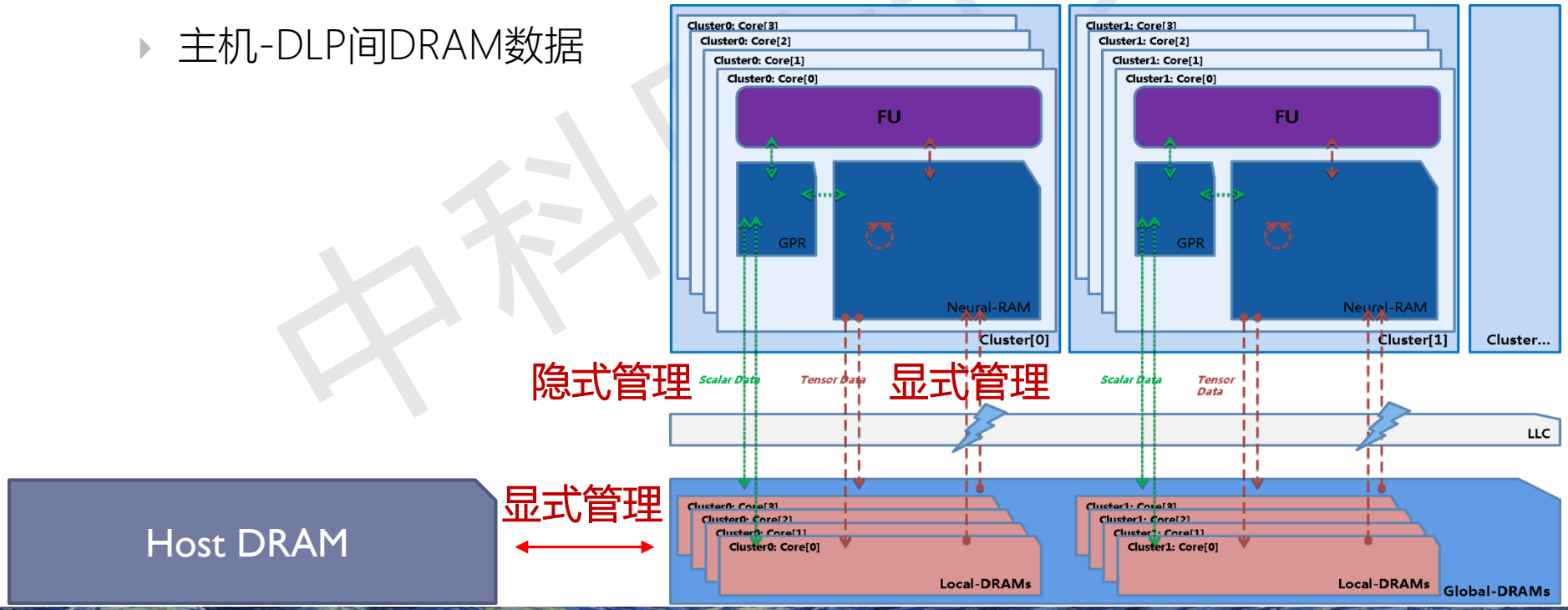
通用智能编程模型：编译器支持

- 任务划分
  - 并行内建变量
    - **硬件**：clusterDim（维度），clusterId（序号），coreDim，coreId
    - **任务**：taskDim[XYZ]，taskId[XYZ]
    - 表示Kernel启动task的规模，有XYZ三个维度，用户根据需求进行指定



  - 任务调度类型
    - 表示Kernel运行调度时需要的硬件核
    - BLOCK类型：Kernel为单核任务，按单核进行调度
    - UNIONx类型：Kernel为多核并行任务（其中x可以为1/2/4，UNION1对应1个cluster4个核）

- 数据通信：为DLP的复杂存储层次提供支持
  - 隐式数据管理：GPR标量数据，由编译器隐式插入Load/Store指令
  - 显式数据管理：
    - DRAM/NRAM/WRAM/SRAM间向量及张量数据
    - 主机-DLP间DRAM数据



- 同步支持：为并行计算架构提供支持

抽象硬件架构中有Chip-Cluster-Core的层次结构，可以提供至少两种不同类型的同步操作：

- __sync_all：**同步任务执行的所有核**，只有所有核到达同步点时才继续往下执行。参与同步的核数由任务的调度类型确定，例如当任务类型为UNION1时，只有4个核参与同步（假定一个Cluster内包含4个核），当任务类型为UNION2时，参与同步的核数为8。

- __sync_cluster：**同步一个Cluster内部的所有核**，当一个Cluster内所有核都达到同步点时才继续往下执行。与CUDA相比，GPU同一个线程块内的thread可以同步，而线程块间的thread无法同步。

- 内建运算：为用户编程提供支持，提高开发效率
  - 通用智能编程语言提供并实现了__conv和__mlp等内建函数接口，分别对应卷积和全连接等典型神经网络运算
  - 上述接口是对C/C++语言的扩展，深度学习处理器端Kernel程序编写时可以调用这些接口，通过编译器将这些接口翻译为底层硬件指令
  - 通用智能编程模型直接实现了神经网络计算的内建接口，能更好地支持智能应用。

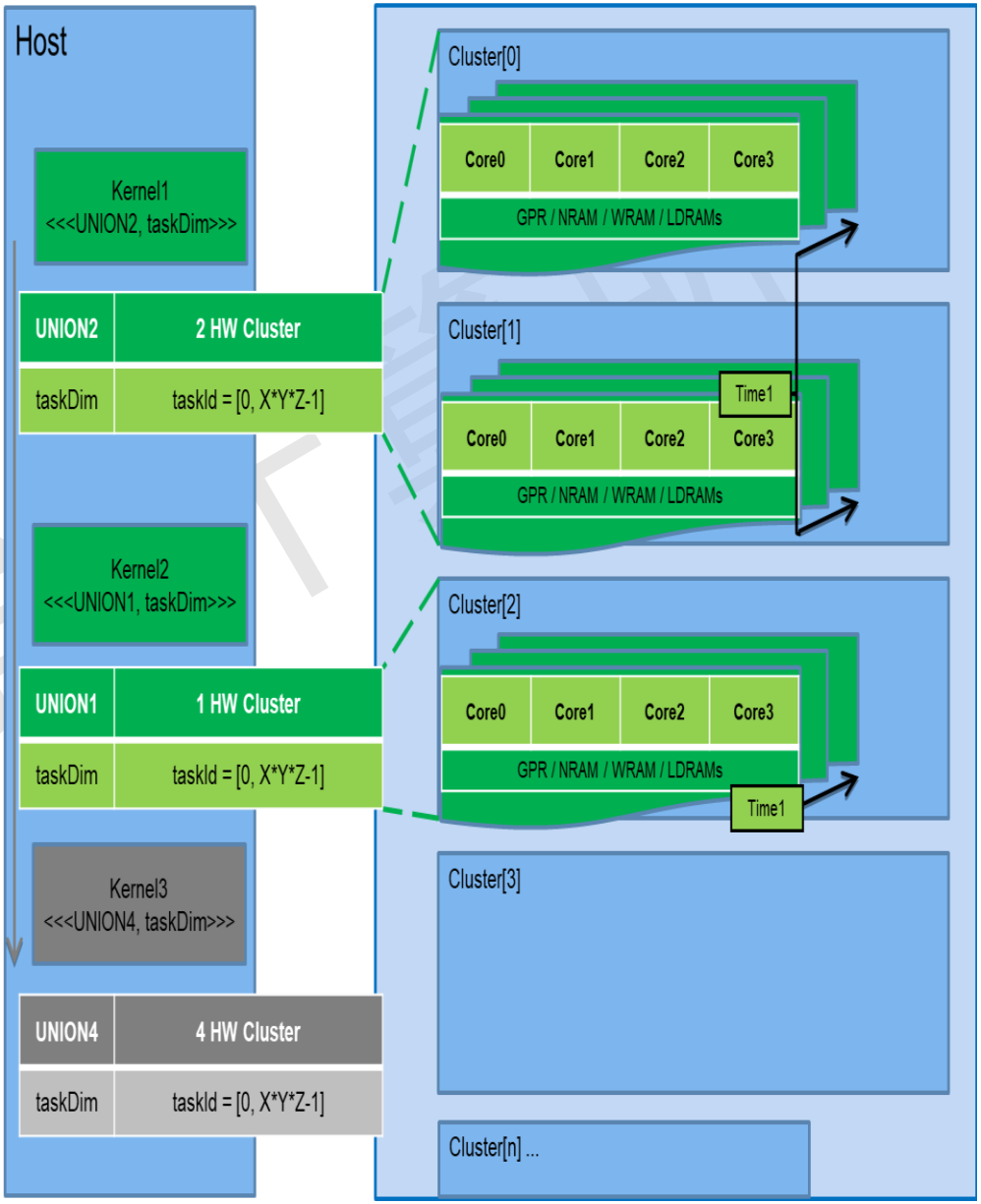
通用智能编程模型：运行时支持

- 任务调度单位（BLOCK/UNIONx）
  - 以调度单位将Kernel中的任务在时间或空间维度展开
    - BLOCK：单核调度，当有一个核空闲时，调度一个任务执行
    - UNION1：调度时需要1个cluster，当有1个cluster空闲时，调度任务执行
    - UNION2：调度时需要2个cluster，当有2个cluster空闲时，调度任务执行
  - 调度单位需要用户在编程时指定。运行时只有当空闲的硬件资源数大于调度单位时，Kernel 才会被调度
- 队列（Queue）
  - 管理需要执行的任务，队列既可以单独工作，也可以协同工作
  - 运行时（或硬件）不断把任务放到队列中，一旦硬件计算资源有空闲，就从队列中取出一个任务执行

- 主机端执行3个Kernel任务
- 设备端有4个Cluster，16个Core

1、**主机端异步发射3个Kernel到Queue中。**用户可以根据同步和通信的需要，在三次发射之间或之后任意位置调用同步接口SyncQueue。假设在三次发射之后调用，则等待Queue中的任务全部完成后再继续执行主机端SyncQueue后面的程序；

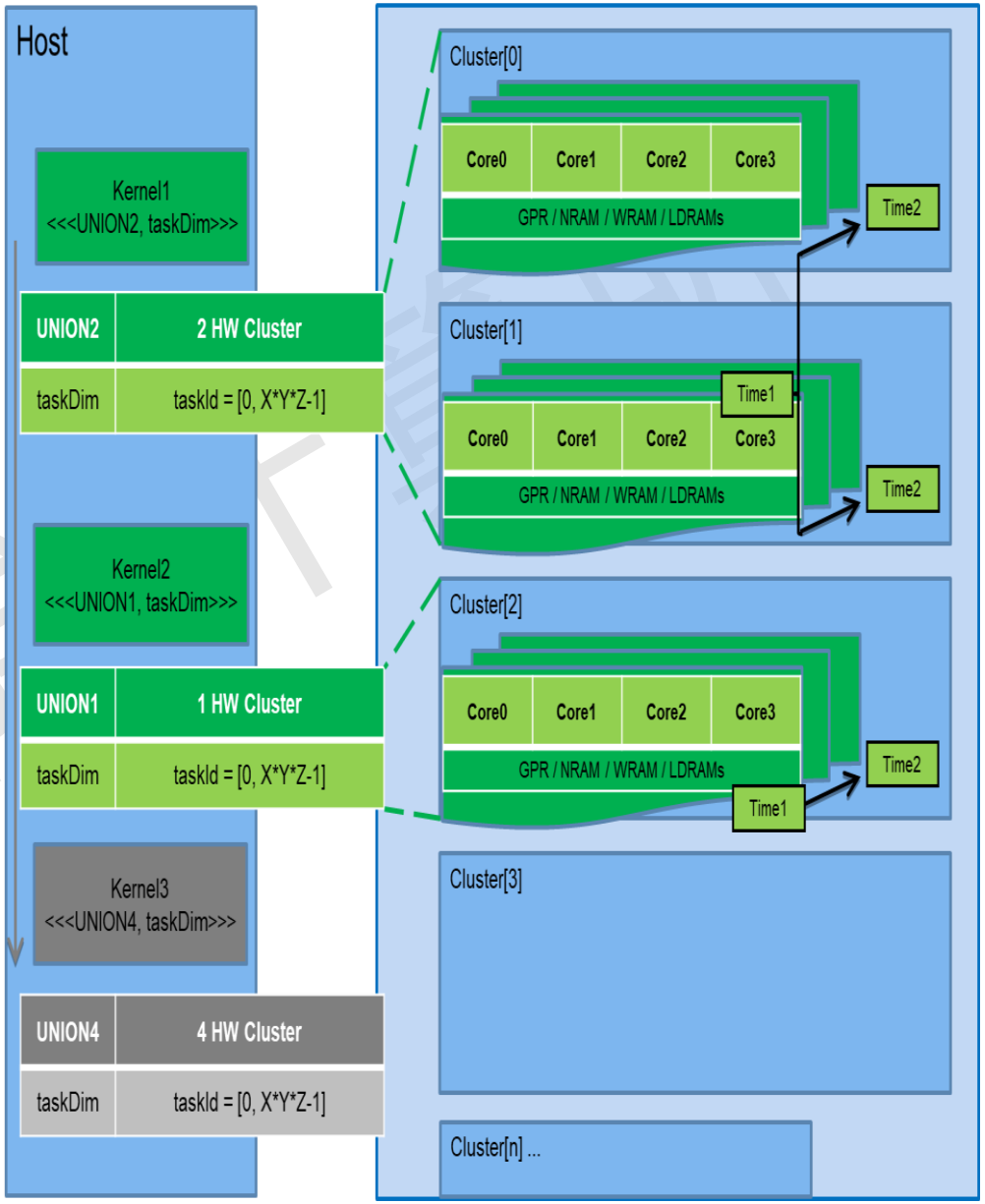
2、第一个任务Kernel1在Time1被发射后立即进入Queue，设备端发现当前全部核心空闲则立即执行Kernel1。**Kernel1的任务类型为UNION2，会从Time1开始占用2个Cluster**执行计算；



3、因没有调用SyncQueue，所以主机端发射Kernel1后立即发射Kernel2，设备端调度器在调度执行Kernel1后发现队列中有了新的Kernel2，也几乎在Time1时刻开始执行Kernel2。Kernel2的任务类型为UNION1，会从Time1开始占用1个Cluster计算；

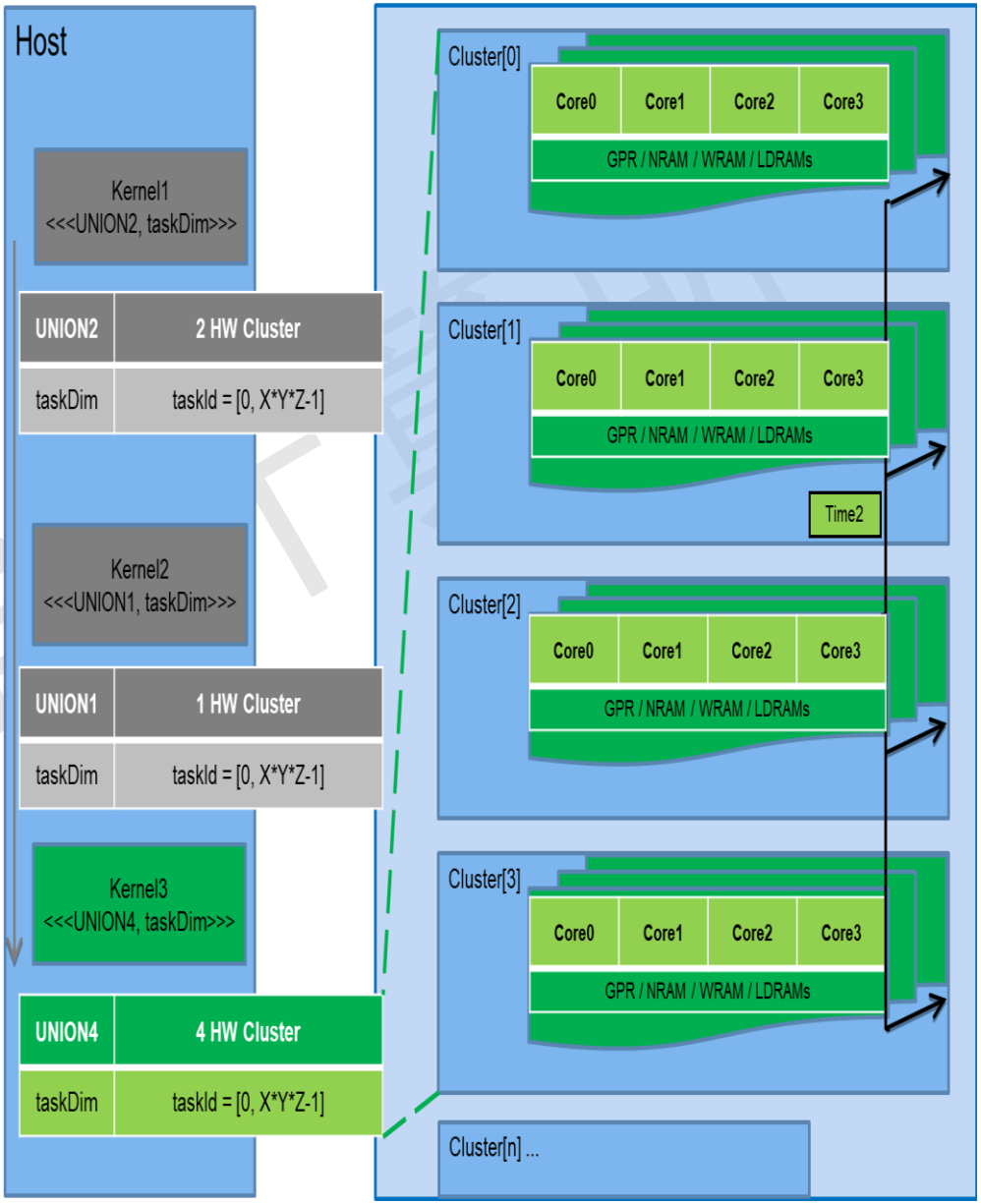
4、假设Kernel1和Kernel2的任务并行总规模taskDim超过了任务类型表示的核数（例如kernel1是UNION2则一次要占用8个Core），则调度器会将同一份Kernel程序在时间序上执行多次；

5、Kernel1和Kernel2几乎同时被调度器执行且假设同时在Time2时刻结束；



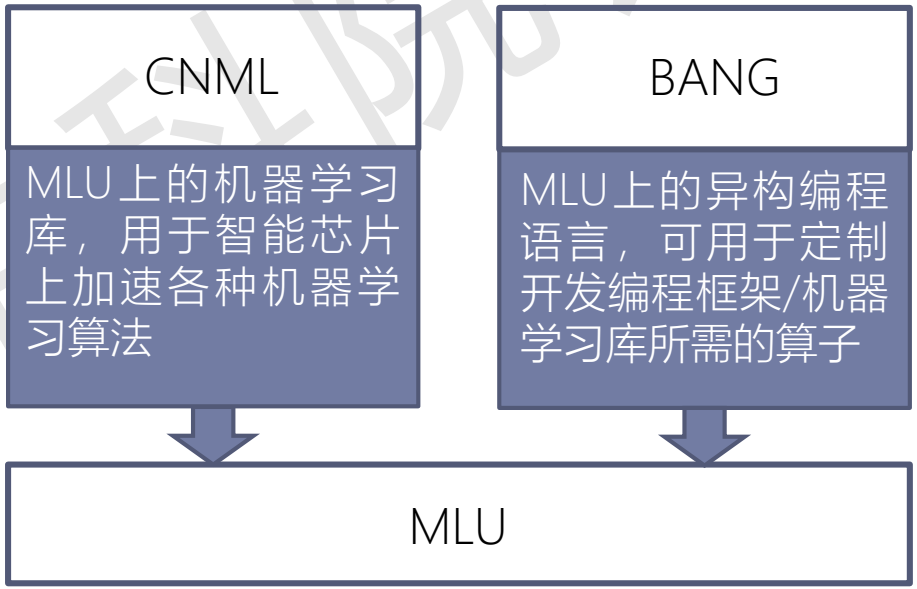
6、回到第4步中主机端的执行流程，当Kernel2被发射后，因为没有执行同步，所以Kernel3也会立即被发射，此时刻也几乎为Time1;

7、Kernel3的任务类型是UNION4，需要4个Cluster，但在Time1时刻到Time2时刻硬件的4个Cluster被占用了3个（假设只有4个Cluster），那么设备端调度器会一直等待Time2时刻有4个Cluster的空闲时才执行Kernel3。



智能编程模型实例：BANG异构编程

- BANG语言是针对 MLU（Machine Learning Unit）硬件提出的编程语言，兼顾云边端等不同目标平台，提供高性能机器学习计算支持
  - 提供通用的异构编程模型，方便用户扩展自己的应用程序
  - 提供高效的编程接口，充分发挥底层硬件特性
  - 基于C/C++语言的扩展，简单易用



BANG异构编程：流程

- BANG异构程序同样采用分离式编程，即主机端与设备端程序分开编程并分别编译，最后链接成一个可执行程序
- 主机端为C/C++程序，通常需调用CNRT运行时接口完成以下步骤

① 准备输入数据

② 拷贝输入数据到MLU

③ 准备Kernel参数

④ 创建Queue

⑤ 指定Kernel任务规模以及调度类型

⑥ 启动Kernel

⑦ MLU到主机的输出数据拷贝

⑧ 资源的释放

常用CNRT接口名称	功能描述
cnrtInit	在当前系统中初始化Runtime环境。
cnrtGetDeviceHandle	获取设备的Handle
cnrtSetCurrentDevice	设置当前设备
cnrtDestroy	释放Runtime API环境上下文，当不再使用Runtime函数时，调用该方法执行清理工作
cnrtCreateQueue	创建一个新的Queue，默认异步运行
cnrtDestroyQueue	销毁Queue
cnrtSyncQueue	直到之前Queue中所有的Function都完成，阻塞其他Function
cnrtGetKernelParamsBuffer	获取参数块用于cnrtInvokeKernel
cnrtKernelParamsBufferAddParam	为一个特定的参数缓冲区增加一个参数对象
cnrtDestroyKernelParamsBuffer	销毁参数块
cnrtInvokeKernel	通过在MLU上给定的参数块，启动Kernel
cnrtMalloc	分配给定空间的设备内存
cnrtFree	释放指针指向的空间
cnrtMemcpy	从源地址拷贝指定字节数据到目的地址

- 设备端使用 BANG 语言特定的语法规则和接口进行编程

BANG异构编程示例：主机端

主机端调用cnrt接口，需要include “cnrt.h”

设备初始化

创建Queue

指定任务规模和调度类型：BLOCK类型

准备kernel的参数

在GDRAM上为输入数据分配空间

输入数据拷入GDRAM

启动Kernel，并绑定到Queue

同步Queue的计算结果

数据拷出

释放资源

```
#include “cnrt.h”
...
cnrtInit(0);
cnrtDev_t dev;
cnrtGetDeviceHandle(&dev, 0);
cnrtSetCurrentDevice(dev);
cnrtQueue_t queue;
cnrtCreateQueue(&queue);
cnrtDim3_t dim; dim.x = 1; dim.y = 1; dim.z = 1;
cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK;
cnrtKernelParamsBuffer_t params;
...
uint32_t param1 = SIZE_INPUT;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &param1, sizeof(uint32_t));
...
half *ptr_mlu_input;
cnrtMalloc((void **)&ptr_mlu_input, N1 * sizeof(half)))
cnrtMemcpy(&ptr_mlu_input prt_host_input, CNRT_MEM_TRANS_DIR_HOST2DEV);
...
cnrtInvokeKernel((void *)&kernel, dim, params, func_type, queue);
cnrtSyncQueue(queue);
cnrtMemcpy(ptr_host_output, ptr_mlu_output, SIZE_OUTPUT,
                CNRT_MEM_TRANS_DIR_DEV2HOST);
cnrtDestroyQueue(queue);
cnrtDestroyKernelParamsBuffer(params);
cnrtDestroy();
```

BANG异构编程示例：设备端

MLU端程序是基于C语言的扩展，其文件名后缀为mlu

MLU端程序需要包含mlu.h头文件

BANG程序kernel的入口函数需要用__mlu_entry__标记

__nram__标记nram空间的变量

将数据从GDRAM拷贝到NRAM上

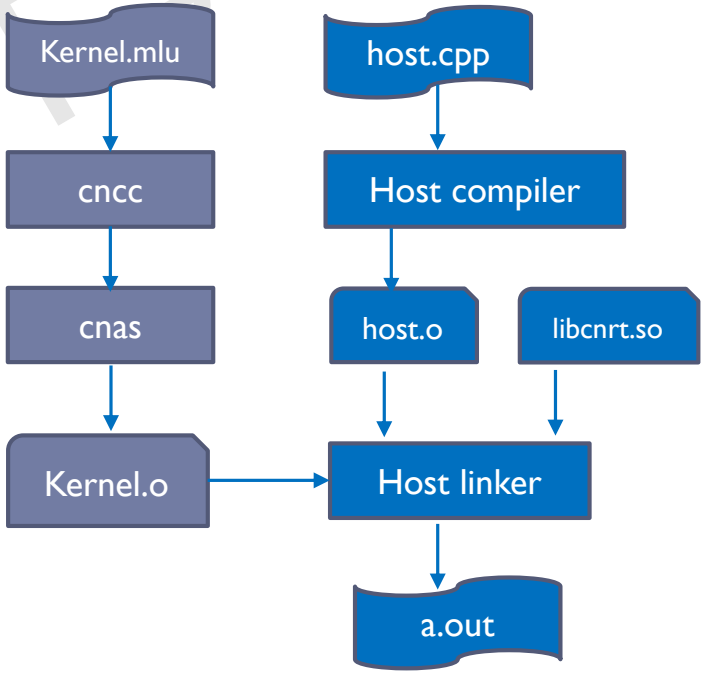
将NRAM上数据进行cycle_add计算

将计算的结果从NRAM拷贝到GDRAM，以供主机端程序读取

```
//file: hello.mlu

#include "mlu.h"
#include "macro.h"
#define s1 N1
#define s2 N2
__mlu_entry__ void kernel(uint32_t size1, uint32_t size2,
                          half* c, half* a, half* b) {
    __nram__ half a_tmp[s1];
    __nram__ half b_tmp[s2];
    __nram__ half c_tmp[s2];
    __memcpy(a_tmp, a, s1 * sizeof(half), GDRAM2NRAM);
    __memcpy(b_tmp, b, s2 * sizeof(half), GDRAM2NRAM);
    __bang_cycle_add(c_tmp, b_tmp, a_tmp, size2, size1);
    __memcpy(c, c_tmp, s2 * sizeof(half), NRAM2GDRAM);
}
```

BANG程序编译与链接

- 基于前述异构编程模型的介绍，BANG程序的编译和链接也采用分离的方式，如下图所示：
  - MLU端程序采用编译器CNCC进行编译，得到MLU程序的目标文件
  - Host端程序采用普通的C/C++编译器GCC/CLANG等进行编译，得到Host程序的目标文件
  - 最后使用Host端链接器将MLU与Host程序的目标文件以及CNRT库，链接成Host端的可执行文件



提纲

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

智能编程语言基础

- 语法概述
- 数据类型
- 宏、常量与内置变量
- I/O操作语句
- 标量计算语句
- 张量计算语句
- 控制流语句
- 串行程序示例
- 并行程序示例
- 张量计算语句实例：BANG数学库

语法概述

- 智能编程语言考虑基于过程式语言
  - 当前大多数语言都是过程式的，可以减少用户学习成本
  - 当前主流智能算法可以描述为明确的过程，适合采用过程式语言描述

- 参考经典的过程式语言C/C++，我们所定义的智能编程语言同样具有数据和函数两个基本要素
  - 例如：定义add_func函数的函数体和C语言一致

```
int add_func (int a, int b) {
  int c = a + b;
  return c;
}
```

基本数据类型

基本数据类型	长度	说明
int8_t	1byte	1 字节整数
uint8_t	1byte	1 字节无符号整数
int16_t	2 byte	2 字节整数
uint16_t	2 byte	2 字节无符号整数
int32_t	4 byte	4 字节整数
uint32_t	4 byte	4 字节无符号整数
half	2 byte	半精度浮点数据类型，采用IEEE-754 fp16格式
float	4 byte	IEEE-754 fp32格式浮点类型,目前仅支持类型转换计算
char	1 byte	对应C语言char类型
bool	1 byte	对应C语言bool类型
指针	8 byte	指针类型

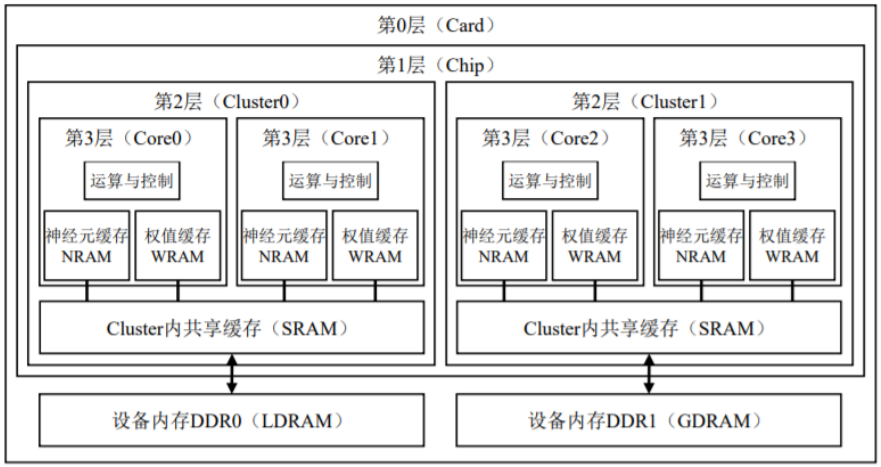
宏、常量与内置变量

- 宏/常量由用户定义，内置变量是语言既有的
  - 宏和常量，宏不仅可以定义常量数据，也可定义一段代码；常量是不可修改的数据，只能在初始化时被赋值。
  - 内置变量，编程语言本身包含的常量和变量，不需用户定义即可直接使用。

内置变量名	具体含义
coreId	DLP中核的编号
clusterId	DLP中簇的编号
taskId	程序运行时分配的任务编号

I/O操作语句

- 不同层次的智能处理节点有各自的本地存储，需要提供不同存储层次间的数据搬移



典型的 NUMA（Non-Uniform Memory Access Architecture）架构

- 不同处理器核对不同位置存储器的访问速度不同。对于核0而言，其访问设备内存 DDR0 的速度比访问 DDR1 的速度更快。DDR0看作是本地存储，DDR1作为全局存储
- 片上存储，除了单核内NRAM和权值WRAM，还有一类共享存储，可用于簇内的多核共享

- 针对上述典型架构（三种片上存储、两种设备内存），可以有多种不同的数据搬移操作类型

NRAM	NRAM <-> GDRAM NRAM <-> LDRAM NRAM <-> SRAM
WRAM	WRAM <-> GDRAM WRAM <-> LDRAM WRAM <-> SRAM
SRAM	SRAM <->GDRAM SRAM <-> LDRAM SRAM <-> SRAM
NRAM	NRAM <-> NRAM

- 针对上述搬移操作类型，可以在智能编程语言中定义相应的内建函数__memcpy，方便用户进行不同类型的数据搬移

```
void __memcpy(void* dst, void* src, uint32 bytes, Direction_t dir);
```

标量计算语句

- 标量即单个数据的计算，标量计算是编程语言的基本功能
  - 智能编程语言的标量计算语句有两种形式：
    - 运算符号（如 +，-，*，/等）
    - 内建函数（如 abs，max，min 等）

- 智能编程语言的标量计算语句由编译器映射到标量计算单元，虽然吞吐上不及张量运算，但具有良好的通用性和灵活性

张量计算语句

- 张量计算是智能编程语言的主要特点，可以通过内建函数直接映射到张量计算单元
- 张量计算直接对精度类型和语义类型等数据直接进行操作

张量计算语句	具体功能
__vec_add(float* out, float *in1, float* in2， int size)	向量对位加
__vec_sub(float* out, float *in1, float* in2， int size)	向量对位减
__vec_mul(float* out, float *in1, float* in2， int size)	向量对位乘
__conv(half* out, int8* in, int8* weight, half bias, int ci, int hi, int wi, int co, int kh, int kw, int sh, int sw)	卷积运算
__mlp(half* out, int8* in, int8* weight, half* bias, int ci, int co)	全连接运算
__maxpool(half* out, half* in, int ci, int hi, int wi, int kh, int kw, int sh, int sw)	最大池化运算

控制流语句

- 与通用编程语言一样，智能编程语言同样需要有分支和循环等控制流语句
  - **分支语句**：用于处理程序的选择逻辑，由判断条件与分支代码段组成，与传统编程语言类似
  - **循环语句**：用于处理程序循环逻辑，由循环执行条件和循环代码段组成，与传统编程语言类似
  - **同步语句**：解决多核间并行数据依赖问题，保证最终计算结果正确。主要分为两类：Cluster内同步（__sync_cluster）与全局同步（__sync_all），Cluster 内同步只保证一个 Cluster 内的所有核同步，而全局同步则是芯片内所有Cluster、所有核都进行同步。

串行程序示例

- 向量中每个数求平方，每次处理64个数

```
#define BASE_NUM 64
void __dlp_entry__ mySquare(float* in, float* out, int size) {
  int quotient = size / BASE_NUM;
  int remainder = size % BASE_NUM;
  __nram__ float tmp[BASE_NUM];

  for (int i = 0; i < quotient; i++) {
    __memcpy(tmp, (in + i * BASE_NUM), (BASE_NUM * sizeof(float)), GDRAM2NRAM);
    __vec_mul(tmp, tmp, tmp, BASE_NUM);
    __memcpy((out + i * BASE_NUM), tmp, (BASE_NUM * sizeof(float)), NRAM2GDRAM);
  }

  if (remainder != 0) {
    __memcpy(tmp, (in + quotient * BASE_NUM), (remainder * sizeof(float)), GDRAM2NRAM);
    __vec_mul(tmp, tmp, tmp, remainder);
    __memcpy((out + quotient * BASE_NUM), tmp, (remainder * sizeof(float)), NRAM2GDRAM);
  }
}
```

并行程序示例

- 矩阵乘法示例，在4个核上并行执行，每个核上代码一致
  - 计算1*32和32*32的矩阵乘法，最终得到4*32的结果
  - 通过运行时API来指定任务规模（4*1*1）及调度方式（UNION1）

任务规模

```
Dim_t dim;  dim.x = 4;  dim.y = 1;  dim.z = 1;
int left[4][32];  int right[32][32];  int out[4][32];
```

… …

```
KernelParamsBuffer_t params;
GetKernelParamsBuffer(&params);
KernelParamsBufferAddParam(params, &left_dev, sizeof(void*));
KernelParamsBufferAddParam(params, &right_dev, sizeof(void*));
KernelParamsBufferAddParam(params, &out_dev, sizeof(void*));
```

调度类型

… …

```
// 启动4个核并行执行矩阵乘法
InvokeKernel((void*)(&mm), dim, params, UNION1, queue);
SyncQueue(queue);
```

```
void __dlp_entry__ mm(int* left, int* right, int* out) {
  if (taskID == 0) {
    __nram__ int tmp[4][32];
    __write_zero(tmp, 4*32*sizeof(int));
    __memcpy(out, tmp, 4*32*sizeof(int), NRAM2GDRAM);
  }

  __sync_all();
  for (int j = 0; j < 32; j++) {
    for (int k = 0; k < 32; k++) {
      out[taskIdX * 32 + j] += left[taskIdX * 32 + k] * right[k * 32 + j];
    }
  }
}
```

设备端

主机端

张量计算语句实例：BANG数学库

- BANG语言将MLU的数学类和神经网络类指令封装成库函数
- BANG数学库函数是在MLU架构上进行高性能编程的关键
- BANG数学库头文件

  /usr/local/neuware/lib/clang/x.x.x/include/__clang_bang_math.h

- **使用约束**：张量计算通常是对批量数据进行操作
  - 源和目的都是NRAM上的数据
  - 数据的长度（字节数）必须是128的整数倍
  - 源和目的地址必须按64位对齐

BANG数学库实例

- 接口：__bang_add(half* dst, half* src0, half* src1, int32_t NumOfEle)
  - 语义：两个向量相加，得到新的向量
  - 约束：源和目的都是half类型

- 接口：__bang_mul_const(half* dst, half* src0, half scale, int32_t NumOfEle)
  - 语义：对向量的每一个元素乘以一个常数

BANG数学库实例

- 接口：__bang_eq(half* dst, half* src0, half* src1, int32_t NumOfEle)
  - 语义：对src0和src1两个张量进行element-wise的eq操作，并将结果存入dst中
- 接口：__bang_select(half* dst, half* src0, half src1, int32_t NumOfEle)
  - 语义：如果src1中某个元素非0，则从src0中选择对应位置元素存入dst中。dst中的结果有两部分，第一部分是选出的元素个数，第二部分是选出的元素集合
  - 示例：

```
__bang_select(dst, src0, src1, NumOfEle);
...
// get the number of selected elements from the first line
uint16_t numberOfSelectedNumbers = *((uint16_t*)dst); // dst[0]
// get the selected elements from the second line
selectedNum[0] = dst[16];
selectedNum[1] = dst[17];
```

BANG数学库实例

- 接口：__bang_max(half* dst, half* src，int32_t NumOfEle)
  - 语义：从张量src中选择最大值的元素，存入dst中，结果有两部分，第一部分是选出的元素，第二部分是选择元素在src中index
  - 示例:

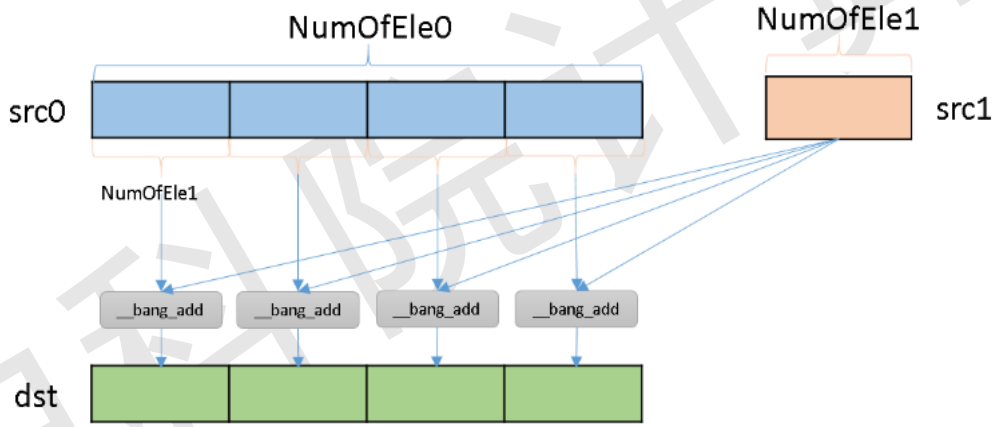
```
__bang_max(dst, src, NumOfEle);
...
half maxNumber = dst[0];
uint16_t indexOfMaxNumber = ((uint16_t*)dst )[1];
```

- 接口：__bang_count(uint16_t* dst, half* src, int32_t NumOfEle)
  - 语义：计算src中非0元素的个数，结果存入dst中
  - 示例：

```
__bang_count(dst, src, NumOfEle);
...
uint16_t counter = ((uint16_t*)dst) [0];
```

BANG数学库实例

- 接口：__bang_cycle_add(half* dst, half* src0, half* src1, int32_t NumOfEle0, int32_t NumOfEle1)
  - 语义：将src0元素分成整数段，每段元素与src1元素个数相等，然后将每一段与src1进行向量add操作，结果存入dst中



- 接口：__bang_transpose(half*dst, half*src, const int32_t h, const int32_t w)
  - 语义：将src[h][w]转置成dst[w][h]

BANG数学库实例

- 接口： __bang_maxpool (half*dst, half*src, const int32_t ic, const int32_t ih, const int32_t iw, const int32_t kh, const int32_t kw [,const int32_t sx, const int32_t sy])
  - 语义：对src进行maxpool操作，结果存入dst中，其中：
  - src[ih, iw, ic]
  - slide-window[kh, kw]
  - stride [sx, sy]，如果没有指定，则使用[kh, kw]作为stride
  - dst[h,w,c]

BANG数学库实例

- 接口： __bang_maxpool_index(uint16_t* dst, half*src, const int32_t ic, const int32_t ih, const int32_t iw, const int32_t kh, const int32_t kw [, const int32_t sx, const int32_t sy])
  - 语义： 与__bang_maxpool语义类似，不同是每次滑窗时选择最大元素的index而非最大元素本身，
  - src[ih, iw, ic]
  - slide-window[kh, kw]
  - stride [sx, sy]，如果没有指定，则使用[kh, kw]作为stride
  - dst[h,w,c]

提纲

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

智能应用编程接口

- Kernel函数接口
- 运行时接口
- 使用示例
- 编程接口实例：BANG程序

Kernel函数接口

- 为了充分利用并行资源，需要在Kernel内部对任务进行有效切分，同时在主机端配置和调用相应的Kernel函数接口
- 任务切分的内置变量

变量	说明
coreDim（核维数）	内置变量，等于单个 Cluster 内部的计算核个数。
coreId（核序号）	内置变量，对应每个硬件计算核在 Cluster 内的逻辑 ID，取值范围为 [0, coreDim - 1]。
clusterDim（簇维数）	内置变量，等于 Chip 内的 Cluster 个数。
clusterId（簇序号）	内置变量，对应程序运行所在 Cluster 的逻辑 ID，取值范围为 [0, clusterDim - 1]。
taskDim（任务维数）	内置变量，等于当前用户指定任务的总规模，taskDim = taskDimX × taskDimY × taskDimZ。
taskId（任务序号）	内置变量，对应程序运行时所分配的任务 ID，取值范围为 [0, taskDim - 1]。taskId 的值对应逻辑规模降维后的任务 ID，即 taskId = taskIdZ × taskDimY × taskDimX + taskIdY × taskDimX + taskIdX。

- 主机端Kernel函数接口
  - 用于将智能编程语言编写的程序加载到深度学习处理器上执行；与Kernel函数相关的接口主要关注Kernel参数设置和Kernel调用
  - GetKernelParamBuffer(KernelParamsBuffer_t *params)

获取 Kernel 的参数块结构，成功返回 RET_SUCCESS，否则返回相应错误码

  - CopyKernelParamsBuffer(KernelParamsBuffer_t dstbuf, KernelParamsBuffer_t srcbuf)

拷贝 srcbuf 到 dstbuf，成功返回 RET_SUCCESS，否则返回相应错误码。

  - KernelParamsBufferAddParam(KernelParamsBuffer_t params, void* data, size_t bytes)

向 KernelParamsBuffer_t 中增加常量参数，成功返回 RET_SUCCESS，否则返回相应错误码

- 主机端Kernel函数接口
  - DestroyKernelParamsBuffer(KernelParamsBuffer_t params)

销毁 KernelParamsBuffer_t 变量，成功返回 RET_SUCCESS，否则返回相应错误码

  - InvokeKernel(const void * function, Dim3_t dim, KernelParamsBuffer_t params, FunctionType_t funcType, Queue_t queue);

通过在设备上给定的参数块，调用 Kernel，成功返回 RET_SUCCESS，否则返回相应的错误码。其中 FunctionType_t 包含 BLOCK 和 UNIONx（如 UNION1 和 UNION2，取决于系统的规模）等类型，分别表示单核任务和多核并行任务。 由于调用 InvokeKernel 时，会有函数参数从主机端到设备端的拷贝过程，所以当追求高性能时，要尽可能减少 InvokeKernel 次数。

运行时接口

- 包括设备管理、队列管理和内存管理等接口
- 设备管理：主要涉及初始化、设备设置、设备销毁等操作
  - Init(unsigned int flags);
  - GetDeviceCount(unsigned int* dev_num);
  - GetDeviceHandle(Dev_t* pdev, int ordinal);
  - SetCurrentDevice(Dev_t dev);
  - Destroy(void);

队列管理

队列是用于执行任务的环境。计算任务可以下发到队列中执行。同一个队列可以容纳多个任务。具体来说，队列具有以下属性：

- **串行性**：下发到同一个队列中的任务，按下发顺序串行执行
- **异步性**：任务下发到队列是异步过程，即下发完成后程序控制流回到主机，主机程序继续往下执行。运行时环境提供队列的同步接口SyncQueue用于等待队列中所有任务完成。
- **并行性**：不同队列中的任务并行执行。 如果希望任务间并行执行，用户可以创建多个队列并将任务分配到不同的队列中。
- CreateQueue(Queue_t *queue);
- SyncQueue(Queue_t queue);
- DestroyQueue(Queue_t queue);

内存管理

内存管理主要分为**主机端内存管理**、**设备端内存管理**和**主机与设备端内存拷贝**三类：

- hostMalloc(void ** ptr, size_t bytes, ...)
- hostFree(void *ptr)
- devMalloc(void ** ptr, size_t bytes)
- devFree(void *ptr)
- Memcpy(void *dst, void* src, size_t bytes, MemTransDir_t dir)

使用实例

- 首先完成Device端的Kernel函数编写

```
__dlp_entry__ void kernel(int *input, int len, int *output) {
  int sum = 0;
  for (int i = 0; i < len; i++) {
    sum += input[i];
  }
  *output = sum;
}
```

注意：对每个智能编程语言编写的程序，有且仅有一个标记为 __dlp_entry__ 的核函数，表示整个 Kernel 函数的入口，其返回值类型必须是 void

- Host端代码

① 设备初始化

```
Init(0);
Dev_t dev;
GetDeviceHandle(&dev, 0);
SetCurrentDevice(dev);
Queue_t pQueue;
CreateQueue(&pQueue);
Dim3_t dim;
dim.x = 1;
dim.y = 1;
dim.z = 1;
```

设备在使用前都需要初始化，主要工作包括：查询可用设备、选择某设备执行及设置任务规模等

- Host端代码

② 主机/设备端数据准备

③ 设备端内存空间分配

```
half *d_input;
half *d_output;
half *dlp_result;
hostMalloc(dlp_result, data_num * sizeof(half));
devMalloc((void **)&d_input, data_num * sizeof(half)));
devMalloc((void **)&d_output, data_num * sizeof(half)));
```

④ 数据至设备端拷贝

```
Memcpy(d_input, h_a_half, sizeof(half)*data_num, HOST2DEV);
```

- Host端代码

⑤ 调用 Kernel 启动设备

```
KernelParamsBuffer_t params;
GetKernelParamsBuffer(&params);
KernelParamsBufferAddParam(params, &d_input, sizeof(half *));
KernelParamsBufferAddParam(params, &size, sizeof(uint32_t));
KernelParamsBufferAddParam(params, &d_output, sizeof(half *));
// 在设置好核函数参数之后，就可以调用InvokeKernel接口启动计算任务
InvokeKernel((void *)&kernel, dim, params, func_type, pQueue);
SyncQueue(pQueue);
```

准备参数

⑥ 运行结果获取

```
Memcpy(dlp_result, d_output, data_num * sizeof(half), DEV2HOST);
```

⑦ 资源释放

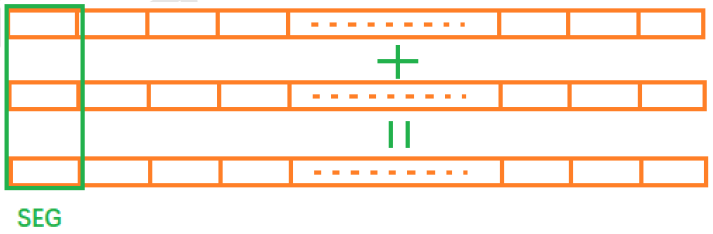
```
devFree(d_input);
devFree(d_output);
hostFree(dlp_result);
DestroyQueue(pQueue);
DestroyKernelParamsBuffer(params);
```

编程接口实例：BANG程序

- BANG单核向量加法
- BANG多核向量加法
- BANG矩阵乘法
- 涉及CNRT的编程接口：包括Kernel控制和运行时接口等

BANG单核向量加法

```
#include "mlu.h"
#define N 65536
#define SEG 256
__mlu_entry__ void kernel(half *a, half *b, half *pc){
 __nram__ half a_cram[N];
 __nram__ half b_cram[N];
 __nram__ half out_cram[N];
 int m = (N+SEG-1)/SEG;
 __memcpy(a_cram, a, N*sizeof(half), GDRAM2NRAM);
 __memcpy(b_cram, b, N*sizeof(half), GDRAM2NRAM);
 //add
 for(int i=0; i<m; i++)
 __bang_add(out_cram+i*SEG, a_cram+i*SEG,b_cram+i*SEG,SEG);
 //mcpy out
 __memcpy(pc,out_cram,N*sizeof(half), NRAM2GDRAM);
}
```



设备端代码：BANG实现

BANG单核向量加法

初始化输入

初始化设备

Float2Half

```
int main{
  cnrtInit(0);
  cnrtDev_t dev;
  cnrtGetDeviceHandle(&dev, 0);
  cnrtSetCurrentDevice(dev);
  cnrtQueue_t pQueue;
  cnrtCreateQueue(&pQueue);
cnrtDim3_t dim;
  dim.x = CORE_NUM;
  dim.y = 1;
  dim.z = 1;
  printf("--0--\n");
... ...
```

用户通过 cnrtDim3_t 指定KERNEL任务的规模：CORE_NUM设置为1

把输入数据送入DEVICE

```
cnrtFunctionType_t cc = CNRT_FUNC_TYPE_BLOCK;
half *half_array1 = (half*)malloc(N*sizeof(half));
half *half_array2 = (half*)malloc(N*sizeof(half));
srand((unsigned)time(NULL));
float *float_array1 = (float*)malloc(N*sizeof(float));
float *float_array2 = (float*)malloc(N*sizeof(float));
for(int i=0; i<N; i++){
  float_array1[i] = 0.5;
  float_array2[i] = 1.5;
}
for(int i=0; i<N; i++){
    cnrtConvertFloattoHalf(&half_array1[i], float_array1[i]);
    cnrtConvertFloattoHalf(&half_array2[i], float_array2[i]);
}
 half* mlu_input1, *mlu_input2,*mlu_ptrc;
if(CNRT_RET_SUCCESS != cnrtMalloc((void **)&mlu_input1, N*sizeof(half))){
   printf("cnrtMalloc FAILED!\n");
   exit(-1);
}
 if(CNRT_RET_SUCCESS != cnrtMalloc((void **)&mlu_input2,
N*sizeof(half))){
   printf("cnrtMalloc FAILED!\n");
   exit(-1);
}
 if(CNRT_RET_SUCCESS != cnrtMalloc((void **)&mlu_ptrc, N*sizeof(half))){
   printf("cnrtMalloc FAILED!\n");
   exit(-1);
}
 if(CNRT_RET_SUCCESS != cnrtMemcpy(mlu_input1, half_array1,
                                   N*sizeof(half),
                                   CNRT_MEM_TRANS_DIR_HOST2DEV)){
    printf("cnrtMemcpy FAILED!\n");
    exit(-1);
}
if(CNRT_RET_SUCCESS != cnrtMemcpy(mlu_input2,half_array2,
                                  N*sizeof(half),
                                 CNRT_MEM_TRANS_DIR_HOST2DEV){
   printf("cnrtMemcpy FAILED!\n");
   exit(-1);
}
```

任务调度类型为BLOCK

分配输入数据空间

分配输出数据空间

主机端代码：数据准备

BANG单核向量加法

```
cnrtNotifier_t start;
cnrtNotifier_t end;
cnrtCreateNotifier(&start);
cnrtCreateNotifier(&end);
float timeTotal = 0.0;

printf("---1--\n");
cnrtKernelParamBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferAddParam(params, &mlu_input1, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &mlu_input2, sizeof(half*));
cnrtKernelParamsBufferAddParam(params, &mlu_ptrc, sizeof(half*));
printf("---2--\n");
cnrtPlaceNotifier(start,pQueue);
if(CNRT_RET_SUCCESS !=
  cnrtInvokeKernel((void *)&kernel, dim, params, cc, pQueue)){
  printf("cnrtInvokeKernel FAILED!\n");
  eixt(-1);
}
cnrtPlaceNotifier(end,pQueue);
cnrtNotifierDuration(start, end, &timeTotal);
printf("Hardware Total Time: %.3f ms\n", timeTotal/1000.0);
cnrtDestroyNotifier(&start);
cnrtDestroyNotifier(&end);
printf("---3--\n");
if(CNRT_RET_SUCCESS != cnrtSyncQueue(pQueue)){
  printf("cnrtSyncQueue FAILED!\n");
  exit(-1);
}
printf("---4--\n");
half *result_half=(half*)malloc(N*sizeof(half));
if(CNRT_RET_SUCCESS != cnrtMemcpy(result_half, mlu_ptrc,
N*sizeof(half),
   CNRT_MEM_TRANS_DIR_DEV2HOST)){
   printf("cnrtMemcpy FAILED!\n");
   exit(-1);
}
```

准备Notifier掐时间

添加kernel参数，并调用Kernel

把数据拷回主机端

half2float

```
float *result_array = (float*) malloc(N*sizeof(float));
for(int i=0; i<N; i++){
   cnrtConvertHalftofloat(result_array+i, result_half[i]);
#if 1
   if(isEqual(result_array[i], (float)(res[i])))
   {}//printf("PASSED\n");
   else
   printf("FAILED\n");
#endif
}
if(CNRT_RET_SUCCESS!= cnrtFree(mlu_ptrc)){
   printf("cnrtFree FAILED!\n");
   exit(-1);
}
if(CNRT_RET_SUCCESS !=
cnrtDestroyQueue(pQueue)){
   printf("cnrtDestroyQueue FAILED!\n");
   eixt(-1);
}
if(CNRT_RET_SUCCESS !=
cnrtDestoryKernelParamsBuffer(params)){
  printf("cnrtDestoryKernelParamsBuffer failed!\n");
  return -1;
}
cnrtDestroy();
free(float_array1);
free(float_array2);
free(half_array1);
free(half_array2);

return 0;
}
```

回收相关资源

主机端代码：Kernel调用

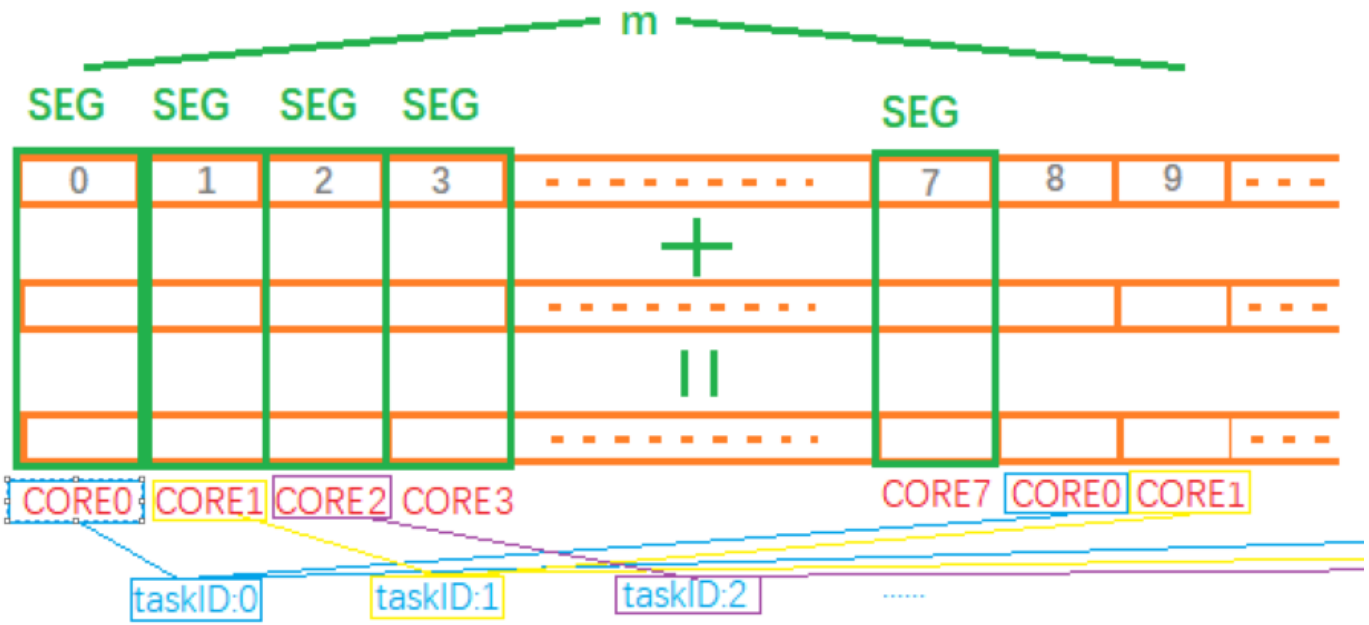
BANG多核向量加法

设备端BANG代码

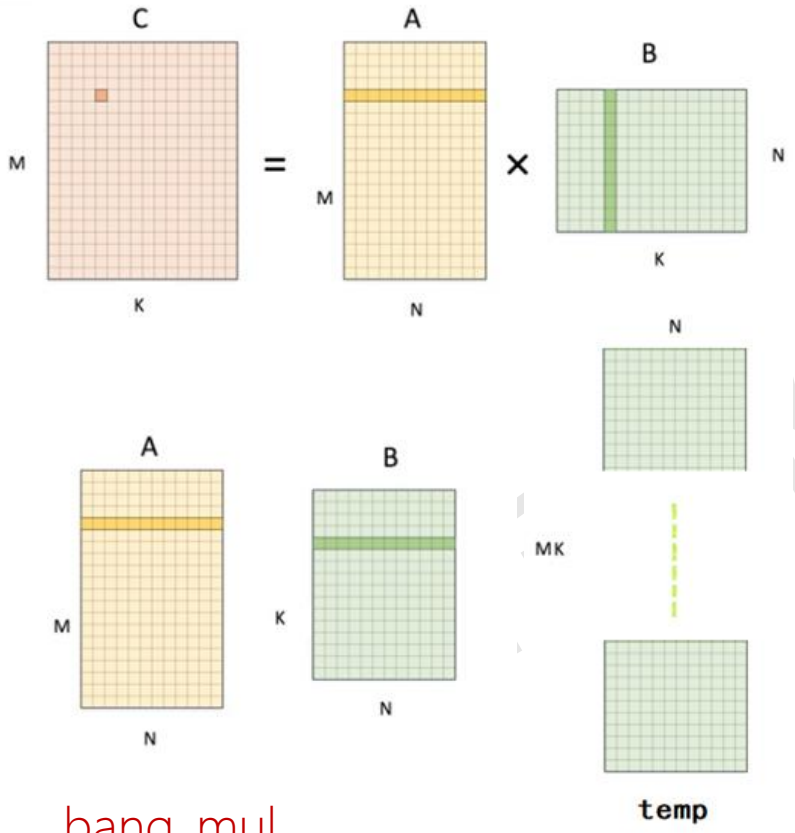
```
for(int i = taskId; i<m; i+=taskDim){
__memcpy(a_cram, a + i * SEG, SEG*sizeof(half), GDRAM2NRAM);
__memcpy(b_cram, b + i * SEG, SEG*sizeof(half), GDRAM2NRAM);
__bang_add(out_cram, a_cram, b_cram, SEG);
__memcpy(pc + i * SEG, out_cram, SEG*sizeof(half),
NRAM2GDRAM);
}
```

主机端任务规模和调度单位

```
#define CORE_NUM 8
cnrtDim3_t dim;
dim.x = CORE_NUM;
dim.y = 1;
dim.z = 1;
...
cnrtFunctionType_t cc =
CNRT_FUNC_TYPE_UNION2;
```



BANG矩阵乘法

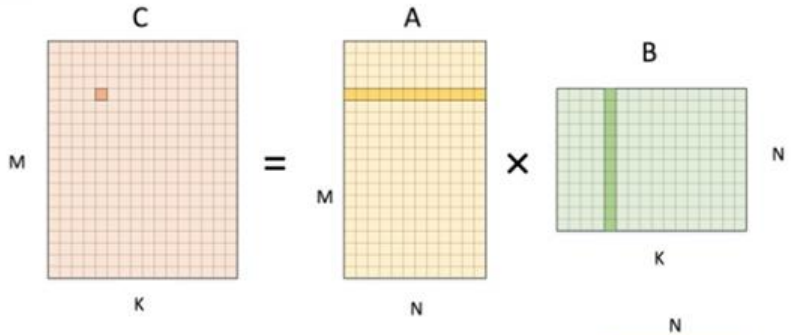


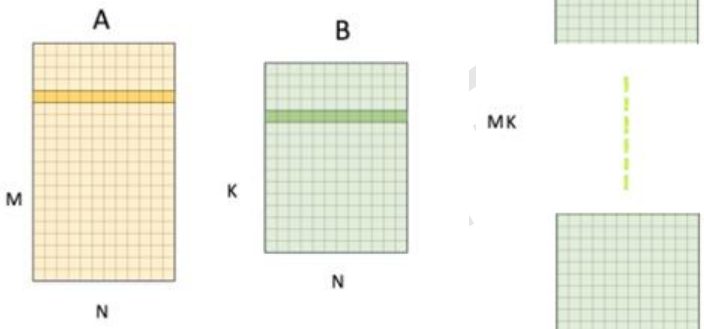
bang_mul
bang_transpose
bang_add

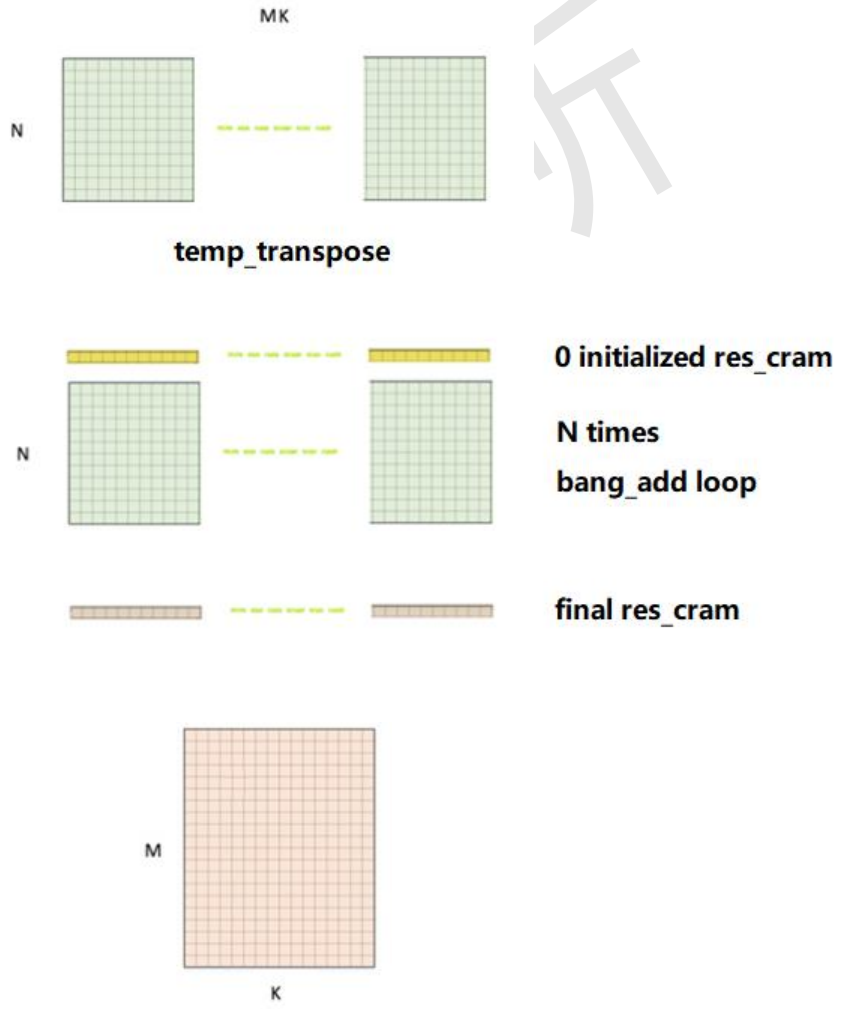
```
#include "mlu.h"
#define M 32
#define N 64
#define K 32
__mlu_entry__ void MultiplyKernel(half* mat1, half* mat2, half* res){
        __nram__ half mat1_cram[M*N];
        __nram__ half mat2_cram[N*K];
        __nram__ half mat2_transpose_cram[N*K];
        __nram__ half temp[M*K*N];
        __nram__ half temp_transpose[M*K*N];
        __nram__ half res_cram[M*K];
        __bang_printf(">>>>>>>%x \n", (half*)temp);
        __nramset_half(res_cram, M*K, 0);
        __memcpy(mat1_cram, mat1, M*N*sizeof(half), GDRAM2NRAM);
        __memcpy(mat2_cram, mat2, N*K*sizeof(half), GDRAM2NRAM);
        __bang_printf('2!\n');
        //transpose mat2 from n*k to k*n
        __bang_transpose(mat2_transpose_cram, mat2_cram, N, K);
        //multiply
        __bang_printf("%d\n", M);
        for(int i=0; i<M; i++){
            for(int j=0; j<k; j++){
__bang_mul(&temp[(i*K+j)*N], &mat1_cram[i*N],
&mat2_transpose_cram[j*N],N);
            }
}
        __bang_print("%d\n", M*K);
        __bang_transpose(temp_transpose,temp, M*K, N);
        for(int i=0; i<N; i++){
            __bang_add(res_cram, res_cram, &temp_transpose[i*M*K],
M*K);
        }
        __bang_print("kernel done!\n");
        // prepare output
        __memcpy(res, res_cram, M*K*sizeof(half), NRAM2GDRAM);
}
```

对位乘法

BANG矩阵乘法







bang_mul
bang_transpose
bang_add

第八章 智能编程语言

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

智能应用功能调试

- 功能调试接口
- 功能调试工具
- 功能调试实践

功能调试接口

- __assert(bool flag)：abort the kernel if flag is false
- __abort(): exit the kernel with error code -1
- exit(int status): exit the kernel with code status
- __bang_printf("<格式化字符串>",<参数变量>): 将字符串打印到屏幕
- __breakdump_scalar/vector()：将标量/向量值dump到指定文件

BANG调试：标量调试

- BANG用户可通过_breakdump_scalar函数打印标量值。其语义是将一组用户指定的标量变量打印到文件dumpscalar_data中，并终止在当前核上程序运行。

其主要的使用格式如下所示:

✓ __breakdump_scalar(var1) // dump一个标量的值

✓ __breakdump_scalar(var1, var2) // 同时dump 两个标量的值

✓ __breakdump_scalar(var1, var2, var3) // 同时dump 三个标量的值

✓ … …

注意:

✓每次可以同时dump最多6个标量的值

✓ _breakdump_scalar语句后，MLU 中当前核会终止程序的执行

✓需要设置环境变量 DUMPMLUSCALAR 才能将数据dump到.dumpscalar_data中 (export DUMPMLUSCALAR=1)

BANG调试：向量调试

- 向量调试接口

BANG 用户可以使用__breakdump_vector 函数打印一个向量值。其语义是将一组用户指定的向量的值打印到文件 .dumpvector_data 中，并终止当前核上程序的执行。

```
__breakdump_vector(void *src, int32_t Bytes,
                   mluBreakDumpAddrSpace_t AddrSpace)
```

该接口的格式如下所示：

参数AddrSpace表示需要dump的地址数据所在的地址空间，目前 BANG 语言支持 nram/ldram 空间上向量数据的 dump 功能，因此 AddrSpace 的取值为 NRAM/LDRAM。

注意：

✓目前支持一次最多 dump 1024 个字节，即 Bytes <= 1024;

✓只有在程序执行前设置了环境变量 DUMPMLUVECTOR，dump 的结果才会输出到 .dumpvector_data 文件中；

MLU端kernel函数，利用__breakdump_vector函数dump出数据

```
__mlu_entry__ void kernel(half* out_data, half *in1, half *in2) {
                __nram__ half in1_nram[64];
                __nram__ half in2_nram[64];
                __nram__ half out_data_nram[64];

                __memcpy(in1_nram, in1, 64 * sizeof(half), GDRAM2NRAM);
                __memcpy(in2_nram, in1, 64 * sizeof(half), GDRAM2NRAM);
                __bang_add(out_data_nram, in1_nram, in2_nram, 64);
                __memcpy(out_data, out_data_nram, 64 * sizeof(half),
NRAM2GDRAM);
                __breakdump_vector(out_data_nram, 64, NRAM);

}
```

在终端中设置环境变量

找到.dumpscalar_data文件

查看文件

```
$ export DUMPMLUVECTOR=1
$ ./a.out
$ ls -a
.dumpvector_data

$ vim .dumpvector_data

MLU taskID 0: Dumping 64 bytes vector data...
00 00 00 40 00 44 00 46
00 48 00 49 00 4a 00 4b
00 4c 80 4c 00 4d 80 4d
00 4e 80 4e 00 4f 80 4f
00 50 40 50 80 50 c0 50
00 51 40 51 80 51 c0 51
00 52 40 52 80 52 c0 52
00 53 40 53 80 53 c0 53
```

BANG调试：格式化输出

- BANG 用户可以使用__bang_printf 函数打印零个或多个标量的值。其语义是将参数按照格式化字符串打印参数到屏幕（标准输出）。
- **该接口的格式如下所示**:
  - __bang_printf (const char* fmt)
  - __bang_printf (const char* fmt, type arg1)
  - __bang_printf (const char* fmt, type arg1, type arg2)
  - __bang_printf (const char* fmt, type arg1, type arg2, type arg3)
  - __bang_printf (const char* fmt, type arg1, type arg2, type arg3, ..., type arg16)
- 并不终止程序的运行

功能调试工具

- 面向智能编程语言BCL的调试器

- 整体流程：调试前准备、调试器托管、状态查看及错误分析

- 调试前准备
  - 配置调试目标设备号，配置设备号为1的命令：

```
export DLP_VISIBLE_DEVICES = 1
```

  - 增加调试信息，编译时（编译器为bclc）使用-g选项

```
bclc -g foo.dlp -o foo
bclc -g1 foo.dlp -o foo
bclc -g2 foo.dlp -o foo
bclc -g3 foo.dlp -o foo
```

调试器托管

- 通过调试器执行被调试程序来实现托管。由于是异构程序，必须从主机端程序启动，如果要在设备端Kernel程序的入口处停住，可以使用：

```
(bcl-gdb) bcl-gdb breakpoint on # 使能设备端的断点功能
(bcl-gdb) break *0x1 # 在Kernel函数的第一条指令上设置断点
(bcl-gdb) run # 执行程序
```

- 针对多核架构DLP考虑在不同核间切换，可以用info命令查看当前调试焦点并使用focus命令进行切换

```
(bcl-gdb) bcl-gdb info
device cluster core pc core state          focus
   0       0       0  1  KERNEL_BREAKPOINT   *
   0       0       1  0  KERNEL_BREAKPOINT
   0       0       2  0  KERNEL_BREAKPOINT
... ...
   0       3       2  0  KERNEL_BREAKPOINT
```

显示当前断点情况

切换到 cluster 2 core 1

```
(bcl-gdb) bcl-gdb focus cluster 2 core 1
[Switch from logical device 0 cluster 0 core 0 to logical device 0 cluster 2
    core 1.]
device cluster core pc core state          focus
... ...
   0       2       0  0  KERNEL_BREAKPOINT
   0       2       1  0  KERNEL_BREAKPOINT *
   0       2       2  0  KERNEL_BREAKPOINT
... ...
```

调试器托管

采用 break 命令可以根据函数名、代码行号、指令地址以及 Kernel 入口来增加断点。 在 break 命令中可以**使用 if 语句配置条件断点**。断点的查看和删除则可以分别使用 info 和 delete 命令来完成

```
(bcl-gdb) break my_function
(bcl-gdb) break my_class::my_method
(bcl-gdb) break int my_templatized_function<int>(int)
(bcl-gdb) break foo.dlp:185
(bcl-gdb) bcl-gdb breakpoint on
(bcl-gdb) break *0x1

(bcl-gdb) break foo.dlp:23 if taskIdx == 1 && i < 5

(bcl-gdb) info break
(bcl-gdb) i b

(bcl-gdb) delete break 1
(bcl-gdb) d b 2 3 4
```

条件断点

断点删除

- 状态查看
  - 查看变量内容：调试时可以直接根据变量名采用 print 命令来打印相关内容

```
(bcl-gdb) print a
        $1 = 233
(bcl-gdb) print array
        $2 = (1, 2, 3, 4)
```

  - 查看寄存器内容：寄存器内容则可以采用 info registers 命令进行查看
  - 查看地址内容：指定地址中的数据内容可以通过 examine （或x）命令进行查看

```
(bcl-gdb) x/2wd 0x2333
        0x2333 : 666 888
```

功能调试实践

- 以智能编程语言BCL 和相应调试器BCL-GDB 为例介绍串行及并行程序的调试示例

- BCL实现的快速排序程序

核心代码

```
#define DATA_SIZE 64
__dlp_func__ int32_t QuickSort(int left,
                               __nram__ int32_t *m,
                               int32_t right) {
  int32_t tag = m[left];
  int32_t temp;
  for (;;) {
    if (left < right) {
      while (m[right] > tag)
        right--;
      if (left >= right)
        break;
... ...
  return right;
}

__dlp_device__ void SplitMiddle(int32_t left,
                                __nram__ int32_t *m,
                                int32_t right) {
  int middle;
  if (left < right) {
    middle = QuickSort(left, m, right);
    SplitMiddle(left, m, middle - 1);
    SplitMiddle(middle + 1, m, right);
  }
}
```

入口函数

```
__dlp_entry__ void kernel(int32_t *pData,
                          int32_t num,
                          int32_t left) {
  __nram__ int32_t nBuff[DATA_SIZE];
  __memcpy(nBuff, pData, num * sizeof(int32_t), GDRAM2NRAM);
  SplitMiddle(left, nBuff, num - 1);
  __memcpy(pData, nBuff, num * sizeof(int32_t), NRAM2GDRAM);
}
```

- 串行调试程序
  - 使用设备端编译器编译出带调试信息的设备端二进制kernel.o，并用gcc编译链接出主机端的可执行程序

指定在函数名SplitMiddle处插入断点
显示在kernel.dlp中的48行

```
(bcl-gdb) b kernel.dlp:SplitMiddle
Breakpoint 1 at 0x554: file kernel.dlp, line 48.
(bcl-gdb) r
Starting program: /xxx/demo/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, SplitMiddle (left=0, m=0x200440, right=63) at kernel.dlp:48
48 if (left < right) {
(bcl-gdb)
```

运行到函数入口处暂停，
并打印出相应代码

串行调试程序

采用 backtrace（bt）命令查看当前函数的调用栈：

```
(bcl-gdb) bt
#0 SplitMiddle (left=0, m=0x200440, right=63) at kernel.dlp:48
#1 0x00000000000002c4 in kernel (pData=0xfffff9c000, num=64, left=0) at
kernel.dlp:61
(bcl-gdb)
```

SplitMiddle在kernel.dlp中的61行被调用

使用 layout src 命令查看源码和当前断点：

```
(bcl-gdb) layout src
+--kernel.dlp------------------------------+
|29 }
|
|30 if (left >= right) {
... ...
|
|69
+-------------------------------------------+
multi-thre Thread 0x7ffff7fcf8 In: SplitMiddle L48 PC:
0x554
```

SplitMiddle的源码

串行调试程序

使用 display 命令和单步执行观察变量状态：

单步执行结果

```
(bcl-gdb) display *m
1: *m = 869
(bcl-gdb) n
1: *m = 869
(bcl-gdb) n

Breakpoint 1, SplitMiddle (left=0, m=0x200440, right=54) at
kernel.dlp:48
1: *m = 128
(bcl-gdb) n
1: *m = 128
(bcl-gdb) n

Breakpoint 1, SplitMiddle (left=0, m=0x200440, right=6) at
kernel.dlp:48
1: *m = 112
(bcl-gdb)
```

串行调试程序

使用 up 命令返回上一级调用栈，然后打印 NRAM 地址空间变量 nBuff 中数据

```
(bcl-gdb) up
#1 0x000000000000069a in SplitMiddle (left=0, m=0x200440, right=54) at
kernel.dlp:50
50 SplitMiddle(left, m, middle - 1);
(bcl-gdb) up
#2 0x000000000000069a in SplitMiddle (left=0, m=0x200440, right=63) at
kernel.dlp:50
50 SplitMiddle(left, m, middle - 1);
(bcl-gdb) up
#3 0x00000000000002c4 in kernel (pData=0xfffff9c000, num=64, left=0) at
kernel.dlp:61
61 SplitMiddle(left, nBuff, num - 1);
(bcl-gdb)
(bcl-gdb) x /64w nBuff
0x600000000000440: 112 80 125 39
0x600000000000450: 91 43 23 128
... ...
0x600000000000520: 973 929 912 978
0x600000000000530: 918 963 986 924
(bcl-gdb)
```

从SplitMiddle的递归调用中返回，直到kernel.dlp中的第一次调用

并行程序调试

- 以kernel.dlp为例，可以通过4个核并行执行该程序，主机端调用时设置UNION1任务执行

通过taskId指定更新：
local[0][0]
local[1][0]
local[2][0]
local[3][0]

```
// 并行程序示例

... ...

__nram__ int local[4][8];
__dlp_device__ int go_deeper(int i) {
  if (i == 0) {
  return 1;
  } else {
    return 1 + go_deeper(i - 1);
  }
}

__dlp_entry__ void kernel(int* input, int len) {
  int line_size = sizeof(int[8]);
  __memcpy(local, input, len * line_size, GDRAM2NRAM);
  local[taskId][0] = go_deeper(taskId + 1);
  __sync_all();
  __memcpy(input + taskId * 8, local + taskId * 8, line_size,
NRAM2GDRAM);
```

- 并行程序调试
  - 通过break、continue、info、focus、list等命令观察多核计算的执行及调试核心的切换

调试核心切换到：cluster 0，core 3

显示代码片段

打印断点信息

```
(bcl-gdb) c
Continuing.
[Switch from logical device 0 cluster 0 core 1 to logical device 0 cluster 0
    core 2.]
Breakpoint 1, ?? () at kernel.dlp:17
17 local[taskId][0] = go_deeper(taskId + 1);
1: local[taskId][0] = 843
```

Continue继续执行

```
(bcl-gdb) bcl-gdb info
device cluster core pc   core state          focus
   0       0      0  842 KERNEL_SYNC
   0       0      1  842 KERNEL_SYNC
   0       0      2  814 KERNEL_BREAKPOINT *
   0       0      3  0   KERNEL_BREAKPOINT
   0       0      4  0   KERNEL_BREAKPOINT
```

打印调试状态

```
(bcl-gdb) bcl-gdb focus cluster 0 core 3
[Switch from logical device 0 cluster 0 core 2 to logical device 0 cluster 0
    core 3.]

(bcl-gdb) bcl-gdb info
device cluster core pc   core state          focus
   0       0      0  842 KERNEL_SYNC
   0       0      1  842 KERNEL_SYNC
   0       0      2  814 KERNEL_BREAKPOINT
   0       0      3  0   KERNEL_BREAKPOINT *
   0       0      4  0   KERNEL_BREAKPOINT
```

```
(bcl-gdb) list
12
13 __dlp_entry__ void kernel(int* input, int len) {
14 int line_size = sizeof(int[8]);
15
16 __memcpy(local, input, len * line_size, GDRAM2NRAM);
17 local[taskId][0] = go_deeper(taskId + 1);
18 __sync_all();
19 __memcpy(input + taskId * 8, local + taskId * 8, line_size,
NRAM2GDRAM);
20 }
```

```
(bcl-gdb) i b
Num Type Disp Enb Address What
1 breakpoint keep y 0x000000000000032e kernel.dlp:17
breakpoint already hit 3 times
```

第八章 智能编程语言

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

智能应用性能调优

- 性能调优方法
- 性能调优接口
- 性能调优工具
- 性能调优实践

性能调优方法

- 核心是如何充分利用大规模的并行计算单元
- 使用片上存储

优化前（完成长度为16384的向量对位乘法）：

```
#define LEN 16384;
__dlp_entry__ void vector_mult(float* in1, float* in2, float* out) {
  for (int i = 0; i < LEN; i++) {
    out[i] = in1[i] * in2[i];
  }
}
```

原始的两个输入
和输出都在GDRAM上

使用NRAM 优化后：

```
#define LEN 16384;
__dlp_entry__ void vector_mult(float* in1, float* in2, float* out) {
  __nram__ float tmp1[LEN];
  __nram__ float tmp2[LEN];
  __memcpy(tmp1, in1, LEN * sizeof(float), GDRAM2NRAM);
  __memcpy(tmp2, in2, LEN * sizeof(float), GDRAM2NRAM);
  for (int i = 0; i < LEN; i++) {
    tmp2[i] = tmp1[i] * tmp2[i];
  }
  __memcpy(out, tmp2, LEN * sizeof(float), NRAM2GDRAM);
}
```

GDRAM的延迟太高，
计算单元大部分时间在等数
使用片上的低延迟NRAM

张量计算

张量化的基本原理是将大量标量计算合并为张量计算，使用智能编程语言的张量计算语句改写代码，充分利用硬件的张量计算单元，提升程序运行速度。

```
#define LEN 16384;
__dlp_entry__ void vector_mult(float* in1, float* in2, float* out) {
  __nram__ float tmp1[LEN];
  __nram__ float tmp2[LEN];
  __memcpy(tmp1, in1, LEN * sizeof(float), GDRAM2NRAM);
  __memcpy(tmp2, in2, LEN * sizeof(float), GDRAM2NRAM);
  __vec_mul(tmp2, tmp1, tmp2, LEN);
  __memcpy(out, tmp2, LEN * sizeof(float), NRAM2GDRAM);
}
```

把标量的for循环替换成一条张量计算语句，
使用张量计算单元来加速

多核并行

针对（程序员可见的）多核，可以将一个任务分拆到多个核上并行计算，进一步提升程序性能。

```
#define LEN 16384;
#define CORE_NUM 4;
#define PER_CORE_LEN (LEN / CORE_NUM);
__dlp_entry__ void vector_mult(float* in1, float* in2, float* out) {
  __nram__ float tmp1[PER_CORE_LEN];
  __nram__ float tmp2[PER_CORE_LEN];
  __memcpy(tmp1, in1 + taskId * PER_CORE_LEN, PER_CORE_LEN * sizeof(float),
    GDRAM2NRAM);
  __memcpy(tmp2, in2 + taskId * PER_CORE_LEN, PER_CORE_LEN * sizeof(float),
    GDRAM2NRAM);
  __vec_mul(tmp2, tmp1, tmp2, PER_CORE_LEN);
  __memcpy(out + taskId * PER_CORE_LEN, tmp2, PER_CORE_LEN * sizeof(float),
    NRAM2GDRAM);
}
```

把向量乘法拆分到4个核上，每个核计算4096大小的向量乘法，用taskId进行索引

性能调优接口

- 在程序中使用性能调优接口，帮助识别程序执行过程的瓶颈
- 识别瓶颈的过程
  - 先找到耗时长的部分 ➡ 通知（Notifier）接口
  - 再通过硬件计数器（Performance Counter）分析硬件执行特征 ➡ 硬件计数器接口

```
Notifier_t start;
CreateNotifier(&start);
Notifier_t end;
CreateNotifier(&end);

Queue_t queue;
CreateQueue(&queue);
PlaceNotifier(start, queue);
InvokeKernel((void*)(&vector_mult), dim, params, UNION1, queue);
PlaceNotifier(end, queue);
SyncQueue(queue);

float time;
NotifierDuration(start, end, &time);
```

使用Notifier接口
获取vector_mult的执行时间

硬件计数器接口

通过提供硬件计数器值的获取接口，方便开发者对程序的行为进行细粒度剖析和优化

接口示例	具体功能
__perf_start	使能硬件性能计数器开始计数
__perf_stop	使能硬件性能计数器停止计数
__perf_get_clock	获取当前硬件时间戳
__perf_get_executed_inst	获取已执行的指令条数
__perf_get_cache_miss	获取指令缓存不命中的次数
__perf_get_compute_alu	获取标量运算单元的运算量
__perf_get_compute_nfu	获取基于 NRAM 的张量运算部件的运算量
__perf_get_compute_wfu	获取基于 WRAM 的张量运算部件的运算量
__perf_get_memory_dram_read	获取从 DRAM 读取的数据量
__perf_get_memory_dram_write	获取向 DRAM 写入的数据量
__perf_get_memory_sram_read	获取从片上共享 SRAM 读取的数据量
__perf_get_memory_sram_write	获取向片上共享 SRAM 写入的数据量

- 硬件计数器接口

关键代码段前后
插入硬件计数器接口

获取的信息包括：
1、NRAM对应的
运算部件计算量
2、DRAM读数据量
3、DRAM写数据量

```
#define LEN 64;
__dlp_entry__ void vector_mult(float* in1, float* in2, float* out) {
  __nram__ float tmp1[LEN];
  __nram__ float tmp2[LEN];

  __perf_start();

  __memcpy(tmp1, in1, LEN * sizeof(float), GDRAM2NRAM);
  __memcpy(tmp2, in2, LEN * sizeof(float), GDRAM2NRAM);
  __vec_mul(tmp2, tmp1, tmp2, LEN);
  __memcpy(out, tmp2, LEN * sizeof(float), NRAM2GDRAM);

  __perf_stop();

  int nComputeNram = __perf_get_compute_nfu();
  int nReadDram = __perf_get_memory_dram_read();
  int nWriteDram = __perf_get_memory_dram_write();

  printf("nComputeNram = %d Byte, nReadDram = %d Byte, nWriteDram = %d Byte\
    n", nComputeNram, nReadDram, nWriteDram);
}
```

```
nComputeNram = 256 Byte, nReadDram = 512 Byte, nWriteDram = 256 Byte
```

性能调优工具

- 在**程序外部**监控程序的运行状态，分析其执行瓶颈，找到优化空间。可以分为两类：
  - 应用级性能剖析工具
  - 系统级性能监控工具

- 应用级性能剖析工具

  具体使用流程分为两个阶段：

  1）采用 record 命令来运行可执行程序并生成相应的性能分析报告；

  2）采用 report 或者 replay 命令查看性能分析报告，获取包括执行时间、调用关系以及性能计数器等信息。

- 应用级性能剖析工具
  - 运行record vecMult ./info_dir
  - 运行report info_dir

IO效率

计算效率

执行时间

```
report info_dir/  # 终端显示如下信息
# PID    Total time   Calls   Function
# ===   ===========   =====   ======================
# 2510   918.00 us    1       SetCurrentDevice
#        698.00 us    3       devMalloc
#         66.00 us    1       InvokeKernel
#
# ----   --------    ----   ----------------------------
# Kernels Info:
# PID    Duration    ComputeSpeed    IOSpeed      IOCount   Function
# ===    ========    ============    =======      =======   ==============
# 2510   15.00 us    7721 GOPs       4.973 GiB/s  196608    vector_mult
# ----   --------    ------------    ----------   ------    ----------------
# DEVICE_TO_HOST size: 64KB   speed: 0.3 GB/s
# HOST_TO_DEVICE size: 192KB  speed: 0.5 GB/s
```

- 应用级剖析工具
  - 运行replay info_dir
  - 以时间线的方式来查看函数的执行时间：

```
replay info_dir/  # 终端显示如下信息
#   PID        TIMESTAMP          TIME              Function
# =====    =============    ============    ============
# ......
#   2510    [386ms,410us]                      InvokeKernel {
#   2510    [386ms,410us]                        vector_mult {
#   2510    [386ms,425us]    [ 15.000 us]      }  /*vector_mult*/
#   2510    [386ms,476us]    [ 66.000 us]    }  /*InvokeKernel*/
# ......
```

系统级性能监控工具

系统级性能监控工具主要利用驱动通过读取寄存器的方式来收集硬件的静态和动态信息

命令示例	具体功能
monitor -info	显示以下所有信息
monitor -type	显示板卡型号
monitor -driver	显示驱动版本
monitor -fan	显示风扇转速比
monitor -power	显示运行功耗
monitor -temp	显示芯片温度
monitor -memory	显示物理内存使用情况
monitor -bandwidth	显示计算核心对设备内存的最大访问带宽
monitor -core	显示各计算核心的利用率

性能调优实践

- 将前述系列方法应用到DFT（离散傅里叶变换）的性能调优

DFT 是将信号从时域变换到频域的算法，且时域和频域都是离散的。DFT 可以求出一个信号由哪些正弦波叠加而成，所得结果是这些正弦波的幅度和相位。其变换公式为：

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn} \tag{1}$$

其中 X(k) 是变换后的频域序列，x(n) 是变换前的时域序列（采样信号，均为实数，无虚部）。如果将上式的 $e^{-j\frac{2\pi}{N}kn}$ 部分，用欧拉公式 $e^{jx} = \cos(x) + j\sin(x)$ 替换，得到如下公式：

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)[\cos(-\frac{2\pi}{N}kn) + j\sin(-\frac{2\pi}{N}kn)] \\ &= \sum_{n=0}^{N-1} [x(n)\cos(\frac{2\pi}{N}kn) - jx(n)\sin(\frac{2\pi}{N}kn)] \end{aligned} \tag{2}$$

- DFT

其中 X(k) 的实部和虚部分别是：

$$real(k) = \sum_{n=0}^{N-1} x(n)\cos(\frac{2\pi}{N}kn)$$
$$imag(k) = \sum_{n=0}^{N-1} -x(n)\sin(\frac{2\pi}{N}kn) \quad (3)$$

由此得到 X(k) 的幅值是：

$$Amp(k) = \sqrt{real(k)^2 + imag(k)^2}$$

- 首先用标量实现的上述DFT 基础算法

```
#define PI 3.14159265
#define N 128
__dlp_entry__ void DFT (float* x, float* Amp) {
  for (int k = 0; k < N; k++) {
    float real = 0.0;
    float imag = 0.0;
    for (int n = 0; n < N; n++) {
      real += x[n] * cos(2 * PI / N * k * n);
      imag += -x[n] * sin(2 * PI / N * k * n);
    }
    Amp[k] = sqrt(real * real + imag * imag);
  }
}
```

- 使用片上缓存优化

```
#define PI 3.14159265
#define N 128
__dlp_entry__ void DFT (float* x, float* Amp) {
  __nram__ float in[N];
  __nram__ float out[N];
  __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);

  for (int k = 0; k < N; k++) {
    float real = 0.0;
    float imag = 0.0;
    for (int n = 0; n < N; n++) {
      real += in[n] * cos(2 * PI / N * k * n);
      imag += -in[n] * sin(2 * PI / N * k * n);
    }
    out[k] = sqrt(real * real + imag * imag);
  }
  __memcpy(Amp, out, N * sizeof(float), NRAM2GDRAM);
}
```

- 将最后的幅值计算从外层循环中拿出，采用向量优化

```
#define PI 3.14159265
#define N 128
__dlp_entry__ void DFT (float* x, float* Amp) {
  __nram__ float in[N];
  __nram__ float out[N];
  __nram__ float real[N];
  __nram__ float imag[N];
  __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);

  for (int k = 0; k < N; k++) {
    real[k] = 0.0;
    imag[k] = 0.0;
    for (int n = 0; n < N; n++) {
      real[k] += in[n] * cos(2 * PI / N * k * n);
      imag[k] += -in[n] * sin(2 * PI / N * k * n);
    }
  }

  __vec_mul(real, real, real, N);
  __vec_mul(imag, imag, imag, N);
  __vec_add(out, real, imag, N);
  __vec_sqrt(out, out, N);
  __memcpy(Amp, out, N * sizeof(float), NRAM2GDRAM);
}
```

$$Amp(k) = \sqrt{real(k)^2 + imag(k)^2}$$

- 从**算法层面**进行优化

在算法逻辑层面可以进行的优化如下：

1）原始算法中 sin 和 cos 括号内值相同，所以计算一遍即可；

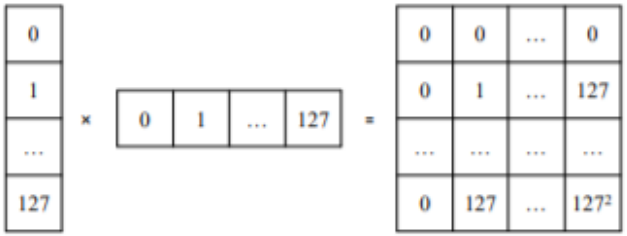
2）“2 * PI / N * k * n”这五个数中前三者是常数，不必重复计算，在循环外计算一次即可；

3）imag 因为后面要取平方，累加的时候不用取负数。

```
#define PI 3.14159265
#define N 128
__dlp_entry__ void DFT (float* x, float* Amp) {
  __nram__ float in[N];
  __nram__ float out[N];
  __nram__ float real[N];
  __nram__ float imag[N];
  __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);
  float con = 2 * PI / N;                          2
  for (int k = 0; k < N; k++) {
    real[k] = 0.0;
    imag[k] = 0.0;
    for (int n = 0; n < N; n++) {
      float tmp = con * k * n;                     1
      real[k] += in[n] * cos(tmp);
      imag[k] += in[n] * sin(tmp);                 3
    }
  }

  __vec_mul(real, real, real, N);
  __vec_mul(imag, imag, imag, N);
  __vec_add(out, real, imag, N);
  __vec_sqrt(out, out, N);
  __memcpy(Amp, out, N * sizeof(float), NRAM2GDRAM);
}
```

算法优化

此时程序主要遗留 real 和 imag 计算未经优化，其中主要计算有两个：一是内层 for 循 环中的 k*n；二是内层 for 循环中的“real[k] += in[n] * cos()”和“imag[k] += in[n] * sin()”。对于两层 for循环中的 k*n，一共会产生N^2个结果，可以用矩阵乘法来表示计算结果。



(1)

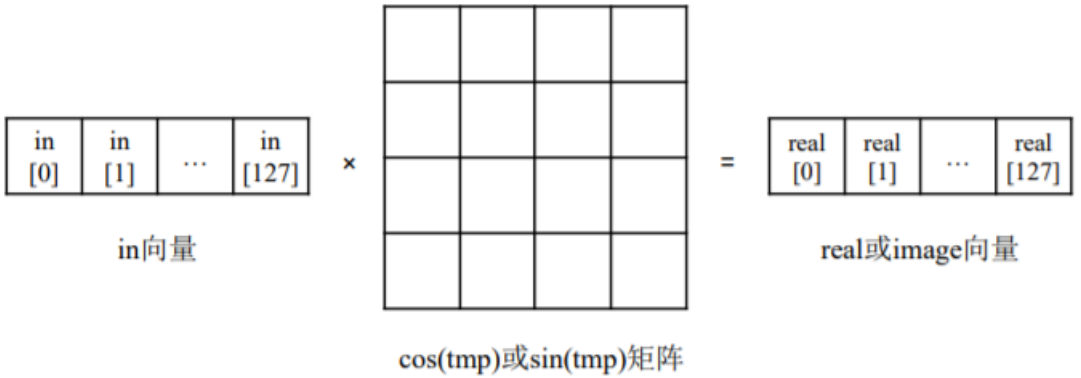
这样 k*n 的结果都保存在矩阵中，后续乘以 2*PI/N（已优化为con） 的过程可以用 __cycle_mul 来完成

(2)

算法优化

之后的变量 sin(tmp) 和 cos(tmp) 同样可以替换为向量计算（vec_sin和vec_cos）；最后剩下的内层 for 循环中的“real[k] += in[n] * cos()”和“imag[k] += in[n] * sin()”也同样可以转换为in向量和N*N矩阵的向量矩阵乘法运算。

```
for (int k = 0; k < N; k++) {
  real[k] = 0.0;
  imag[k] = 0.0;
  for (int n = 0; n < N; n++) {
    float tmp = con * k * n;
    real[k] += in[n] * cos(tmp);
    imag[k] +=  in[n] * sin(tmp);
  }
}
```



算法优化结果

用矩阵乘法优化 DFT 算法

N*1和1*N矩阵的乘法

1*N和N*N矩阵的乘法

```
#define PI 3.14159265
#define N 128
__dlp_entry__ void DFT (float* x, float* Amp) {
  __nram__ float in[N];
  __nram__ float real[N];
  __nram__ float imag[N];
  __nram__ float con[1];
  __nram__ float sequence[N];
  __nram__ float kn[N * N];
  __nram__ float cos_res[N * N];

  __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);
  con[0] = 2 * PI / N;
  for (int k = 0; k < N; k++) {
    sequence[k] = k;
  }

  __mat_mul(kn, sequence, sequence, N, 1, N);  // 构造"k*n"矩阵
  __cycle_mul(kn, kn, con, N * N, 1);          // 把"k*n"矩阵乘以常量2 * PI / N
  __vec_cos(cos_res, kn, N * N);               // 计算cos(kn)
  __vec_sin(kn, kn, N * N);                    // 计算sin(kn)，结果保存在kn中
  __mat_mul(real, in, cos_res, 1, N, N);       // 计算实部real向量
  __mat_mul(imag, in, kn, 1, N, N);            // 计算虚部imag向量

  __vec_mul(real, real, real, N);
  __vec_mul(imag, imag, imag, N);
  __vec_add(imag, real, imag, N);              // real和imag的求和结果仍保存在imag
  __vec_sqrt(imag, imag, N);
  __memcpy(Amp, imag, N * sizeof(float), NRAM2GDRAM);
}
```

常数预处理

对于值固定的部分（k*n矩阵、sin/cos等）可以提前算好，对于处理大量不同输入时不必重复计算。

cos_mat和sin_mat是提前计算好的

```
#define PI 3.14159265
#define N 128
__dlp_entry__ void DFT (float* x, float* cos_mat, float* sin_mat, float* Amp)
      {
  __nram__ float in[N];
  __nram__ float real[N];
  __nram__ float imag[N];
  __nram__ float cos_res[N * N];
  __nram__ float sin_res[N * N];

  __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);
  __memcpy(cos_res, cos_mat, N * N * sizeof(float), GDRAM2NRAM);
  __memcpy(sin_res, sin_mat, N * N * sizeof(float), GDRAM2NRAM);

  __mat_mul(real, in, cos_res, 1, N, N);  // 计算实部real向量
  __mat_mul(imag, in, sin_res, 1, N, N);  // 计算虚部imag向量

  __vec_mul(real, real, real, N);
  __vec_mul(imag, imag, imag, N);
  __vec_add(imag, real, imag, N);  // real和imag的求和结果仍保存在imag
  __vec_sqrt(imag, imag, N);       // 张量开方语句
  __memcpy(Amp, imag, N * sizeof(float), NRAM2GDRAM);
}
```

- 优化结果分析
  - 都是张量运算且使用NRAM，DFT的运算整体性能提升近1800 倍
  - 核心目的是充分利用硬件的计算资源
    - 充分**利用近端存储**，降低访存延迟带来的计算单元空闲
    - 充分**利用张量计算单元**
    - 充分**降低计算量**、节省存储空间（通过算法优化）
    - 充分**利用多核并行**，利用多核的计算单元

优化方法	性能提升倍数
原始标量程序	1（基准性能）
使用片上缓存	1.49
张量化	10.65
算法优化	29.27
常数预处理	1794

第八章 智能编程语言

- 为什么需要智能编程语言
- 智能计算系统抽象架构
- 智能编程模型
- 智能编程语言基础
- 智能应用编程接口
- 智能应用功能调试
- 智能应用性能调优
- 基于智能编程语言的系统开发

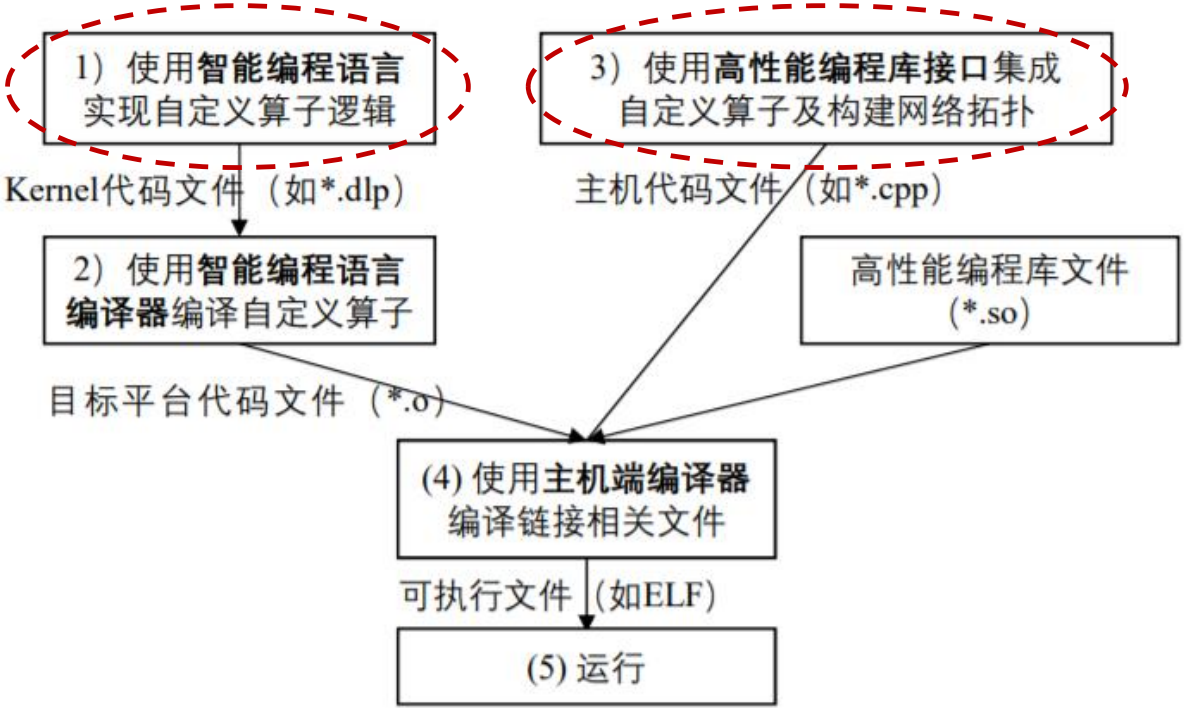
基于智能编程语言的系统开发

- 轻量级高性能库算子开发
- 实践：基于BANG的高性能库算子开发
- 编程框架算子开发
- 实践：基于BANG+CNML+TensorFlow系统开发与优化

高性能库算子开发

- 高性能库（如MKL、cuDNN和CNML等）提供了常见算子在特定平台上的高性能实现，方便用户以API的形式直接调用
- 如何用智能编程语言扩展高性能库算子

高性能库算子开发的关键在于：
1）Kernel 代码逻辑的开发与优化
2）高性能库算子接口 API 的使用



高性能库算子开发：自定义算子集成

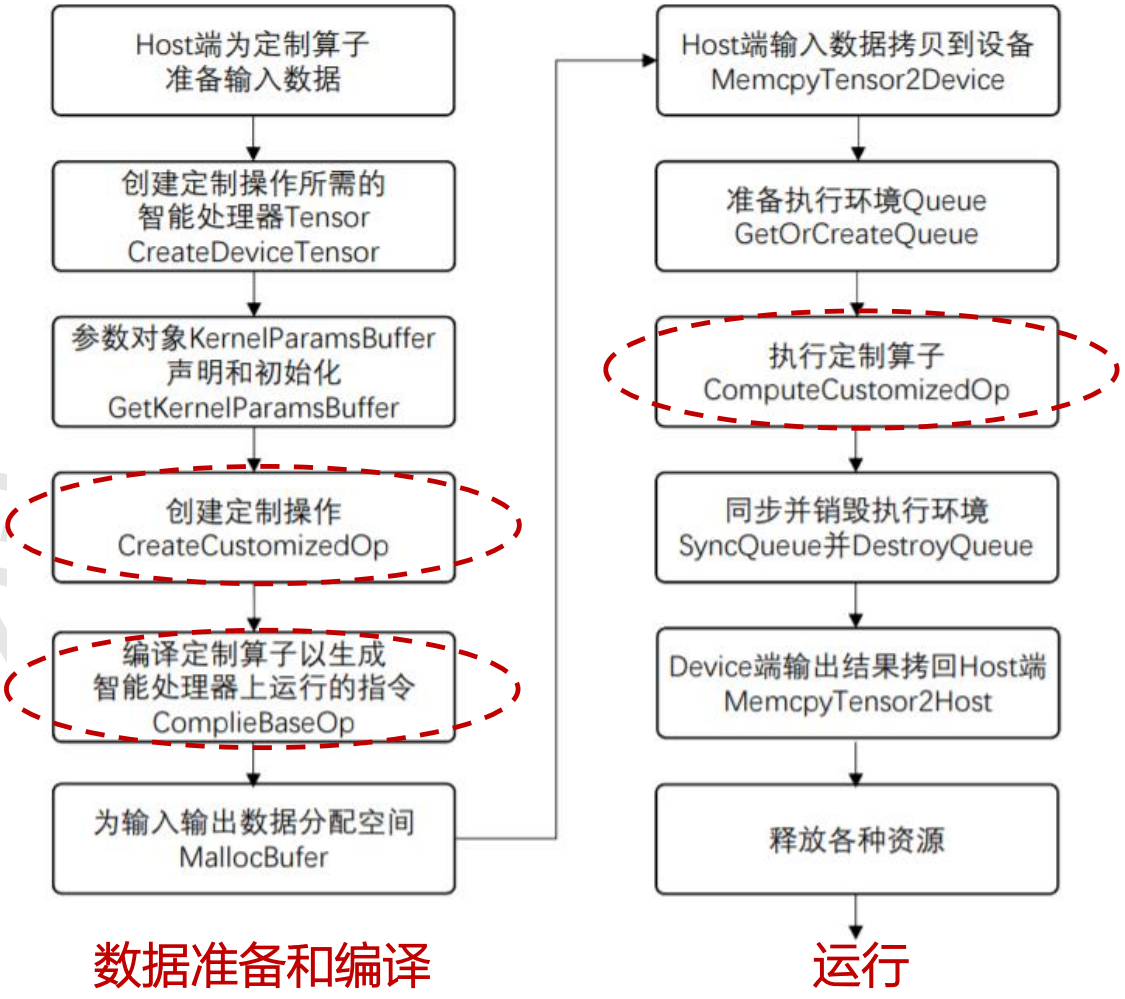
- 算子集成API介绍

典型高性能库中自定义算子集成的主要 API，主要包括自定义算子的创 建（CreateCustomizedOp）、（前向）计算（ComputeCustomizedOp）和销毁（DestroyBaseOp） 等接口

```
// 创建新的定制算子描述符
CreateCustomizedOp();

// 在指定设备上执行CustomizedOp操作
ComputeCustomizedOp();

// 销毁定制算子描述符
DestroyBaseOp();
```

高性能库算子开发：基础算子集成

- 作为自定义的单个基础算子集成进高性能库中执行



采用已有的CompileBaseOp
接口来编译相关指令

高性能库算子开发：融合算子集成

- 作为自定义算子和高性能库中已有算子进行融合计算
  - 减少中间数据片外交换，提高性能
  - 主要使用方式

往FusionOp中插入多个算子，包括库中已有算子，以及新增加的自定义算子，然后进行统一的编译优化。

高性能库算子开发示例

- 实现自定义的向量Power算子

$$y = x^c = e^{lnx^c} = e^{clnx}, (x > 0)$$

- 编写Kernel代码逻辑

```
#define N 1024
__dlp_entry__ void myPower(float* x, float* c, float* y) {
  __nram__ float in[N];
  __nram__ float cc[1];
  __memcpy(in, x, N * sizeof(float), GDRAM2NRAM);
  __memcpy(cc, c, sizeof(float), GDRAM2NRAM);

  __vec_log(in, in, N);  // 张量ln语句
  __cycle_mul(in, cc, in, N, 1);  // 长、短向量循环乘
  __vec_exp(in, in, N);  // 张量exp语句

  __memcpy(y, in, N * sizeof(float), NRAM2GDRAM);
}
```

作为基础算子集成

高性能库自定义算子的 C++ 代码

创建自定义算子
CreateCustomizedOp

编译自定义算子
CompileBaseOp

创建队列并通过
ComputeCustomizedOp
完成计算

```
int main() {
  // 创建 Tensor
  Tensor_t input;   CreateTensor(&input, FLOAT32, DIM_NCHW, 1, 1, 1, 1024);
  Tensor_t powerC;   CreateTensor(&powerC, FLOAT32, DIM_NCHW, 1, 1, 1, 1);
  Tensor_t output;   CreateTensor(&output, FLOAT32, DIM_NCHW, 1, 1, 1, 1024);

  // 创建自定义算子
  BaseOp_t powerOp;
  CreateCustomizedOp(&powerOp, "kernel", reinterpret_cast <void*>(&Power), params, {input, powerC}, 2,
{output}, 1, nullptr, 0);

  // 编译自定义算子
  CompileBaseOp(powerOp);

  // 分配设备端内存空间
  void* input_d = devMalloc(1024 * sizeof(float));
  void* c_d = devMalloc(sizeof(float));
  void* output_d = devMalloc(1024 * sizeof(float));

  // 分配主机端内存空间
  void* input_h = hostMalloc(1024 * sizeof(float));
  void* c_h = hostMalloc(sizeof(float));
  void* output_h = hostMalloc(1024 * sizeof(float));
  …… ……

  // 将输入数据从主机端内存拷贝到设备端内存
  Memcpy(input_d, input_h, 1024 * sizeof(float), HOST2DEV);
  Memcpy(c_d, c_h, sizeof(float), HOST2DEV);

  // 设备计算
  Queue_t queue;
  CreateQueue(&queue);
  ComputeCustomizedOp(powerOp, {input_d, c_d}, 2, {output_d}, 1, queue);
  SyncQueue(queue);
  DestroyQueue(queue);

  // 将输出数据从设备端内存拷贝到主机端内存
  Memcpy(output_h, output_d, 1024 * sizeof(float), DEV2HOST);

  // 释放内存空间
  …… ……
}
```

作为融合算子集成

- Conv+Power+ReLU进行融合

创建Conv、Power和ReLU算子

创建融合算子并进行编译

完成融合算子计算

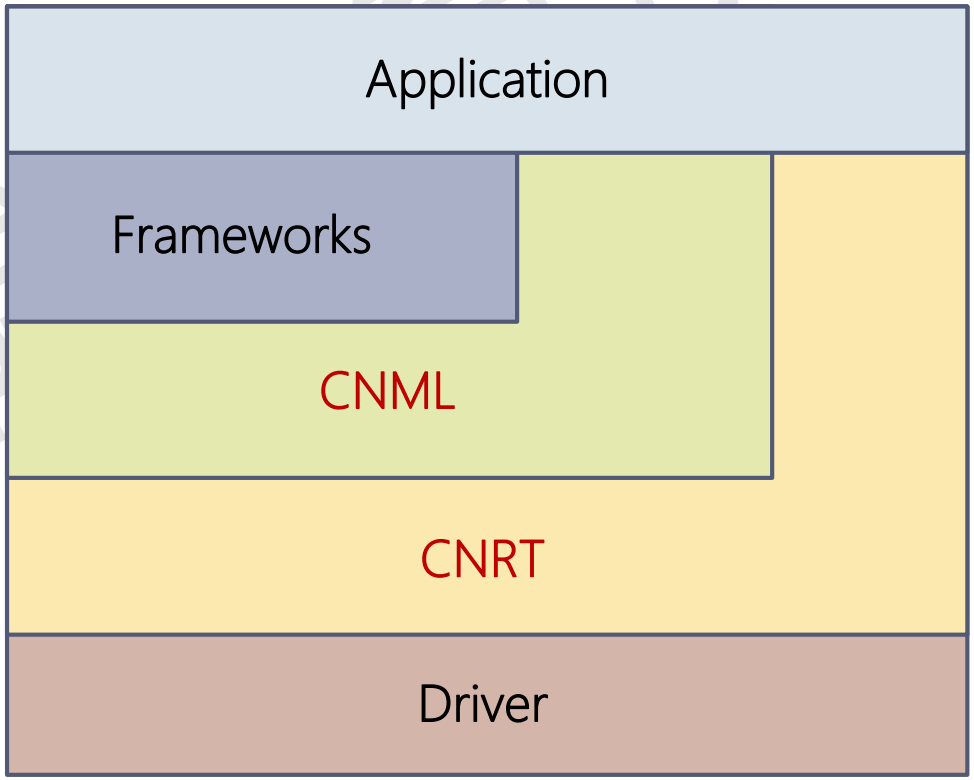
```
// 创建 Conv 算子、Power 自定义算子以及 ReLU 算子
BaseOp_t conv;
CreateConvOp(&conv, conv_param, input, convOut, filter, NULL);
BaseOp_t power;
CreateCustomizedOp(&power, "kernel", reinterpret_cast <void*>(&Power), params, {convOut, powerC}, 2,
{powerOut}, 1, NULL, 0);
BaseOp_t relu;
CreateActiveOp(&relu, ACTIVE_RELU, powerOut, reluOut);

// 创建融合算子并编译
FusionOp_t net;
CreateFusionOp(&net);
FuseOp(conv, net);
FuseOp(power, net);
FuseOp(relu, net);
SetFusionIO(net, {input, filter, powerC}, 3, {reluOut}, 1);
CompileFusionOp(net);

// 申请设备端空间 input_d, filter_d, powerC_d, reluOut_d
... ...
// 申请主机端空间 input_h, filter_h, powerC_h, reluOut_h
... ...

// 将输入数据从主机内存拷贝到设备内存
Memcpy(input_d, input_h, 128 * 128 * sizeof(float), HOST2DEV);
Memcpy(filter_d, filter_h, 2 * 2 * sizeof(float), HOST2DEV);
Memcpy(powerC_d, powerC_h, sizeof(float), HOST2DEV);

// 计算网络
Queue_t queue;
CreateQueue(&queue);
ComputeFusionOp(net, {input_d, filter_d, powerC_d}, {reluOut_d}, queue);
SyncQueue(queue);
DestroyQueue(queue);
```

实践：基于BANG的高性能库算子开发

- 以BANG和CNML为例详细介绍高性能库算子的开发实践
  - 自定义算子接口的具体实例：CNML PluginOp
  - PluginOp API介绍
  - PluginOp集成
    - 基础算子集成
    - 融合算子集成

PluginOp简介

- PluginOp接口主要在软件栈的CNML层提供，将用户用BANG实现的算子"插入"到CNML库中
- 自定义算子可以与CNML中已有算子实现（如Conv、Pooling等）执行逻辑统一，从而利用CNML的特性（如融合执行、离线指令生成等）



PluginOp API介绍

- PluginOp所涉及到的接口列表：
  - cnmlCreatePluginOp();
  - cnmlComputePluginOpForward_V4();
  - cnrtGetKernelParamsBuffer();
  - cnrtKernelParamsBufferMarkInput();
  - cnrtKernelParamsBufferMarkOutput();
  - cnrtKernelParamsBufferMarkStatic();
  - cnrtKernelParamsBufferAddParam();
  - cnrtDestoryKernelParamsBuffer();

CNML

CNRT

创建一个新的PluginOp描述符

```
cnmlStatus_t cnmlCreatePluginOp(
        cnmlBaseOp_t* op,    // 操作描述符，比如说convOp，powerOp等
        const char* name,    // 字符串描述的Plugin算子名称，区分网络中不同算子
        void* kernel,,       // Kernel中定义的函数名，用于索引.o/.so中的kernel符号
        cnrtKernelParamsBuffer_t params, // Kernel运行时所需的额外参数，输入、输出，静态数据，常量参数等
        cnmlTensor_t* input_ptrs,  // 输入数据的Tensor描述符数组
        int input_num,             // 输入数据的Tensor描述符个数
        cnmlTensor_t* output_ptrs,  // 输出数据的Tensor描述符数组
        int output_num,            // 输出数据的Tensor描述符个数
        cnmlTensor_t* static_ptrs,    // 静态数据的Tensor描述符数组
        int static_num);           // 静态数据的Tensor描述符个数
```

在指定设备上执行PluginOp操作

```
cnmlStatus_t cnmlComputePluginOpForward_V4(
        cnmlBaseOp_t op,                //基础操作描述符
        cnmlTensor_t  input_tensors[],          //指向为输入Tensor描述符所分配内存的指针数组
        void *inputs[],                 //指针数组，每个元素指向输入数据的MLU地址
        int input_num,                  //输入数据指针数组的元素个数
        cnmlTensor_t  output_tensors[],         //指向为输出Tensor描述符所分配内存的指针数组
        void *outputs[],                //指针数组，每个元素指向输出数据的MLU地址
        int output_num,                 //输出数据指针数组的元素个数
        cnrtQueue_t queue,              //执行队列
        void *extra);                   //额外参数，默认为NULL
```

- 初始化cnrtKernelParamsBuffer_t参数对象，用于cnmlCreatePluginOp中

```
cnrtRet_t cnrtGetKernelParamsBuffer(
        cnrtKernelParamsBuffer_t *params);
```

- 在cnrtKernelParamsBuffer_t中为**输入数据地址指针**占位，以便运行时填入输入数据的指针地址

```
cnrtRet_t cnrtGetKernelParamsBufferMarkInput(
        cnrtKernelParamsBuffer_t params);
```

- 在cnrtKernelParamsBuffer_t中为**输出数据地址指针**占位，以便运行时填入输出数据的指针地址

```
cnrtRet_t cnrtGetKernelParamsBufferMarkOutput(
        cnrtKernekParamsBuffer_t params);
```

- 在cnrtKernelParamsBuffer_t中为**静态数据地址指针**占位，以便运行时填入静态数据的指针地址，比如权重数据和均值数据等可通过该方式传入

```
cnrtRet_t cnrtGetKernelParamsBufferMarkStatic(
        cnrtKernelParamsBuffer_t params);
```

- 向cnrtKernelParamsBuffer_t中增加**常数参数**

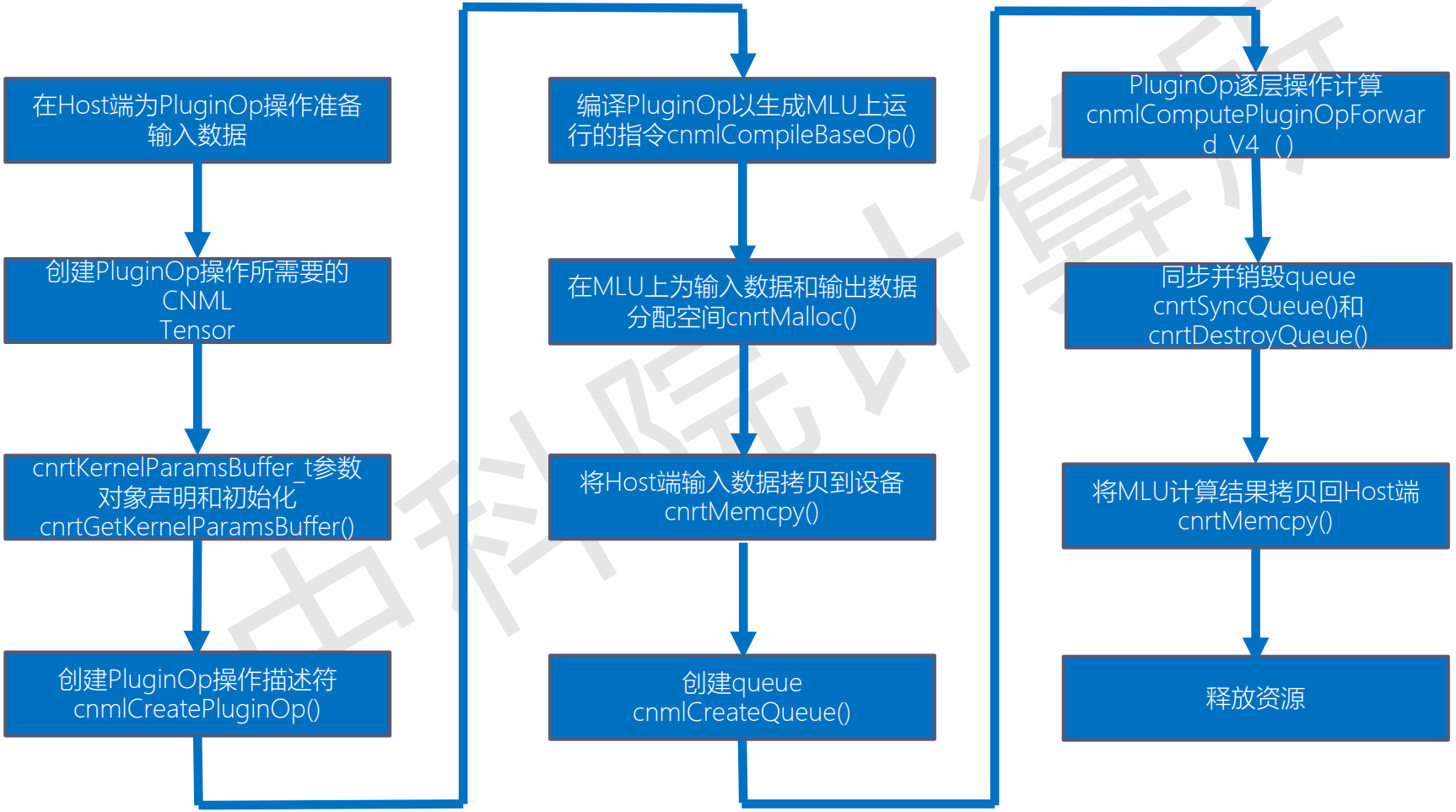
```
cnrtRet_t cnrtGetKernelParamsBufferAddParam(
                cnrtKernelParamsBuffer_t params,
                void *data,      // 参数的主机端数据指针
                size_t nBytes); // 以字节表示的参数长度
```

- 销毁cnrtKernelParamsBuffer_t变量

```
cnrtRet_t cnrtDestroyKernelParamsBuffer(
        cnrtKernelParamsBuffer_t params);
```

params: cnrtKernelParamsBuffer_t指针对象；
返回值: CNRT_RET_SUCCESS表示成功销毁cnrtKenrelParamsBuffer_t参数对象，否则，返回相应的错误码。

PluginOp算子集成：基础算子



主机侧代码

① 在Host端为PluginOp准备输入数据
输入数据维度为: 1 x 64 x 32 x 32 (NCHW),
输出数据维度为: 1 x 64 x 30 x 30(NCHW)
在Host端分配buffer并初始化输入数据:

```
const int ni = 1, ci = 64, hi = 32, wi = 32;
const int no = 1, co = 64, ho = 30, wo = 30;
int input_count = ni * ci * hi * wi;
int output_count = no * co * ho * wo;

float *input_cpu_data = (float*) malloc(input_count * sizeof(float));
float *output_cpu_data = (float*) malloc(output_count * sizeof(float));
for (int index = 0; index < input_count; index++) {
        input_cpu_data[index] = (rand() % 100 / 100.0);
}
for (int index = 0; index < output_count; ++index) {
    output_cpu_data[index] = 0.0;
}

int16_t *input_cpu_ptr = (int16_t *)malloc(input_count * sizeof(int16_t));
int16_t *output_cpu_ptr = (int16_t *)malloc(output_count * sizeof(int16_t
));
cnrtCastDataType(input_cpu_data, CNRT_FLOAT32, input_cpu_ptr, CNR
T_FLOAT16, input_count, NULL);
... ...
```

②创建PluginOp操作所需的cnml Tensor
创建输入CNML Tensor,
创建输出CNML Tensor

```
const int dimNum = 4;
int input_shape[] = {ni, ci, hi, wi};
int output_shape[] = {no, co, ho, wo};
cnmlTensor_t input_tensor = NULL;
cnmlCreateTensor_V2(&input_tensor, CNML_TENSOR);
cnmlSetTensorShape_V2(input_tensor, dimNum, input_shape, NULL);
cnmlSetTensorDataType(input_tensor, CNML_DATA_FLOAT16);

cnmlTensor_t output_tensor = NULL;
cnmlCreateTensor_V2(&output_tensor, CNML_TENSOR);
cnmlSetTensorShape_V2(output_tensor, dimNum, output_shape, NULL
);
cnmlSetTensorDataType(output_tensor, CNML_DATA_FLOAT16);
```

③ cnrtKernelParamsBuffer_t参数对象声明和初始化

参数一一对应

```
typedef uint16_t half;
int param1;
bool param2;
float param3, param4;
half param3_half, param4_half;
cnrtConvertFloatToHalf(&param3_half, param3);
cnrtConvertFloatToHalf(&param4_half, param4);

cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferMarkInput(params);
cnrtKernelParamsBufferMarkOutput(params);
cnrtKernelParamsBufferAddParam(params, &param1, size
of(int));
cnrtKernelParamsBufferAddParam(params, &param2, size
of(bool));
cnrtKernelParamsBufferAddParam(params, &param3_half,
 sizeof(half));
cnrtKernelParamsBufferAddParam(params, &param4_half,
 sizeof(half));
```

MLU端的kernel函数

```
__mlu_entry__ void Kernel(
    half* input_ptr,  //输入数据指针
    half* output_ptr,  //输出数据指针
    int param1,
    bool param2,
    half param3,
    half param4);
```

④ 创建PluginOp操作描述符

⑤ 编译PluginOp以生成MLU运行的指令

⑥ 在MLU上为输入数据和输出数据分配空间

⑦ 将Host端的输入数据拷贝到设备空间

```
cnmlTensor_t cnml_input_array[1];
cnmlTensor_t cnml_output_array[1];
cnml_input_array[0] = input_tensor;
cnml_output_array[0] = output_tensor;

cnmlBaseOp_t op;
cnmlCreatePluginOp(&op,
    "test",
    reinterpret_cast<void**>(&Kernel),
    params,
    cnml_input_array,
    1,
    cnml_output_array,
    1,
    nullptr,
    0);

cnmlCompileBaseOp(op, CNML_MLU270, 1);

void *input_cnml_data = NULL;
void *output_cnml_data = NULL;
cnrtMalloc(&input_cnml_data, input_count * sizeof(float));
cnrtMalloc(&output_cnml_data, output_count * sizeof(float));

cnrtMemcpy(input_cnml_data, input_cpu_ptr, input_count *
sizeof(float), CNRT_MEM_TRANS_DIR_HOST2DEV);
```

Kernel函数名

Kernel参数结构

输入输出Tensor地址

⑧ 创建queue

```
cnrtQueue_t queue;
cnrtCreateQueue(&queue);
```

⑨ PluginOp操作计算

```
void* mlu_input_ptrs[1];
void* mlu_output_ptrs[1];
mlu_input_ptrs[0] = input_cnml_data;
mlu_output_ptrs[0] = output_cnml_data;

cnmlComputePluginOpForward_V4(op,
    &input_tensor,
    mlu_input_ptrs,
    1,
    &output_tensor,
    mlu_output_ptrs,
    1,
    &queue,
    NULL);
```

⑩ 同步并销毁queue

```
cnrtSyncQueue(queue);
cnrtDestroyQueue(queue);
```

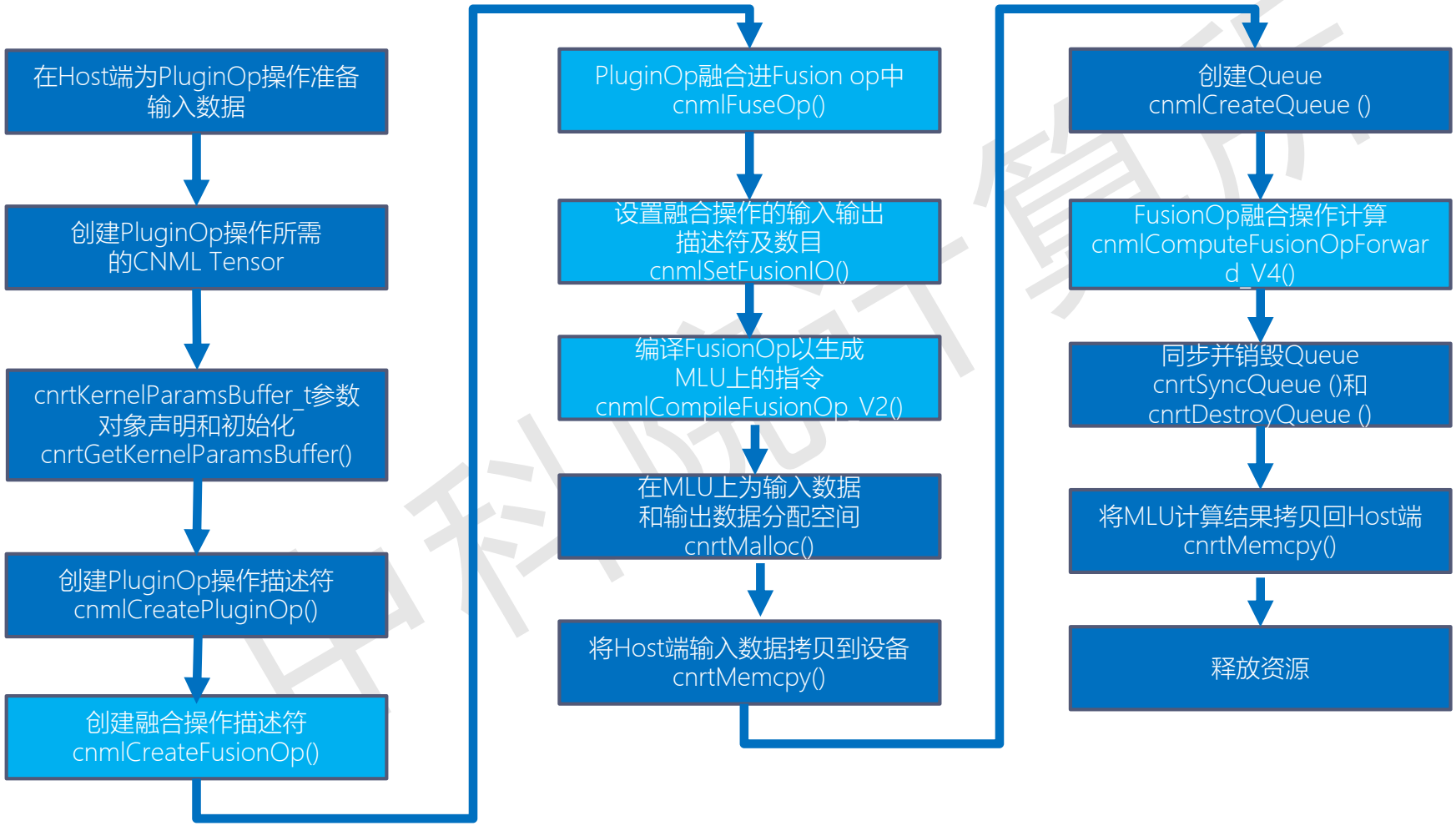
⑪ 将MLU端计算结果拷贝回Host端

```
cnrtMemcpy(output_cpu_prt output_cnml_data, output_co
unt * sizeof(float), CNRT_MEM_TRANS_DIR_DEV2HOST);
cnrtCastDataType(output_cpu_ptr, CNRT_FLOAT16, output_
cpu_data, CNRT_FLOAT32, output_count, NULL);
```

⑫ 释放资源

```
cnrtDestroyKernelParamsBuffer(params);
cnmlDestroyTensor(&input_tensor);
cnmlDestroyTensor(&output_tensor);
cnmlDestroyBaseOp(&op);
free(input_cpu_data);
free(output_cpu_data);
```

PluginOp算子集成：融合算子



主机侧代码

① 在Host端为PluginOp准备输入数据
输入数据维度为: 1 x 64 x 32 x 32 (NCHW),
输出数据维度为: 1 x 64 x 30 x 30(NCHW)
在Host端分配buffer并初始化输入数据:

```
const int ni = 1, ci = 64, hi = 32, wi = 32;
const int no = 1, co = 64, ho = 30, wo = 30;
int input_count = ni * ci * hi * wi;
int output_count = no * co * ho * wo;

float *input_cpu_data = (float*) malloc(input_count * sizeof(float));
float *output_cpu_data = (float*) malloc(output_count * sizeof(float));
for (int index = 0; index < input_count; index++) {
        input_cpu_data[index] = (rand() % 100 / 100.0);
}
for (int index = 0; index < output_count; ++index) {
    output_cpu_data[index] = 0.0;
}

int16_t *input_cpu_ptr = (int16_t *)malloc(input_count * sizeof(int16_t));
int16_t *output_cpu_ptr = (int16_t *)malloc(output_count * sizeof(int16_t
));
cnrtCastDataType(input_cpu_data, CNRT_FLOAT32, input_cpu_ptr, CNR
T_FLOAT16, input_count, NULL);
... ...
```

②创建PluginOp操作所需的cnml Tensor
创建输入CNML Tensor,
创建输出CNML Tensor

```
const int dimNum = 4;
int input_shape[] = {ni, ci, hi, wi};
int output_shape[] = {no, co, ho, wo};
cnmlTensor_t input_tensor = NULL;
cnmlCreateTensor_V2(&input_tensor, CNML_TENSOR);
cnmlSetTensorShape_V2(input_tensor, dimNum, input_shape, NULL);
cnmlSetTensorDataType(input_tensor, CNML_DATA_FLOAT16);

cnmlTensor_t output_tensor = NULL;
cnmlCreateTensor_V2(&output_tensor, CNML_TENSOR);
cnmlSetTensorShape_V2(output_tensor, dimNum, output_shape, NULL
);
cnmlSetTensorDataType(output_tensor, CNML_DATA_FLOAT16);
```

③ cnrtKernelParamsBuffer_t参数对象声明和初始化

参数一一对应

```
typedef uint16_t half;
int param1;
bool param2;
float param3, param4;
half param3_half, param4_half;
cnrtConvertFloatToHalf(&param3_half, param3);
cnrtConvertFloatToHalf(&param4_half, param4);

cnrtKernelParamsBuffer_t params;
cnrtGetKernelParamsBuffer(&params);
cnrtKernelParamsBufferMarkInput(params);
cnrtKernelParamsBufferMarkOutput(params);
cnrtKernelParamsBufferAddParam(params, &param1, size
of(int));
cnrtKernelParamsBufferAddParam(params, &param2, size
of(bool));
cnrtKernelParamsBufferAddParam(params, &param3_half,
 sizeof(half));
cnrtKernelParamsBufferAddParam(params, &param4_half,
 sizeof(half));
```

MLU端的kernel函数

```
__mlu_entry__ void Kernel(
    half* input_ptr,  //输入数据指针
    half* output_ptr,  //输出数据指针
    int param1,
    bool param2,
    half param3,
    half param4);
```

④ 创建PluginOp操作描述符

⑤ 创建融合操作描述符
cnmlCreateFusionOp()

⑥ 将PluginOp融合进Fusion Op中

⑦ 设置融合操作的输入数据和输出数据的CNML Tensor描述符及其数目

```
cnmlTensor_t cnml_input_array[1];
cnmlTensor_t cnml_output_array[1];
cnml_input_array[0] = input_tensor;
cnml_output_array[0] = output_tensor;

cnmlBaseOp_t op;
cnmlCreatePluginOp(&op,
    "test",
    reinterpret_cast<void**>(&Kernel),
    params,
    cnml_input_array,
    1,
    cnml_output_array,
    1,
    nullptr,
    0);

cnmlFusionOp_t  fusion_op;
cnmlCreateFusionOp(&fusion_op);
cnmlSetFusionOpCorenum(fusion_op, 1);
cnmlSetFusionOpCoreVersion(fusion_op, CNML_MLU2
70);

cnmlFuseOp(op, fusion_op);

cnmlSetFusionIO(fusion_op, cnml_input_array, 1,
cnml_output_array, 1);
```

⑧ 编译FusionOp以生产MLU上运行的指令

```
cnmlCompileFusionOp_V2(fusion_op);
```

⑨ 在MLU上为输入数据和输出数据分配空间

```
void *input_cnml_data = NULL;
void *output_cnml_data = NULL;
cnrtMalloc(&input_cnml_data, input_count * sizeof(float));
cnrtMalloc(&output_cnml_data, output_count * sizeof(float));
```

⑩ 将Host端的输入数据拷贝到设备空间

```
cnrtMemcpy(input_cnml_data, input_cpu_ptr, input_count * si
zeof(float), CNRT_MEM_TRANS_DIR_HOST2DEV);
```

⑪ 创建queue

```
cnrtQueue_t queue;
cnrtCreateQueue(&queue);
```

⑫ FusionOp融合计算

```
void* mlu_input_ptrs[1];
void* mlu_output_ptrs[1];
mlu_input_ptrs[0] = input_cnml_data;
mlu_output_ptrs[0] = output_cnml_data;

cnmlComputeFusionOpForward_V4(fusion_op,
                              &input_tensor, mlu_input_ptrs,
                               1,
                              &output_tensor, mlu_output_ptrs,
                               1,
                              queue,
                              NULL);
```

⑬ 同步并销毁queue

```
cnrtSyncQueue(queue);
cnrtDestroyQueue(queue);
```

⑭ 将MLU端计算结果拷贝回Host端

```
cnrtMemcpy(output_cpu_ptr,output_cnml_data,output_count * sizeof(int16_t), CNRT_MEM_TRANS_DIR_DEV2HOST);

cnrtCastDataType(output_cpu_ptr, CNRT_FLOAT16, output_cpu_data, CNRT_FLOAT32, output_count, NULL);
```

⑮ 释放资源

```
cnrtDestroyKernelParamsBuffer(params);
cnmlDestroyTensor(&input_tensor);
cnmlDestroyTensor(&output_tensor);
cnmlDestroyFusionOp(&fusion_op);
free(input_cpu_data);
free(output_cpu_data);
```

编程框架算子开发

- TensorFlow集成自定义算子

通过高性能库自定义算子API，可以将用编程语言开发的新算子集成到TensorFlow等编程框架中。在框架中的主要流程包括：

1）为新的自定义算子进行注册；

2）为新的自定义算子编写前向传播接口函数；

3）根据新的自定义算子接口编写定制算子底层实现；

4）完善Bazel Build和头文件，并重新编译TensorFlow源码。

编程框架算子开发：集成自定义算子

- 以集成anchor_generator的自定义算子到TensorFlow为例
  - 涉及修改的目录树

自定义算子的C++接口
声明和注册；
前向计算的wrapper实现

自定义算子的Python接口注册

自定义算子的BCL实现

调用高性能库
自定义算子API实现

```
tensorflow/
├ core
|    ├ BUILD
|    ├ kernels
|    |    ├ anchor_generator_dlp_op.cc
|    |    └ BUILD
|    └ ops
|         └ anchor_generator_ops.cc
├ python
|    ├ BUILD
|    └ ops
|         ├ anchor_generator_ops.py
|         └ standard_ops.py
└ stream_executor
     ├ BUILD
     └ dlp
          ├ lib_ops
          |    ├ dlp_anchor_generate_op.cc
          ├ macro.h
          ├ dlp_anchor_generator_kernel.dlp
          ├ dlp_anchor_generator_kernel.o
          ├ dlp.h
          ├ dlp_lib_common.cc
          ├ dlp_lib_math_ops.h
          ├ dlp_lib_nn_ops.h
          ├ dlp_stream.cc
          ├ dlp_stream.h
          └ ops
               └ anchor_generator_op.cc
```

- 在tensorflow/core/ops/anchor_generator_ops.cc中进行AnchorGenerator的算子注册

算子名
输入输出
算子参数
（名字、类型及默认值）

```
namespace tensorflow {

REGISTER_OP("AnchorGenerator")
    .Input("feature_map_shape: int32")
    .Output("anchors: float")
    .Attr("scales: list(float) = [0.5, 1, 2]")
    .Attr("aspect_ratios: list(float) = [0.5, 1, 2]")
    .Attr("base_anchor_sizes: list(float) = [256, 256]")
    .Attr("anchor_strides: list(float) = [16, 16]")
    .Attr("anchor_offsets: list(float) = [8, 8]")
    //others attrs

    .SetShapeFn(AnchorGeneratorShapeFn);

}  // namespace tensorflow
```

- 在tensorflow/core/kernels/anchor_generator_dlp_op.cc中定义DLPAnchorGeneratorOp类
  - 实际计算的外层wrapper封装

获取输入并创建输出张量

自定义计算入口

```
namespace tensorflow {

class DLPAnchorGeneratorOp : public OpKernel {
  public:
    explicit DLPAnchorGeneratorOp(OpKernelConstruction* context) : OpKernel(
     context) {
    }

    void Compute(OpKernelContext* context) override {
      // 获取自定义算子的输入张量和属性等
      Tensor input_tensor = GetInputTensor();
      // 创建输出张量
      Tensor output_tensor = CreateAndMallocOutputTensor();
      // 调用执行器中执行运算的接口
      context->Compute();
    }
};

REGISTER_KERNEL_BUILDER(Name("AnchorGenerator").Device(DEVICE_DLP),
      DLPAnchorGeneratorOp);

}  // namespace tensorflow
```

- 底层tensorflow/stream_executor/dlp/中ops及lib_ops的C++代码直接调用高性能计算库自定义算子接口，实现前向计算
  - 调用CreateCustomizedOp
  - CompileBaseOp
  - ComputeCustomizedOP等接口

创建自定义算子

编译基础算子

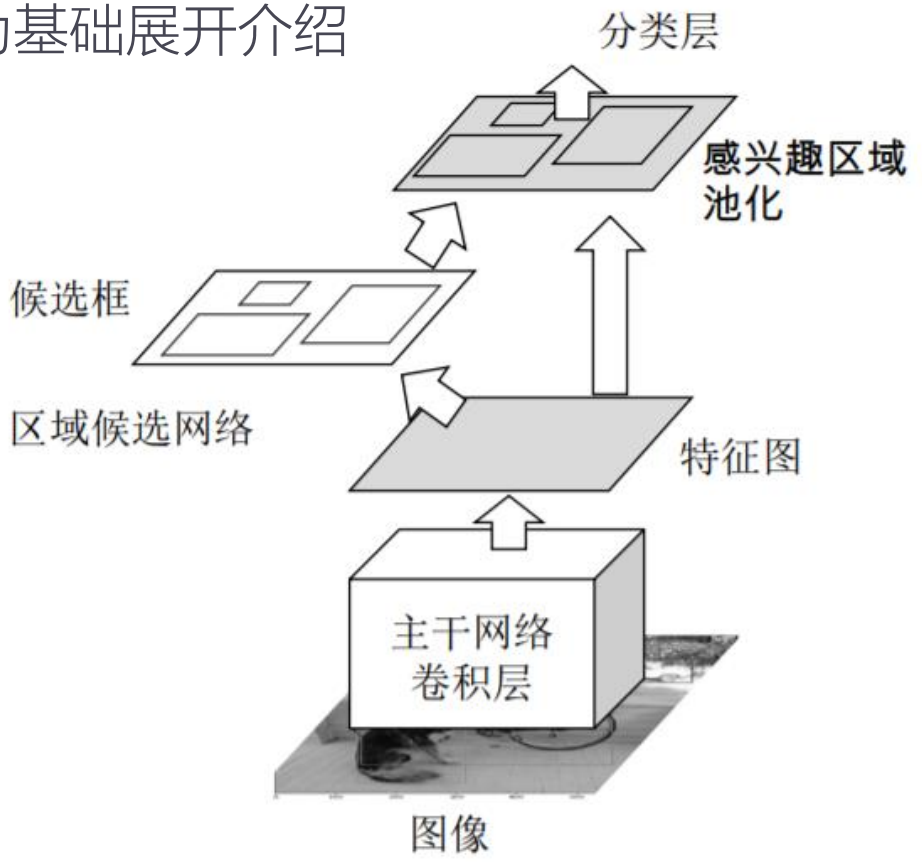
自定义算子计算

```
namespace stream_executor {
namespace dlp {

DLPStatus DLPStream::AnchorGenerate(std::vector<Tensor*> inputs,
                                    std::vector<Tensor*> outputs,
                                    half *feature_map_shape_dlp,
                                    /*other params */
                                    ... ...) {
    ... ...

    // 创建参数缓存
    KernelParamsBuffer_t params;
    GetKernelParamsBuffer(&params);
    KernelParamsBufferAddParam(params, &feature_map_shape_dlp, sizeof(half*))
     ;

    // 创建自定义算子
    BaseOp_t op;
    CreateCustomizedOp(&op, "anchorgenerator",
                       reinterpret_cast<void*>(AnchorGeneratorKernel),
                       params,
                       inputs.data(), inputs.size(),
                       outputs.data(), outputs.size(),
                       nullptr, 0);

    // 编译基础算子
    CompileBaseOp(op);

    // 数据分配与准备
    ... ...

    // 自定义算子计算
    ComputeCustomizedOp(op,
                        inputs.data(),
                        inputs.size(),
                        outputs.data(),
                        outputs.size(),
                        queue);
    ... ...
    return DLP_STATUS_SUCCESS;
```

实践：系统开发与优化

- 以典型的目标检测网络Faster R-CNN为例，阐述完整的系统级开发与优化方法及流程
  - 这里直接以BANG和CNML为基础展开介绍

Faster R-CNN是典型的两阶段检测网络，提出了RPN层来解决传统检测网络生成检测框耗时的问题。通过RPN层来**直接生成检测框**，极大地提升了检测网络性能。

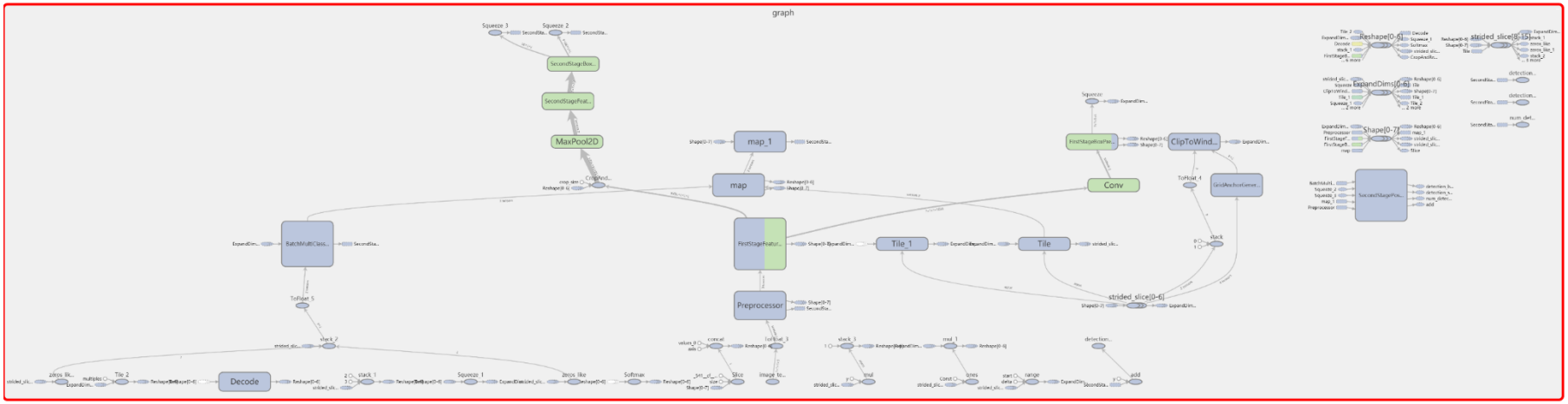


- Faster R-CNN网络主要流程
  - 主干网络（*backbone*）：常见的深度卷积网络（VGG、ResNet或Inception系列）作为特征提取的主干网络。提取出来的网络特征被后续的RPN 层和分类网络共享；
  - 区域候选网络（*Region Proposal Network，RPN*）：用于生成检测候选框Region Proposals，结合预设在每个特征点的Anchors。该层通过softmax 判断每个Anchor 属于正样本或负样本，再利用位置回归得到较为精确的Region Proposals。**此处为网络的第一阶段**；
  - 兴趣区域池化（*Region-of-interest Pooling，ROI Pooling*）：该层以特征图和RegionProposals 为输入，把每个Proposal 映射到对应的特征图中，并取出该特征进行池化得到相同的尺寸，送入后续分类层；
  - 分类层：该层对ROI Pooling 处理后的特征图进行类别判断和位置回归修正，得到最终类别和位置。**此处为网络的第二阶段**。

- 用TensorBoard查看TensorFlow官方Faster R-CNN计算图

1）蓝色部分是某版本DLP软件栈暂不支持的算子，需要放在CPU上执行

2）导致DLP运算被分段，CPU运算（蓝色部分）过多，网络整体性能差



进行性能分析后得到

网络名	CPU 执行时间	数据拷贝时间	DLP 执行时间
Faster R-CNN	26.23%	17.04%	55.40%

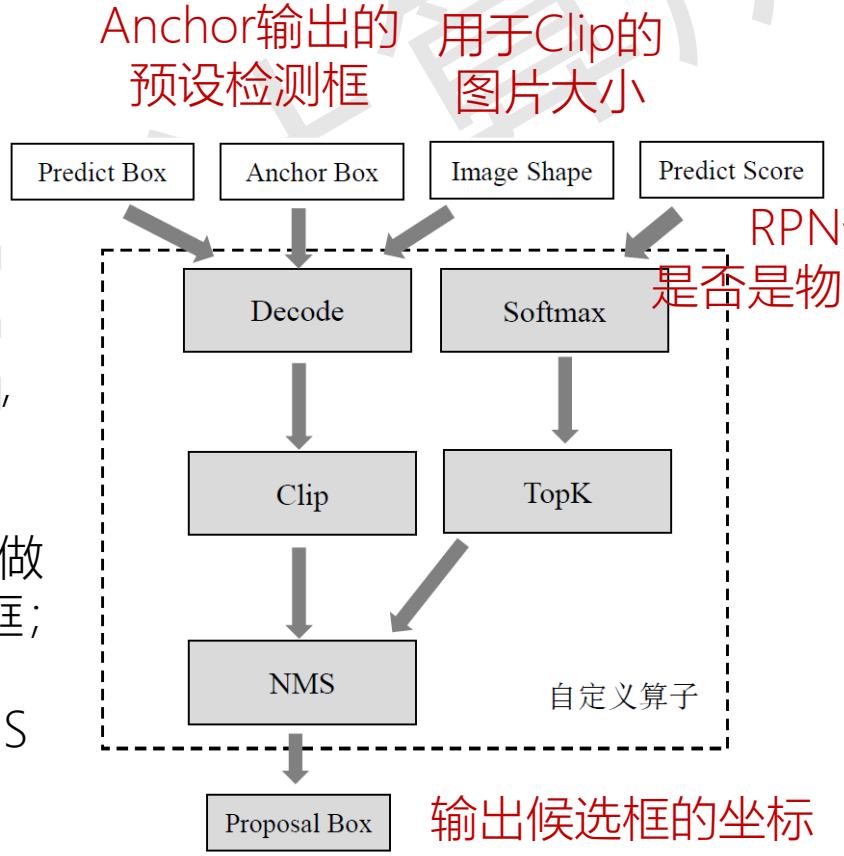
- 优化思路：**减少网络中CPU运算，通过自定义算子实现在DLP上运行**，减少数据拷贝，利用DLP并行算力

- 使用BANG进行自定义算子开发和优化的主要流程：
  - 分析网络中暂不支持操作的位置和功能，明确要开发的自定义算子；
  - 采用BANG实现和优化分析出来的自定义算子；
  - 采用CNML自定义算子接口将新开发的自定义算子集成到TensorFlow框架中并重新编译；
  - 采用集成了自定义算子的TensorFlow对替换后的网络模型进行性能和精度验证

融合算子开发

- 观察到Faster R-CNN在CPU上执行的部分主要是**两个阶段的后处理部分**，针对两阶段的后处理开发相应的**融合算子**

- 第一阶段后处理算子处理流程



1、对偏移量和Anchor进行decode，用clip对得到的检测框进行裁剪；

2、对预测的分数进行softmax，并做topK排序找到目标概率较大的K个框；

3、根据排序的分数和检测框做NMS滤除重复的框。

- Decode & Clip运算
  - 计算anchor中心坐标和宽高

$$cy = (y_0 + y_1)/2$$

$$cx = (x_0 + x_1)/2$$

$$h = y_1 - y_0$$

$$w = x_1 - x_0$$

  - 计算修正后框的中心坐标和宽高

$$ncy = cy + dy \times h$$

$$ncx = cx + dx \times w$$

$$nh = exp(dh) \times h$$

$$nw = exp(dw) \times w$$

  - 计算修正后框的最小/最大x，y坐标

$$y_{min} = max(ncy - nh/2, 0.0)$$

$$x_{min} = max(ncx - nw/2, 0.0)$$

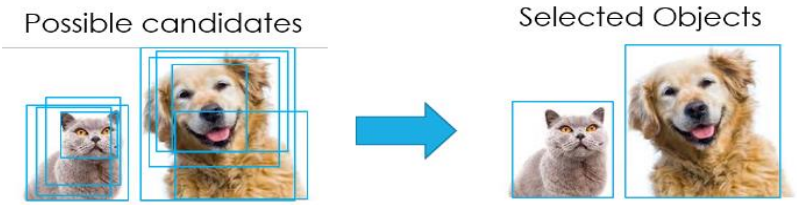
$$y_{max} = min(ncy + nh/2, img\_h)$$

$$x_{max} = min(ncx + nw/2, img\_w)$$

```
// y0 = anchor;
// x0 = anchor + SEGMENT_SIZE;
// y1 = anchor * 2 * SEGMENT_SIZE;
// x1 = anchor + 3 * SEGMENT_SIZE;
// cy = (y0 + y1) / 2;
// cx = (x0 + x1) / 2;
// h  = y1 - y0;
// w  = x1 - x0;
__bang_add(tmp1, anchor_y1, anchor_y2, SEGMENT_SIZE);
__bang_mul_const(_cy, tmp1, 0.5, SEGMENT_SIZE);
__bang_add(tmp1, anchor_x1, anchor_x2,SEGMENT_SIZE);
__bang_mul_const(_cx, tmp1, 0.5, SEGMENT_SIZE);
__bang_sub(_h, anchor_y2, anchor_y1, SEGMENT_SIZE);
__bang_sub(_w, anchor_x2, anchor_x1, SEGMENT_SIZE);                    1

// dy = delt;
// dx = delt + SEGMENT_SIZE;
// dh = delt + 2 * SEGMENT_SIZE;
// dw = delt + 3 * SEGMENT_SIZE;
// ncy = cy + dy * h;
// ncx = cx + dx * w;
// nh  = exp(dh) * h;
// nw  = exp(dw) * w;
__bang_mul(tmp1, delt_dy, _h, SEGMENT_SIZE);
__bang_add(_ncy, tmp1, _cy, SEGMENT_SIZE);

__bang_mul(tmp2, delt_dx, _w, SEGMENT_SIZE);
__bang_add(_ncx, tmp2, _cx, SEGMENT_SIZE);

__bang_active_exp(tmp3, delt_dh, SEGMENT_SIZE);
__bang_active_exp(tmp4, delt_dw, SEGMENT_SIZE);

__bang_mul(_nh, tmp3, _h , SEGMENT_SIZE);
__bang_mul(_nw, tmp4, _w , SEGMENT_SIZE);                              2
// box[k + i + 0 * AWH_PLUS] = max(_ncy[i] - _nh[i] / 2, (half)0);
__bang_mul_const(_nh, _nh, 0.5, SEGMENT_SIZE);
__bang_sub(tmp1, _ncy, _nh, SEGMENT_SIZE);
__bang_active_relu(_y0, tmp1, SEGMENT_SIZE);

// box[k + i + 1 * AWH_PLUS] = max(_ncx[i] - _nw[i] / 2, (half)0);
__bang_mul_const(_nw, _nw, 0.5, SEGMENT_SIZE);
__bang_sub(tmp1, _ncx, _nw, SEGMENT_SIZE);
__bang_active_relu(_x0, tmp1, SEGMENT_SIZE);

// box[k + i + 2 * AWH_PLUS] = min(_ncy + _nh / 2, im_h);
__bang_add(tmp2, _ncy, _nh, SEGMENT_SIZE);
__bang_minrelu_256(_y1, tmp2, im_h, tmp128);

// box[k + i + 3 * AWH_PLUS] = min(_ncx + _nw / 2, im_w);
__bang_add(tmp2, _ncx, _nw, SEGMENT_SIZE);
__bang_minrelu_256(_x1, tmp2, im_w, tmp128);                           3
```

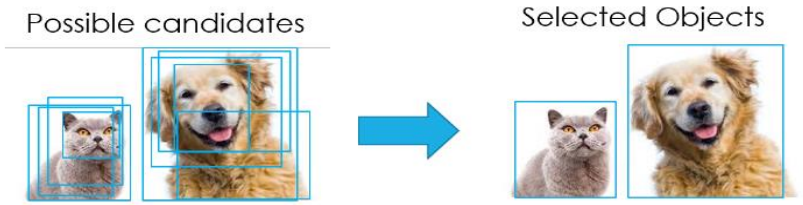
- Softmax & TopK运算
  - 删除面积为0的框
  - 对分数做softmax
  - 去除softmax后是物体的概率小于0.01的框
  - 有选择的拷贝结果到gdram

```
// remove 0-area boxes
__bang_write_zero(tmp1, SEGMENT_SIZE);
__bang_sub(tmp2, _y1, _y0, SEGMENT_SIZE);
__bang_gt(tmp2, tmp2, tmp1, SEGMENT_SIZE);
__bang_sub(tmp3, _x1, _x0, SEGMENT_SIZE);
__bang_gt(tmp3, tmp3, tmp1, SEGMENT_SIZE);
__bang_mul(valid_box, tmp2, tmp3, SEGMENT_SIZE);                    1

// do softmax:
// exp(score_+SEGMENT_SIZE)/(exp(score_)+exp(score_+SEGMENT_SIZE))
__bang_active_exp(tmp1, score_p, SEGMENT_SIZE);
__bang_active_exp(tmp2, score_n, SEGMENT_SIZE);
__bang_add(tmp2, tmp1, tmp2, SEGMENT_SIZE);
__bang_active_recip(tmp2, tmp2, SEGMENT_SIZE);
__bang_mul(score_p, tmp1, tmp2, SEGMENT_SIZE);                       2

// remove small score less than 0.01
nram_memset(tmp1, SEGMENT_SIZE, min_nms_score);
__bang_gt(tmp3, score_p, tmp1, SEGMENT_SIZE);
__bang_mul(valid_box, valid_box, tmp3, SEGMENT_SIZE);                3

// copy valid_box out
__bang_select(tmp1, _y0, valid_box, SEGMENT_SIZE);
__memcpy(temp_memory + count + 0 * AWH_PLUS_ +
                ALIGN_UP_TO(taskId * (AWH_PLUS_ / taskDim), 64),
         tmp1 + 64, SEGMENT_SIZE * sizeof(half), NRAM2GDRAM);
__bang_select(tmp1, _x0, valid_box, SEGMENT_SIZE);
__memcpy(temp_memory + count + 1 * AWH_PLUS_ +
                ALIGN_UP_TO(taskId * (AWH_PLUS_ / taskDim), 64),
         tmp1 + 64, SEGMENT_SIZE * sizeof(half), NRAM2GDRAM);
__bang_select(tmp1, _y1, valid_box, SEGMENT_SIZE);
__memcpy(temp_memory + count + 2 * AWH_PLUS_ +
                ALIGN_UP_TO(taskId * (AWH_PLUS_ / taskDim), 64),
         tmp1 + 64, SEGMENT_SIZE * sizeof(half), NRAM2GDRAM);
__bang_select(tmp1, _x1, valid_box, SEGMENT_SIZE);
__memcpy(temp_memory + count + 3 * AWH_PLUS_ +
                ALIGN_UP_TO(taskId * (AWH_PLUS_ / taskDim), 64),
         tmp1 + 64, SEGMENT_SIZE * sizeof(half), NRAM2GDRAM);
__bang_select(tmp1, score_p, valid_box, SEGMENT_SIZE);
uint16_t num_valid = *((uint16_t *)tmp1);
uint16_t num_valid_align = ALIGN_UP_TO(num_valid, 64);
for (int m = 0; m < (num_valid_align - num_valid); m++)
  *(tmp1 + 64 + num_valid + m) = NE_INF;
__memcpy(temp_memory + count + 4 * AWH_PLUS_ +
                ALIGN_UP_TO(taskId * (AWH_PLUS_ / taskDim), 64),
         tmp1 + 64, SEGMENT_SIZE * sizeof(half), NRAM2GDRAM);        4
count += num_valid_align;
```



- NMS运算
  - 计算所有候选框的面积
  - 取出当前候选框中概率分数最大的框，并把其概率分数置为NE_INF

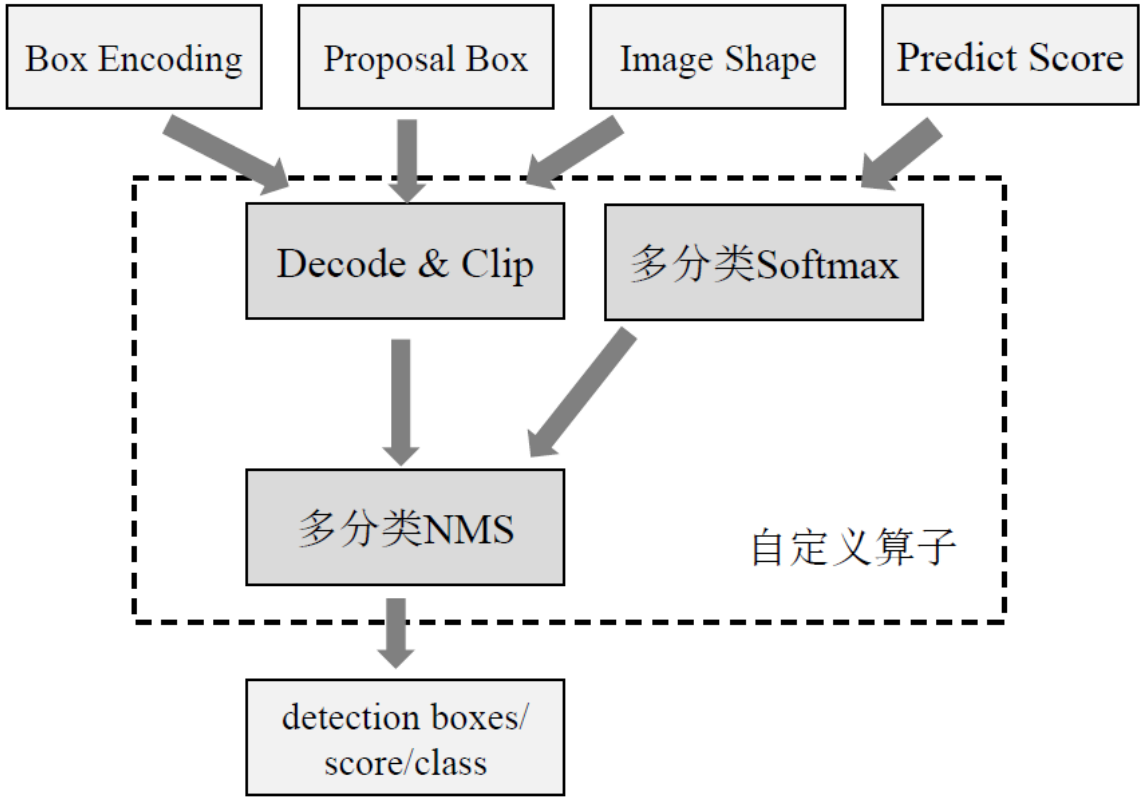
```
//(x2-x1)*(y2-y1)
// calc box area
__bang_write_zero(box_, box_count_aligned); //x=0
__bang_sub(box_, box + 2 * box_count_aligned, box, box_count_aligned);  //x=x2-x1
__bang_mul_const(box_, box_, nms_scale, box_count_aligned); // x=x*nms_scale
__bang_write_zero(area, box_count_aligned);  //y=0
__bang_sub(area, box + 3 * box_count_aligned, box + box_count_aligned, box_count_aligned); //y=y2-y1
__bang_mul_const(area, area, nms_scale, box_count_aligned); //y=y*nms_scale
__bang_mul(area, area, box_, box_count_aligned); //y=y*x                    1
__nram__ half result[ALIGN_SIZE * MAX_CORE_NUM];
__bang_write_zero(result, ALIGN_SIZE * MAX_CORE_NUM);

int local_max_idx;
half local_max_score;
half global_max_score;
int global_max_idx;
half box_idx_0, box_idx_1, box_idx_2, box_idx_3;
half area_idx;

for (int i = 0; i < nms_num; i++) {
  // 1. find local max
  __bang_max(result, scores_, box_count_aligned);
  local_max_idx = (int32_t)(*(uint32_t *)(result + 1));
  local_max_score = result[0];
  global_max_idx = local_max_idx + taskId * (AWH_PLUS_ / taskDim);
  result[0] = local_max_score;
  *(int16_t *)(result + 1) = (int16_t)global_max_idx;
  result[2] = box[local_max_idx + 0 * box_count_aligned];
  result[3] = box[local_max_idx + 1 * box_count_aligned];
  result[4] = box[local_max_idx + 2 * box_count_aligned];
  result[5] = box[local_max_idx + 3 * box_count_aligned];
  result[6] = area[local_max_idx];
  // 2. send local max to gdram_buf
  __memcpy(tmp_gdram_buf + taskId * ALIGN_SIZE, result,
           ALIGN_SIZE * sizeof(half), NRAM2GDRAM);
  if (taskDim > 1) {
    __sync_all_ipu();
  }
  // 3. fetch gdram_buf
  __memcpy(result, tmp_gdram_buf, ALIGN_SIZE * taskDim * sizeof(half),
           GDRAM2NRAM);
  // 4. all cores find global max
  global_max_idx = (int32_t)(*(int16_t *)(result + 1));
  global_max_score = result[0];
  int max_j = 0;
  for (int j = ALIGN_SIZE; j < ALIGN_SIZE * taskDim; j += ALIGN_SIZE) {
    int tmp_idx = (int32_t)(*(int16_t *)(result + j + 1));
    half tmp_score = result[j];
    if (global_max_score < tmp_score) {
      global_max_score = tmp_score;
      global_max_idx = tmp_idx;
      max_j = j;
    }
  }
  global_max_score = result[max_j];
  global_max_idx = (int32_t)(*(int16_t *)(result + max_j + 1));
  box_idx_0 = result[max_j + 2];
  box_idx_1 = result[max_j + 3];
  box_idx_2 = result[max_j + 4];
  box_idx_3 = result[max_j + 5];
  area_idx = result[max_j + 6];
  // if max_idx is on this core, mark it
  if (max_j == ALIGN_SIZE * taskId) {
    scores_[local_max_idx] = NE_INF;
  }
  __nramset_half(intmp_areas, NMS_STEP_SIZE, area_idx);          2
  const int upper_bound =
```



- NMS运算
  - 选出的候选框与剩余的框依次计算IOU（交并比）
  - IOU大于阈值的框，概率分数被置为NE_INF
  - 循环回到步骤2继续，找到nms_num个候选框后退出

```
// if max idx is on this core, mark it
if (max_j == ALIGN_SIZE * taskId) {
  scores_[local_max_idx] = NE_INF;
}
__nramset_half(intmp_areas, NMS_STEP_SIZE, area_idx);

const int upper_bound =
    box_count_aligned - box_count_aligned % NMS_STEP_SIZE;
for (int k = 0; k < upper_bound; k += NMS_STEP_SIZE) {
  __bang_maxrelu_NMS_STEP(inter_x1, box + k, box_idx_0, int_GE_ab);
  __bang_minrelu_NMS_STEP(inter_x2, box + k + 2 * box_count_aligned,
                          box_idx_2, int_GE_ab);
  __bang_maxrelu_NMS_STEP(inter_y1, box + k + box_count_aligned, box_idx_1,
                          int_GE_ab);
  __bang_minrelu_NMS_STEP(inter_y2, box + k + 3 * box_count_aligned,
                          box_idx_3, int_GE_ab);

  // max(0, inter_x2 - inter_x1 +1);
  __bang_sub(intmp2, intmp_const_1, inter_x1, NMS_STEP_SIZE);
  __bang_add(intmp2, intmp2, inter_x2, NMS_STEP_SIZE);
  __bang_active_relu(intmp2, intmp2, NMS_STEP_SIZE);
  __bang_mul_const(intmp2, intmp2, nms_scale, NMS_STEP_SIZE);

  // max(0, inter_y2 - inter_y1 +1);
  __bang_sub(intmp3, intmp_const_1, inter_y1, NMS_STEP_SIZE);
  __bang_add(intmp3, intmp3, inter_y2, NMS_STEP_SIZE);
  __bang_active_relu(intmp3, intmp3, NMS_STEP_SIZE);
  __bang_mul_const(intmp1, intmp3, nms_scale, NMS_STEP_SIZE);

  // inter_area = max(0, inter_x2 - inter_x1 + 1) * max(0, inter_y2 -
  // inter_y1 + 1);
  __bang_mul(intmp3, intmp2, intmp1, NMS_STEP_SIZE);

  // over = area[idx] + area[k] - inter_area
  __bang_sub(intmp2, intmp_areas, intmp3, NMS_STEP_SIZE);
  __bang_add(intmp1, intmp2, area + k, NMS_STEP_SIZE);
                                                          3
  __bang_mul_const(intmp1, intmp1, nms_thresh, NMS_STEP_SIZE);
  __bang_le(intmp2, intmp3, intmp1, NMS_STEP_SIZE);

  __bang_not(intmp3, intmp2, NMS_STEP_SIZE);
  __bang_mul_const(intmp1, intmp3, NE_INF, NMS_STEP_SIZE);
  __bang_mul(scores_ + k, intmp2, scores_ + k, NMS_STEP_SIZE);
  __bang_add(scores_ + k, intmp1, scores_ + k, NMS_STEP_SIZE);
                                                          4
}
```

- 第二阶段后处理算子处理流程



通过CNML接口进行框架集成

- 需修改的TensorFlow目录结构
  - tensorflow/core

新算子接口声明，新算子注册，新算子forward最外层wrapper

  - tensorflow/python

Python接口的封装

  - tensorflow/stream_executor/mlu/mlu_api/ops
  - tensorflow/stream_executor/mlu/mlu_api/lib_ops

新算子forward内层wrapper及实现

```
|-- core
|   |-- kernels
|   |   |-- BUILD
|   |   |-- cwise_op_postprocess_rpn_op.cc
|   |   `-- cwise_op_postprocess_rpn_op_mlu.h
|   `-- ops
|       `-- math_ops.cc
|-- python
`-- stream_executor
    |-- BUILD
    `-- mlu
        |-- mlu.h
        |-- mlu_api
        |   |-- lib_ops
        |   |   |-- mlu_lib_common.cc
        |   |   |-- mlu_lib_common.h
        |   |   |-- mlu_lib_ops.cc
        |   |   |-- mlu_lib_ops.h
        |   |   |-- util.cc
        |   |   `-- util.h
        |   |-- ops
        |   |   `-- postprocess_rpn_op.cc
        |   |-- tf_mlu_intf.cc
        |   `-- tf_mlu_intf.h
        |-- mlu_executor_cache.cc
        |-- mlu_executor_cache.h
        |-- mlu_lib_math_ops.h
        |-- mlu_platform.cc
        |-- mlu_platform.h
        |-- mlu_platform_id.cc
        |-- mlu_platform_id.h
        |-- mlu_stream.cc
        |-- mlu_stream.h
        |-- mlu_stream_executor.cc
        |-- mlu_stream_executor.h
        |-- mlu_stream_records.cc
        |-- mlu_stream_records.h
        `-- ops
```

- 第一阶段后处理融合算子接口声明和注册
  - 算子输入tensor名字，类型声明
  - 算子输出tensor名字，类型声明
  - 算子属性名字，类型，默认值声明
  - 算子注册，设备为DEVICE_MLU，对应接口类为MLUPostprocessRpnOp

tensorflow/core/ops/math_ops.cc

tensorflow/core/kernels/cwise_op_postprocess_rpn_op.cc

```
REGISTER_OP("PostprocessRpn")
  //input
  .Input("rpn_box_encodings_batch: T")
  .Input("rpn_objectness_predictions_with_background_batch: T")
  .Input("image_shapes: int32")
  .Input("anchors: T")
  //output
  .Output("proposal_boxes: T")

  //attr
  .Attr("T : {bfloat16, float, half, float32, double, int32, int64, complex64, "
        "complex128}")
  .Attr("score_thresh: float = 0.0")
  .Attr("iou_thresh: float = 0.7")
  .Attr("max_size_per_class: int = 300")
  .Attr("max_total_size: int = 300")
  .Attr("scale_xy: float = 0.1")
  .Attr("scale_wh: float = 0.2")
  .Attr("min_nms_score: float = 0.01")

  //shape
  .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
    //input shape check
    shape_inference::ShapeHandle rpn_box_shape;//[batch_size, num_anchors, 1, self._box_coder.code_size]
    TF_RETURN_IF_ERROR(c->WithRank(c->input(0), 4, &rpn_box_shape));
    shape_inference::ShapeHandle rpn_score_shape;//[batch_size, num_anchors, 2]
    TF_RETURN_IF_ERROR(c->WithRank(c->input(1), 3, &rpn_score_shape));
    shape_inference::ShapeHandle image_shape;//shape:[batch_size, h, w, c]
    TF_RETURN_IF_ERROR(c->WithRank(c->input(2), 1, &image_shape));
    shape_inference::ShapeHandle anchors_shape;//[1, height, width, num_anchors_per_location * 4]
    TF_RETURN_IF_ERROR(c->WithRank(c->input(3), 4, &anchors_shape));

    //set output shape
    shape_inference::DimensionHandle batch_size      = c->Dim(rpn_box_shape,0);
    shape_inference::DimensionHandle anchors_num     = c->Dim(rpn_box_shape,1);
    shape_inference::DimensionHandle box_encode_size = c->Dim(rpn_box_shape,3);
    shape_inference::DimensionHandle class_with_bg   = c->Dim(rpn_score_shape,2);

    int max_nms_out;
    c->GetAttr("max_total_size", &max_nms_out);
    shape_inference::DimensionHandle max_nms_dim  = c->MakeDim(max_nms_out);
    shape_inference::DimensionHandle constant_one = c->MakeDim(1);
    shape_inference::DimensionHandle constant_16  = c->MakeDim(16);

    vector<shape_inference::DimensionHandle> output;
    output.push_back(batch_size);
    output.push_back(max_nms_dim);
    output.push_back(box_encode_size);
    c->set_output(0, c->MakeShape(output));//[batch_size, max_num_proposals, 4]

    return Status::OK();
  });
```

```
REGISTER_KERNEL_BUILDER(Name("PostprocessRpn").Device(DEVICE_MLU).TypeConstraint<T>("T"),MLUPostprocessRpnOp<T>);
```

- 算子最外层wrapper接口实现
  - 获取输入Tensor
  - 根据算子的实际特性，创建算子需要的输出Tensor
  - 调用MLU stream的PostProcessRpn进入算子内层接口

tensorflow/core/kernels/cwise_post process_rpn_op_mlu.h

```
void ComputeOnMLU(OpKernelContext* context) override {
  // some checks to be sure ...
  DCHECK_EQ(4, context->num_inputs());
  // get the input tensor
  const Tensor& rpn_box_encodings_batch = context->input(0);//[batch_size, num_anchors, 1,self._box_coder.code_size]
  const Tensor& rpn_objectness_predictions_with_background_batch = context->input(1);//[batch_size, num_anchors, 2]
  const Tensor& image_shapes = context->input(2);//[batch_size, h, w, c]
  const Tensor& anchors = context->input(3);//[1, height, width, num_anchor_per_location*4]
  // check shapes of input
  const TensorShape& rpn_box_shape    = rpn_box_encodings_batch.shape();
  const TensorShape& rpn_score_shape = rpn_objectness_predictions_with_background_batch.shape();
  const TensorShape& anchors_shape    = anchors.shape();
  const TensorShape& img_shape = image_shapes.shape();

  DCHECK_EQ(rpn_box_shape.dims(), 4);
  DCHECK_EQ(rpn_box_shape.dim_size(3), 4);
  DCHECK_EQ(rpn_score_shape.dims(), 3);
  DCHECK_EQ(rpn_score_shape.dim_size(2), 2);
  DCHECK_EQ(anchors_shape.dims(), 4);
  DCHECK_EQ(img_shape.dims(), 1);
  //useful variables
  int batch_size    = rpn_box_shape.dim_size(0);
  int num_anchors   = rpn_box_shape.dim_size(1);
  int box_code_size = rpn_box_shape.dim_size(2);
  int class_with_bg = rpn_score_shape.dim_size(2);
  int max_nms_out   = max_total_size_;
  int max_nms_out_align = ALIGN_UP_TO(max_nms_out, 16);
  auto img_shape_flat = image_shapes.flat<int32>();
  float im_height = 600.0;//float(img_shape_flat(1));
  float im_width  = 800.0;//float(img_shape_flat(2));
  int feature_h   = anchors_shape.dim_size(1);
  int feature_w   = anchors_shape.dim_size(2);
  int anchor_per_feature = num_anchors / feature_h / feature_w;
  // create output shape
  TensorShape proposal_box_shape;//[batch_size, max_num_proposals, 4]
  proposal_box_shape.AddDim(batch_size);
  proposal_box_shape.AddDim(max_nms_out);
  proposal_box_shape.AddDim(4);
  // create output tensor
  Tensor* proposal_box = NULL;
  OP_REQUIRES_OK(context, context->allocate_output(0, proposal_box_shape, &proposal_box));
  auto* stream = context->op_device_context()->mlu_stream();
  stream->PostprocessRpn(context, const_cast<Tensor *>(&rpn_box_encodings_batch),
                         const_cast<Tensor *>(&rpn_objectness_predictions_with_background_batch),
                         const_cast<Tensor *>(&anchors),
                         batch_size, num_anchors, max_nms_out, iou_thresh_,
                         im_height, im_width,
                         feature_h, feature_w, anchor_per_feature,
                         scale_xy_, scale_wh_, min_nms_score_,
                         proposal_box);
}
```

- MLUStream层
  - MLUStream层负责MLUOps算子类的实例化，其接口都定义在tensorflow/stream_executor/mlu/mlu_stream.h中

```
Status PostprocessRpn(OpKernelContext *ctx,
    Tensor *rpn_box_encodings_batch, Tensor *rpn_objectness_predictions_with_background_batch,
    Tensor *anchors,
    int batch_size, int num_anchors, int max_nms_out, float iou_thresh_,
    float im_height, float im_width,
    int feature_h, int feature_w, int anchor_per_feature,
    float scale_xy, float scale_wh, float min_nms_score,
    Tensor *proposal_box){
    ops::MLUPostprocessRpnParam op_param(batch_size,num_anchors,max_nms_out,iou_thresh_,
    im_height,im_width,feature_h,feature_w,anchor_per_feature,scale_xy,scale_wh,
    min_nms_score);

    return CommonOpImpl<ops::MLUPostprocessRpn>(ctx,{rpn_box_encodings_batch,
    rpn_objectness_predictions_with_background_batch,anchors},
    {proposal_box},static_cast<void*>(&op_param));
    }
```

- 算子内层wrapper实现
  - Ops层
  - Create方法主要调用底层lib层的CreateMLUTensor和CreatePostprocessRpnOp，得到相应指针postprocessRpn_op
  - Compute方法的功能在于调用lib层的computePostprocessRpnOp函数进行计算

tensorflow/stream_executor/mlu/mlu_api/ops/postprocess_rpn_op.c

```
namespace stream_executor {
namespace mlu {
namespace ops {
Status MLUPostprocessRpn::CreateMLUOp(std::vector<MLUTensor *> &inputs,
std::vector<MLUTensor *> &outputs, void *param){
    TF_PARAMS_CHECK(inputs.size() > 0, "Missing input");
    TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
    MLUBaseOp *postprocessRpn_op = nullptr;
    MLUTensor *rpn_box_encodings_batch= inputs.at(0);
    MLUTensor *rpn_objectness_predictions_with_background_batch= inputs.at(1);
    MLUTensor *anchors= inputs.at(2);
    MLUTensor *out_tmp= outputs.at(0);
    lib::MLUTensorUtil input_tensor_util(inputs[0]);
    MLUPostprocessRpnParam* p = static_cast<MLUPostprocessRpnParam*>(param);
    int batch_size = p->batch_size_, num_anchors = p->num_anchors_, max_nms_out = p->max_nms_out_;
    float iou_thresh_ = p->iou_thresh__, im_height = p->im_height_, im_width = p->im_width_;
    int feature_h = p->feature_h_, feature_w = p->feature_w_, anchor_per_feature = p->anchor_per_feature_;
    float scale_xy = p->scale_xy_, scale_wh = p->scale_wh_, min_nms_score = p->min_nms_score_;
    std::vector<int> tmp_shape(4, 1); //tmp tensor
    tmp_shape[3] = (((num_anchors - 1) / ALIGN_SIZE + 1) * ALIGN_SIZE * 18);
    MLUTensor *tmp_tensor;
    TF_STATUS_CHECK(lib::CreateMLUTensor(&tmp_tensor, MLU_TENSOR, input_tensor_util.dtype(), tmp_shape));
    intmd_tensors_.push_back(tmp_tensor);
    cnmlPluginPostProcessRpnOpParam* params;
    TF_STATUS_CHECK(lib::CreatePostprocessRpnOp(&postprocessRpn_op, params,
    box_trans, score_trans, anchor_trans, tmp_tensor,
    out_tmp,batch_size, num_anchors, max_nms_out, iou_thresh_,
    im_height, im_width, scale_xy, scale_wh, min_nms_score));
    base_ops_.push_back(postprocessRpn_op);
    return Status::OK();
}
Status MLUPostprocessRpn::Compute(const std::vector<void *> &inputs,
const std::vector<void *> &outputs, cnrtQueue_t queue) {
  void *rpn_box_encodings_batch_mlu_ptr = inputs.at(0);
  void *rpn_objectness_predictions_with_background_batch_mlu_ptr = inputs.at(1);
  void *anchors_mlu_ptr = inputs.at(2);
  void *proposal_box_mlu_ptr = outputs.at(0);
  void *tmp_tensor_mlu_ptr;
  size_t tmp_size;
  cnmlGetTensorSize_V2(intmd_tensors_.at(0), &tmp_size); //tmp tensor
  cnrtMalloc(&tmp_tensor_mlu_ptr, tmp_size);
  TF_STATUS_CHECK(lib::ComputePostprocessRpnOp(base_ops_.at(0),
                                               rpn_box_encodings_batch_mlu_ptr,
                                               rpn_objectness_predictions_with_background_batch_mlu_ptr,
                                               anchors_mlu_ptr,
                                               tmp_tensor_mlu_ptr,
                                               proposal_box_mlu_ptr,
                                               queue));
  TF_CNRT_CHECK(cnrtSyncQueue(queue));
  return Status::OK();
}
}  // namespace ops
}  // namespace mlu
}  // namespace stream_executor
```

- 内层实现：lib_ops层
  - 该层主要是对CNML和CNRT等接口的封装

tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc

```
namespace stream_executor {
namespace mlu {
namespace lib {

using namespace std;
tensorflow::Status CreatePostprocessRpnOp(MLUBaseOp** op, cnmlPluginPostProcessRpnOpParam* params,
                                          cnmlTensor* box_trans, cnmlTensor* score_trans,
                                          cnmlTensor* anchor_trans, cnmlTensor* tmp_tensor,
                                          cnmlTensor* out_tmp,
                                          int batch_size, int num_anchors, int max_nms_out, float iou_thresh_,
                                          float im_height, float im_width,
                                          float scale_xy, float scale_wh, float min_nms_score
                                          ){

  cnmlCreatePluginPostProcessRpnOpParam(&params, batch_size, num_anchors, max_nms_out,
  iou_thresh_, im_height, im_width, scale_xy, scale_wh, min_nms_score);
  CNML_RETURN_STATUS(cnmlCreatePluginPostProcessRpnOp(
                                                      op,
                                                      params,
                                                      box_trans,
                                                      score_trans,
                                                      anchor_trans,
                                                      tmp_tensor,
                                                      out_tmp));
}

tensorflow::Status ComputePostprocessRpnOp(MLUBaseOp* op,
                                           void *rpn_box_encodings_batch_mlu_ptr,
                                           void *rpn_objectness_predictions_with_background_batch_mlu_ptr,
                                           void *anchors_mlu_ptr,
                                           void *tmp_tensor_mlu_ptr,
                                           void *proposal_box_mlu_ptr,
                                           MLUCnrtQueue* queue){
  CNML_RETURN_STATUS(cnmlComputePluginPostProcessRpnForward(op,
                                                            rpn_box_encodings_batch_mlu_ptr,
                                                            rpn_objectness_predictions_with_background_batch_mlu_ptr,
                                                            anchors_mlu_ptr,
                                                            tmp_tensor_mlu_ptr,
                                                            proposal_box_mlu_ptr,
                                                            queue));


}
```

- 内层实现
  - 算子创建函数，调用CNRT和CNML的相关函数
  - 数据类型转换
  - kernel参数块创建
  - cnmCreatePluginOp创建plugin算子
  - 销毁kernel参数块

```
cnmlStatus_t cnmlCreatePluginPostProcessRpnOp(
    cnmlBaseOp_t *op_ptr,
    cnmlPluginPostProcessRpnOpParam_t param,
    cnmlTensor_t rpn_box_encodings_batch,
    cnmlTensor_t rpn_objectness_predictions_with_background_batch,
    cnmlTensor_t anchors,
    cnmlTensor_t tmp_tensor,
    cnmlTensor_t proposal_box)
{
    int batch_size = param->batch_size;
    int num_anchors = param->num_anchors;
    int max_nms_out = param->max_nms_out;
    float iou_thresh = param->iou_thresh;
    float im_height = param->im_height;
    float im_width = param->im_width;
    float scale_xy = param->scale_xy;
    float scale_wh = param->scale_wh;
    float min_nms_score = param->min_nms_score;
    int input_num = 4;
    int output_num = 1;
    cnmlTensor_t cnml_input_ptr[4];
    cnml_input_ptr[0] = rpn_box_encodings_batch;
    cnml_input_ptr[1] = rpn_objectness_predictions_with_background_batch;
    cnml_input_ptr[2] = anchors;
    cnml_input_ptr[3] = tmp_tensor;
    cnmlTensor_t cnml_output_ptr[1];
    cnml_output_ptr[0] = proposal_box;
    half in_nms_thresh, in_im_h, in_im_w, in_scale_xy, in_scale_wh, in_min_nms_score;
    cnrtConvertFloatToHalf(&in_nms_thresh, iou_thresh);
    cnrtConvertFloatToHalf(&in_im_h, im_height);
    cnrtConvertFloatToHalf(&in_im_w, im_width);
    cnrtConvertFloatToHalf(&in_scale_xy, scale_xy);
    cnrtConvertFloatToHalf(&in_scale_wh, scale_wh);
    cnrtConvertFloatToHalf(&in_min_nms_score, min_nms_score);
    cnrtKernelParamsBuffer_t params;
    cnrtGetKernelParamsBuffer(&params);
    cnrtKernelParamsBufferMarkInput(params);
    cnrtKernelParamsBufferMarkInput(params);
    cnrtKernelParamsBufferMarkInput(params);
    cnrtKernelParamsBufferMarkOutput(params);
    cnrtKernelParamsBufferAddParam(params, &batch_size, sizeof(int));
    cnrtKernelParamsBufferAddParam(params, &num_anchors, sizeof(int));
    cnrtKernelParamsBufferAddParam(params, &max_nms_out, sizeof(int));
    cnrtKernelParamsBufferAddParam(params, &in_nms_thresh, sizeof(half));
    cnrtKernelParamsBufferAddParam(params, &in_im_h, sizeof(half));
    cnrtKernelParamsBufferAddParam(params, &in_im_w, sizeof(half));
    cnrtKernelParamsBufferAddParam(params, &in_scale_xy, sizeof(half));
    cnrtKernelParamsBufferAddParam(params, &in_scale_wh, sizeof(half));
    cnrtKernelParamsBufferAddParam(params, &in_min_nms_score, sizeof(half));
    cnmlCreatePluginOp(
        op_ptr,
        "POSTPROCESSRPN",
        reinterpret_cast<void **>(&PostprocessRpnKernel),
        params,
        cnml_input_ptr,
        input_num,
        cnml_output_ptr,
        output_num,
        nullptr,
        0);
    cnrtDestroyKernelParamsBuffer(params);
    std::cout << __FILE__ << __LINE__ << std::endl;
    return CNML_STATUS_SUCCESS;
}
```

1

2

3

4

- 内层实现
  - 算子执行函数，调用cnmlComputePluginOpForward完成真正的计算

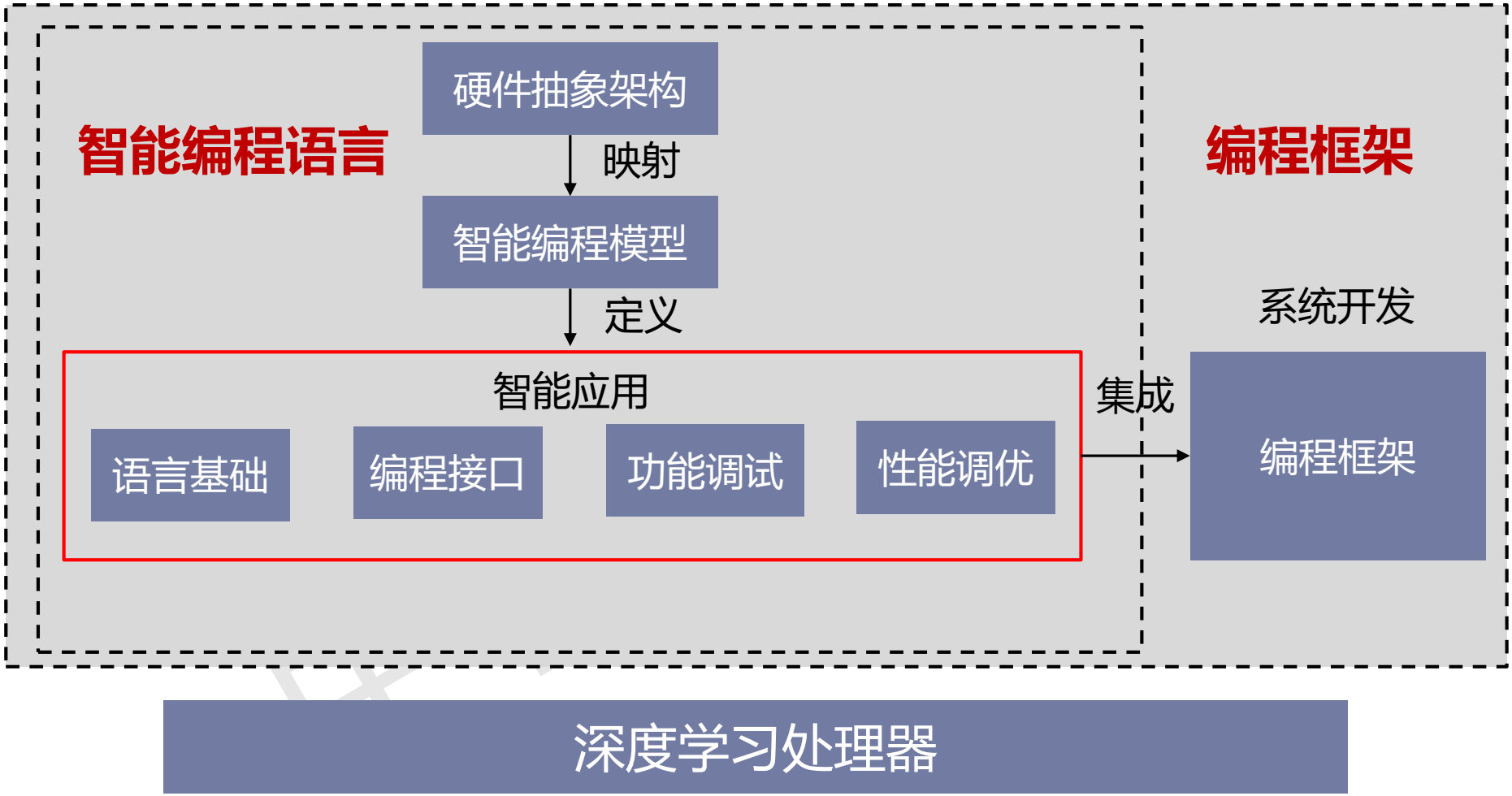
```
cnmlStatus_t cnmlComputePluginPostProcessRpnForward(
    cnmlBaseOp_t op,
    void *rpn_box_encodings_batch_mlu_ptr,
    void *rpn_objectness_predictions_with_background_batch_mlu_ptr,
    void *anchors_mlu_ptr,
    void *tmp_tensor_mlu_ptr,
    void *proposal_box_mlu_ptr,
    cnrtQueue_t queue)
{
    const int input_num = 4;
    const int output_num = 1;
    void *input_mlu_ptrs[input_num];
    void *output_mlu_ptrs[output_num];
    std::cout << __FILE__ << __LINE__ << std::endl;

    input_mlu_ptrs[0] = rpn_box_encodings_batch_mlu_ptr,
    input_mlu_ptrs[1] = rpn_objectness_predictions_with_background_batch_mlu_ptr,
    input_mlu_ptrs[2] = anchors_mlu_ptr,
    input_mlu_ptrs[3] = tmp_tensor_mlu_ptr,
    std::cout << __FILE__ << __LINE__ << std::endl;

    output_mlu_ptrs[0] = proposal_box_mlu_ptr;
    std::cout << __FILE__ << __LINE__ << std::endl;
    cnmlComputePluginOpForward_V4(
        op,
        NULL,
        input_mlu_ptrs,
        input_num,
        NULL,
        output_mlu_ptrs,
        output_num,
        queue,
        NULL);
    std::cout << __FILE__ << __LINE__ << std::endl;
    return CNML_STATUS_SUCCESS;
}
```

优化结果分析

运算器类型	原始的 Faster R-CNN	优化后的 Faster R-CNN
总节点数	11171	679
DLP 处理节点数	3590	370
CPU 处理节点数	7581	309
融合总段数	295	6

- 由于使用智能编程语言实现自定义算子，网络总节点数大幅度下降，大量细碎的在CPU上运行的节点被整合到规整的自定义算子中，在DLP上运行的计算比例大幅度增加，充分利用了DLP的计算资源
- 融合段数大幅度减少，从295段减少到只有6段，使得DLP可以执行更优的策略，也减少了DLP和CPU间的数据拷贝开销，**整体性能提升了近8倍**。

小结



通过智能编程语言+编程框架提供高开发效率、高性能和高可移植的编程方法

谢谢大家！