CANN 社区版
8.0.0.alpha002

Ascend C 自定义算子开发指南

文档版本	01
发布日期	2024-12-25

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

安全声明

产品生命周期政策

华为公司对产品生命周期的规定以“产品生命周期终止政策”为准，该政策的详细内容请参见如下网址：
https://support.huawei.com/ecolumnsweb/zh/warranty-policy

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

华为初始证书权责说明

华为公司对随设备出厂的初始数字证书，发布了“华为设备初始数字证书权责说明”，该说明的详细内容请参见如下网址：
https://support.huawei.com/enterprise/zh/bulletins-service/ENEWS2000015766

华为企业业务最终用户许可协议(EULA)

本最终用户许可协议是最终用户（个人、公司或其他任何实体）与华为公司就华为软件的使用所缔结的协议。最终用户对华为软件的使用受本协议约束，该协议的详细内容请参见如下网址：
https://e.huawei.com/cn/about/eula

产品资料生命周期策略

华为公司针对随产品版本发布的售后客户资料（产品资料），发布了“产品资料生命周期策略”，该策略的详细内容请参见如下网址：
https://support.huawei.com/enterprise/zh/bulletins-website/ENEWS2000017760

目 录

1 Ascend C 简介

概述

Ascend C是CANN针对算子开发场景推出的编程语言，原生支持C和C++标准规范，兼具开发效率和运行性能。基于Ascend C编写的算子程序，通过编译器编译和运行时调度，运行在昇腾AI处理器上。使用Ascend C，开发者可以基于昇腾AI硬件，高效的实现自定义的创新算法。

使用Ascend C进行自定义算子开发的突出优势有：

- 遵循C/C++编程规范，匹配用户开发习惯

- 自动并行调度，获得最优执行性能
- 结构化核函数编程，简化算子开发逻辑
- CPU/NPU孪生调试，提升算子调试效率

您可以通过**Ascend C主页**了解更详细的内容。

算子开发成长地图

使用须知

当前Ascend C支持的产品型号为：

- **Atlas 推理系列产品**
- **Atlas 训练系列产品**
- **Atlas A2训练系列产品/Atlas 800I A2推理产品**
- **Atlas 200/500 A2推理产品**

2 环境准备

进行环境准备前，你需要了解如下基本概念，以便更好的理解后续操作。

- 开发环境：指编译开发代码的环境。
- 运行环境：指运行算子、推理程序、训练程序等的环境。运行环境必须带昇腾AI处理器。
- 开发环境与运行环境合设场景：指带昇腾AI处理器的机器既作为开发环境又作为运行环境。此种场景下，代码开发与代码运行在同一台机器上。
- 开发环境与运行环境分设场景：开发环境和运行环境不在同一台机器上，开发者使用带有昇腾AI处理器的机器作为运行环境；使用其他独立机器进行代码开发与编译，作为开发环境。

进行Ascend C算子开发之前，需要安装**驱动固件**和**CANN软件包**，完成开发环境和运行环境的准备。

驱动固件的安装步骤可参见《CANN软件安装指南》的“安装NPU驱动固件”章节，本节不再给出安装示例。

本节以开发环境和运行环境合设、操作系统架构X86_64、root用户操作为例，介绍**CANN软件包**安装的步骤，其他场景的安装步骤类似，详细的可参见《CANN软件安装指南》。

步骤1 安装相关依赖。

- Debian系列（Ubuntu、Debian、UOS20、UOS20 SP1）：
  ```
  apt-get install -y gcc make net-tools python3 python3-dev python3-pip
  ```
- openEuler系列（openEuler、CentOS、Kylin、BCLinux、BC-Linux-for-Euler、UOS201050e、UOS20 1020e、UOSV20、AntOS、CTyunOS、CULinux、Tlinux）：
  ```
  yum install -y gcc make net-tools python3 python3-devel python3-pip
  ```

步骤2 检查系统是否安装满足版本要求的Python开发环境（支持Python3.7.*x*至3.11.4版本）。

```
python3 --version
pip3 --version
```

如果返回信息满足Python版本要求，则直接进入下一步，若不满足请参考《CANN软件安装指南》中的“编译安装Python3.7.5”章节。

步骤3 执行如下命令安装所需的Python第三方库。

```
pip3 install attrs cython numpy==1.24.0 decorator sympy cffi pyyaml pathlib2 psutil protobuf==3.20 scipy requests absl-py
```


当前命令会安装最新或指定版本的依赖，若安装报错或更多版本要求请参考《**CANN软件安装指南**》的“依赖列表”。

步骤4 安装CANN开发套件包。

1. 将CANN开发套件包上传至安装环境的任意目录，执行如下命令增加对软件包的可执行权限。
   ```
   chmod +x Ascend-cann-toolkit_8.0.RC3.alpha001_linux-x86_64.run
   ```
2. 执行如下命令校验软件包的一致性和完整性。
   ```
   ./Ascend-cann-toolkit_8.0.RC3.alpha001_linux-x86_64.run --check
   ```
3. 执行如下命令安装CANN开发套件包。
   ```
   ./Ascend-cann-toolkit_8.0.RC3.alpha001_linux-x86_64.run --install
   ```
   用户需签署**华为企业业务最终用户许可协议（EULA）**后进入安装流程，根据回显页面输入**y或Y**接受协议，安装完成后，若显示如下信息，则说明软件安装成功。
   ```
   [INFO] Ascend-cann-toolkit install success
   ```
4. 配置CANN环境变量。
   ```
   source /usr/local/Ascend/ascend-toolkit/set_env.sh
   ```

----结束

说明

安装第三方依赖时，gcc/g++的版本应满足《**CANN软件安装指南**》中所列出的要求。

3 快速入门

3.1 HelloWorld

下面是一个简单的Ascend C的"Hello World"样例，展示了一个Ascend C核函数（设备侧实现的入口函数）的基本写法，及其如何被调用的流程。

- 包含核函数的Kernel实现文件hello_world.cpp代码如下：核函数hello_world的核心逻辑为打印"Hello World"字符串。hello_world_do封装了核函数的调用程序，通过<<<>>>内核调用符对核函数进行调用。

```
#include "kernel_operator.h"
extern "C" __global__ __aicore__ void hello_world()
{
    AscendC::printf("Hello World!!!\n");
}

void hello_world_do(uint32_t blockDim, void* stream)
{
    hello_world<<<blockDim, nullptr, stream>>>();
}
```

- 调用核函数的应用程序main.cpp代码如下（您可以通过代码注释了解其主要的流程）：

```
#include "acl/acl.h"
extern void hello_world_do(uint32_t coreDim, void* stream);

int32_t main(int argc, char const *argv[])
{
    // AscendCL初始化
    aclInit(nullptr);
    // 运行管理资源申请
    int32_t deviceId = 0;
    aclrtSetDevice(deviceId);
    aclrtStream stream = nullptr;
    aclrtCreateStream(&stream);

    // 设置参与运算的核数为8
    constexpr uint32_t blockDim = 8;
    // 用内核调用符<<<>>>调用核函数，hello_world_do中封装了<<<>>>调用
    hello_world_do(blockDim, stream);
    aclrtSynchronizeStream(stream);
```



```
    // 资源释放和AscendCL去初始化
    aclrtDestroyStream(stream);
    aclrtResetDevice(deviceId);
    aclFinalize();
    return 0;
}
```

通过如下的代码工程对上述文件进行组织，您可以通过**LINK**获取样例工程，并参考README完成CMakeLists中的AI处理器的型号、软件包安装路径配置。

```
|-- CMakeLists.txt                    // CMake编译配置文件
|-- hello_world.cpp                   // Kernel实现
|-- main.cpp                          // 调用核函数的主程序
|-- run.sh                            // 一键式编译运行脚本
```

通过执行如下脚本实现样例程序的编译和运行：

```
bash run.sh -v <soc_version>
```

说明

AI处理器的型号<*soc_version*>请通过如下方式获取：

- 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

本样例共调度八个核，分别打印了每个核的核号和"Hello World"信息。

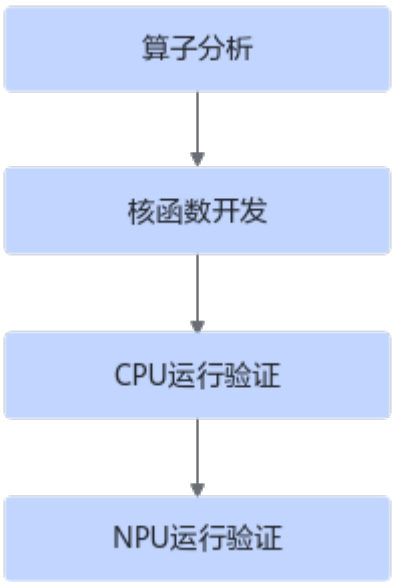
3.2 基于 Kernel 直调工程的算子开发

本入门教程，将会引导你完成以下任务，体验Ascend C算子开发的基本流程。

1. 使用Ascend C完成Add算子**核函数开发**；
2. 使用ICPU_RUN_KF CPU调测宏完成算子核函数**CPU侧运行验证**；
3. 使用<<<>>>内核调用符完成算子核函数**NPU侧运行验证**。

在正式的开发之前，还需要先完成**环境准备**和**算子分析**工作，开发Ascend C算子的基本流程如下图所示：

图 3-1 开发 Ascend C 算子的基本流程



说明

- 请点击**矢量算子样例**获取样例代码。
- 使用本教程只需要您具有一定的C/C++基础，在此基础上，如果您已经对Ascend C编程模型有一定的了解，您可以在实际操作的过程中加深对理论的理解；如果您还没有开始了解Ascend C编程模型，也无需担心，您可以先尝试跑通教程中的样例，参考教程最后的指引进行进一步的学习。

环境准备

- CANN软件安装

  开发算子前，需要先准备好开发环境和运行环境，开发环境和运行环境的介绍和具体的安装步骤可参见《**CANN软件安装指南**》。

- 环境变量配置

  安装CANN软件后，使用CANN运行用户进行编译、运行时，需要以CANN运行用户登录环境，执行**source ${install_path}/set_env.sh**命令设置环境变量。其中${install_path}为CANN软件的安装目录，例如：/usr/local/Ascend/ascend-toolkit。

算子分析

主要分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。

步骤1 明确算子的数学表达式及计算逻辑。

Add算子的数学表达式为：

$$\vec{z} = \vec{x} + \vec{y}$$

计算逻辑是：Ascend C提供的矢量计算接口的操作元素都为LocalTensor，输入数据需要先搬运进AI Core的内部存储Local Memory，然后使用计算接口完成两个输入参数相加，得到最终结果，再搬出到外部存储Global Memory上。

步骤2 明确输入和输出。

- Add算子有两个输入：x与y，输出为z。
- 本样例中算子的输入支持的数据类型为half（float16），算子输出的数据类型与输入数据类型相同。
- 算子输入支持shape（8，2048），输出shape与输入shape相同。
- 算子输入支持的format为：ND。

步骤3 确定核函数名称和参数。

- 本样例中核函数命名为add_custom。
- 根据对算子输入输出的分析，确定核函数有3个参数x，y，z；x，y为输入在Global Memory上的内存地址，z为输出在Global Memory上的内存地址。

步骤4 确定算子实现所需接口。

- 实现涉及外部存储和内部存储间的数据搬运，查看Ascend C API参考中的数据搬移接口，需要使用DataCopy来实现数据搬移。
- 本样例只涉及矢量计算的加法操作，查看Ascend C API参考中的矢量计算接口矢量计算，初步分析可使用双目指令Add接口Add实现x+y。

- 计算中使用到的Tensor数据结构，使用AllocTensor、FreeTensor进行申请和释放。
- 并行流水任务之间使用Queue队列完成通信和同步，会使用到EnQue、DeQue等接口。

----结束

通过以上分析，得到Ascend C Add算子的设计规格如下：

表 3-1 Ascend C Add 算子设计规格

算子类型（OpType）	AddCustom			
算子输入	**name**	**shape**	**data type**	**format**
	x	(8, 2048)	half	ND
	y	(8, 2048)	half	ND
算子输出	z	(8, 2048)	half	ND
核函数名称	add_custom			
使用的主要接口	DataCopy：数据搬移接口			
	Add：矢量双目指令接口			
	AllocTensor、FreeTensor：内存管理接口			
	EnQue、DeQue接口：Queue队列管理接口			
算子实现文件名称	add_custom.cpp			

核函数开发

完成环境准备和初步的算子分析后，即可开始Ascend C核函数的开发。开发之前请先从**矢量算子样例**获取样例代码目录，以下样例代码在add_custom.cpp中实现。

本样例中使用多核并行计算，即把数据进行分片，分配到多个核上进行处理。Ascend C核函数是在一个核上的处理函数，所以只处理部分数据。分配方案是：数据整体长度TOTAL_LENGTH为8 * 2048，平均分配到8个核上运行，每个核上处理的数据大小BLOCK_LENGTH为2048。下文的核函数，只关注长度为BLOCK_LENGTH的数据应该如何处理。

步骤1 首先，您需要根据**核函数定义**中介绍的规则进行**核函数的定义，**并在核函数中调用算子类的Init和Process函数。

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

- 使用__global__函数类型限定符来标识它是一个核函数，可以被<<<>>>调用；使用__aicore__函数类型限定符来标识该核函数在设备端AI Core上执行。指针入参

变量需要增加变量类型限定符__gm__，表明该指针变量指向Global Memory上某处内存地址。为了统一表达，使用GM_ADDR宏来修饰入参，GM_ADDR宏定义如下：

```
#define GM_ADDR __gm__ uint8_t*
```

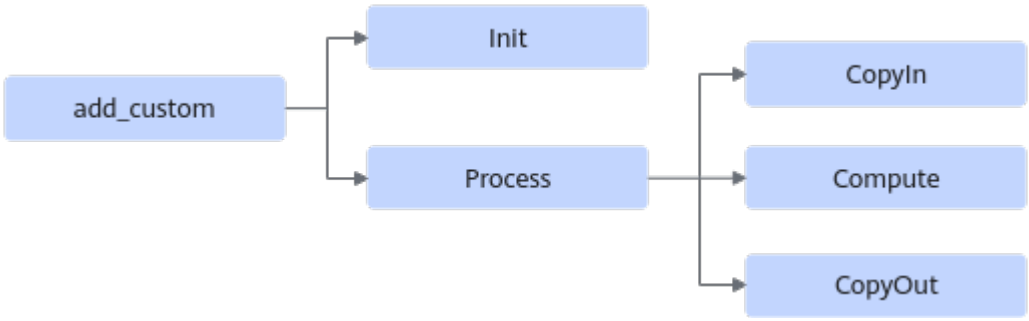
- 算子类的Init函数，完成内存初始化相关工作，Process函数完成算子实现的核心逻辑。

步骤2 然后**根据矢量编程范式实现算子类**，本样例中定义KernelAdd算子类，其具体成员如下：

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd(){}
    // 初始化函数，完成内存初始化相关操作
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z){}
    // 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操作
    __aicore__ inline void Process(){}

private:
    // 搬入函数，完成CopyIn阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyIn(int32_t progress){}
    // 计算函数，完成Compute阶段的处理，被核心Process函数调用
    __aicore__ inline void Compute(int32_t progress){}
    // 搬出函数，完成CopyOut阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyOut(int32_t progress){}

private:
    AscendC::TPipe pipe;  //Pipe内存管理对象
    AscendC::TQue<AscendC::QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;  //输入数据Queue队列管理对象，QuePosition为VECIN
    AscendC::TQue<AscendC::QuePosition::VECOUT, BUFFER_NUM> outQueueZ;  //输出数据Queue队列管理对象，QuePosition为VECOUT
    AscendC::GlobalTensor<half> xGm;  //管理输入输出Global Memory内存地址的对象，其中xGm, yGm为输入，zGm为输出
    AscendC::GlobalTensor<half> yGm;
    AscendC::GlobalTensor<half> zGm;
};
```

内部函数的调用关系示意图如下：

图 3-2 核函数调用关系图



由此可见除了Init函数完成初始化外，Process中完成了对**流水任务**：“搬入、计算、搬出”的调用，开发者可以重点关注三个流水任务的实现。

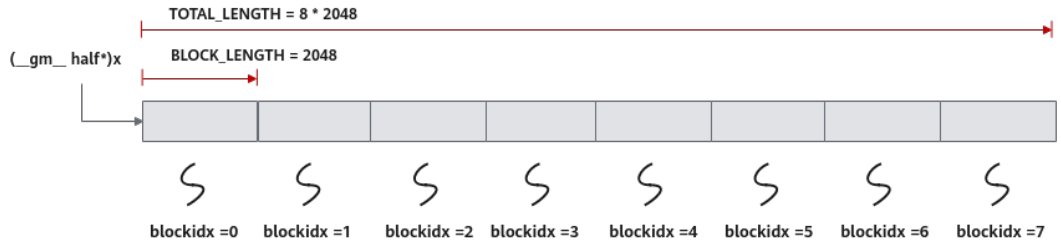
步骤3 初始化函数**Init**主要完成以下内容：设置输入输出Global Tensor的Global Memory内存地址，通过Pipe内存管理对象为输入输出Queue分配内存。

上文我们介绍到，本样例将数据切分成8块，平均分配到8个核上运行，每个核上处理的数据大小BLOCK_LENGTH为2048。那么我们是如何实现这种切分的呢？

每个核上处理的数据地址需要在起始地址上增加GetBlockIdx() * BLOCK_LENGTH（每个block处理的数据长度）的偏移来获取。这样也就实现了多核并行计算的数据切分。

以输入x为例，x + BLOCK_LENGTH * GetBlockIdx()即为单核处理程序中x在Global Memory上的内存偏移地址，获取偏移地址后，使用GlobalTensor类的SetGlobalBuffer接口设定该核上Global Memory的起始地址以及长度。具体示意图如下。

图 3-3 多核并行处理示意图

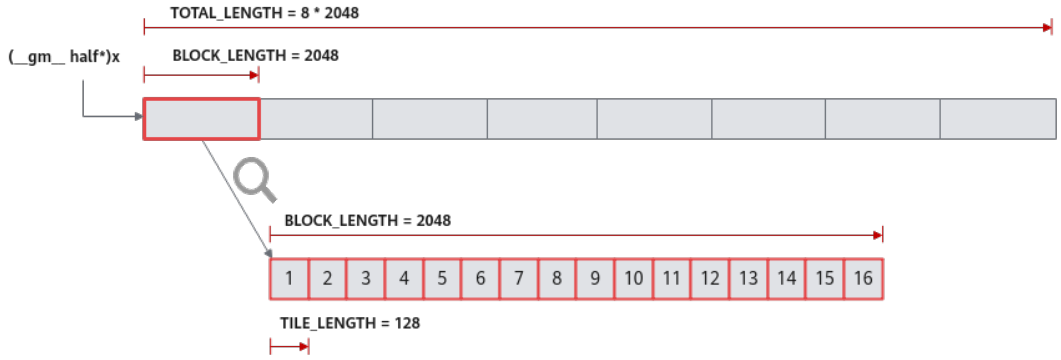


上面已经实现了多核数据的切分，那么单核上的处理数据如何进行切分？

对于单核上的处理数据，可以进行数据切块（Tiling），在本示例中，仅作为参考，将数据切分成8块（并不意味着8块就是性能最优）。切分后的每个数据块再次切分成2块，即可开启**double buffer**，实现流水线之间的并行。

这样单核上的数据（2048个数）被切分成16块，每块TILE_LENGTH（128）个数据。Pipe为inQueueX分配了两块大小为TILE_LENGTH * sizeof(half)个字节的内存块，每个内存块能容纳TILE_LENGTH（128）个half类型数据。数据切分示意图如下。

图 3-4 单核数据切分示意图



具体的初始化函数代码如下：

```
#include "kernel_operator.h"

constexpr int32_t TOTAL_LENGTH = 8 * 2048;                            // total length of data
constexpr int32_t USE_CORE_NUM = 8;                                   // num of core used
constexpr int32_t BLOCK_LENGTH = TOTAL_LENGTH / USE_CORE_NUM;         // length computed of each core
constexpr int32_t TILE_NUM = 8;                                       // split data into 8 tiles for each core
constexpr int32_t BUFFER_NUM = 2;                                     // tensor num for each queue
constexpr int32_t TILE_LENGTH = BLOCK_LENGTH / TILE_NUM / BUFFER_NUM; // separate to 2 parts, due to double buffer

__aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    // get start index for current core, core parallel
```



```
    xGm.SetGlobalBuffer((__gm__ half*)x + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
    yGm.SetGlobalBuffer((__gm__ half*)y + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
    zGm.SetGlobalBuffer((__gm__ half*)z + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
    // pipe alloc memory to queue, the unit is Bytes
    pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(inQueueY, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(outQueueZ, BUFFER_NUM, TILE_LENGTH * sizeof(half));
}
```

步骤4 基于矢量编程范式，将核函数的实现分为3个基本任务：CopyIn，Compute，CopyOut。**Process**函数中通过如下方式调用这三个函数。

```
__aicore__ inline void Process()
{
    // loop count need to be doubled, due to double buffer
    constexpr int32_t loopCount = TILE_NUM * BUFFER_NUM;
    // tiling strategy, pipeline parallel
    for (int32_t i = 0; i < loopCount; i++) {
        CopyIn(i);
        Compute(i);
        CopyOut(i);
    }
}
```

1. CopyIn函数实现。
   a. 使用DataCopy接口将GlobalTensor数据拷贝到LocalTensor。
   b. 使用EnQue将LocalTensor放入VecIn的Queue中。

```
__aicore__ inline void CopyIn(int32_t progress)
{
    // alloc tensor from queue memory
    AscendC::LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
    AscendC::LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
    // copy progress_th tile from global tensor to local tensor
    AscendC::DataCopy(xLocal, xGm[progress * TILE_LENGTH], TILE_LENGTH);
    AscendC::DataCopy(yLocal, yGm[progress * TILE_LENGTH], TILE_LENGTH);
    // enque input tensors to VECIN queue
    inQueueX.EnQue(xLocal);
    inQueueY.EnQue(yLocal);
}
```

2. Compute函数实现。
   a. 使用DeQue从VecIn中取出LocalTensor。
   b. 使用Ascend C接口Add完成矢量计算。
   c. 使用EnQue将计算结果LocalTensor放入到VecOut的Queue中。
   d. 使用FreeTensor将释放不再使用的LocalTensor。

```
__aicore__ inline void Compute(int32_t progress)
{
    // deque input tensors from VECIN queue
    AscendC::LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    AscendC::LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    AscendC::LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    // call Add instr for computation
    AscendC::Add(zLocal, xLocal, yLocal, TILE_LENGTH);
    // enque the output tensor to VECOUT queue
    outQueueZ.EnQue<half>(zLocal);
    // free input tensors for reuse
    inQueueX.FreeTensor(xLocal);
    inQueueY.FreeTensor(yLocal);
}
```

3. CopyOut函数实现。
   a. 使用DeQue接口从VecOut的Queue中取出LocalTensor。
   b. 使用DataCopy接口将LocalTensor拷贝到GlobalTensor上。

c. 使用FreeTensor将不再使用的LocalTensor进行回收。

```
__aicore__ inline void CopyOut(int32_t progress)
{
    // deque output tensor from VECOUT queue
    AscendC::LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
    // copy progress_th tile from local tensor to global tensor
    AscendC::DataCopy(zGm[progress * TILE_LENGTH], zLocal, TILE_LENGTH);
    // free output tensor for reuse
    outQueueZ.FreeTensor(zLocal);
}
```

----结束

核函数运行验证

异构计算架构中，NPU（kernel侧）与CPU（host侧）是协同工作的，完成了kernel侧核函数开发后，即可编写host侧的核函数调用程序，实现从host侧的APP程序调用算子，执行计算过程。

除了上文核函数实现文件add_custom.cpp外，核函数的调用与验证还需要准备以下文件：

- 调用算子的应用程序：main.cpp。
- 输入数据和真值数据生成脚本文件：gen_data.py。
- 验证输出数据和真值数据是否一致的验证脚本：verify_result.py。
- 编译cpu侧或npu侧运行的算子的编译工程文件：CMakeLists.txt。
- 编译运行算子的脚本：run.sh。

本文仅介绍调用算子的应用程序的编写，该应用程序在main.cpp中体现，其他内容您可以在**矢量算子样例**中直接获取。

步骤1 host侧应用程序框架的编写。

内置宏ASCENDC_CPU_DEBUG是区分运行CPU模式或NPU模式逻辑的标志，在同一个main函数中通过对ASCENDC_CPU_DEBUG宏定义的判断来区分CPU和NPU侧的运行程序。

```
#include "data_utils.h"
#ifndef ASCENDC_CPU_DEBUG
#include "acl/acl.h"
extern void add_custom_do(uint32_t coreDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y, uint8_t* z);
#else
#include "tikicpulib.h"
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z);
#endif

int32_t main(int32_t argc, char* argv[])
{
    size_t inputByteSize = 8 * 2048 * sizeof(uint16_t);  // uint16_t represent half
    size_t outputByteSize = 8 * 2048 * sizeof(uint16_t);  // uint16_t represent half
    uint32_t blockDim = 8;

#ifdef ASCENDC_CPU_DEBUG
    // 用于CPU调试的调用程序

#else
    // NPU侧运行算子的调用程序

#endif
    return 0;
}
```

步骤2 编写用于CPU调试的调用程序。

完成算子核函数**CPU侧运行验证的步骤如下：**

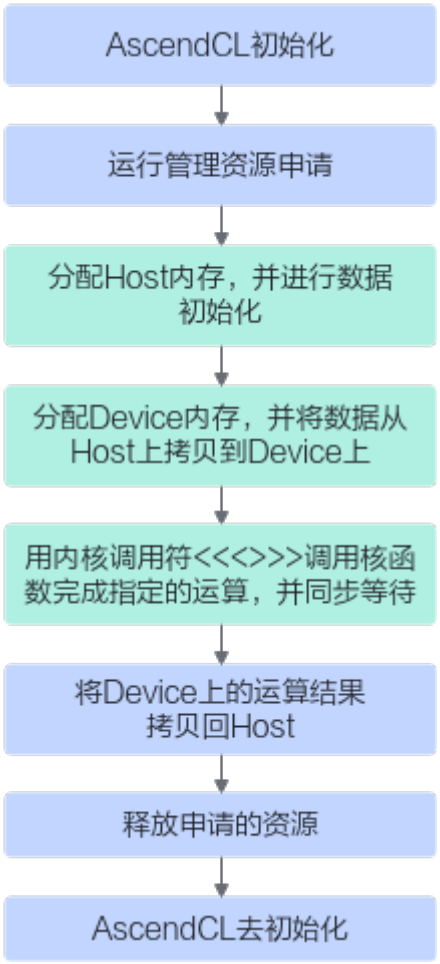
图 3-5 CPU 侧运行验证步骤


```
// 使用GmAlloc分配共享内存，并进行数据初始化
uint8_t* x = (uint8_t*)AscendC::GmAlloc(inputByteSize);
uint8_t* y = (uint8_t*)AscendC::GmAlloc(inputByteSize);
uint8_t* z = (uint8_t*)AscendC::GmAlloc(outputByteSize);

ReadFile("./input/input_x.bin", inputByteSize, x, inputByteSize);
ReadFile("./input/input_y.bin", inputByteSize, y, inputByteSize);
// 调用ICPU_RUN_KF调测宏，完成核函数CPU侧的调用
AscendC::SetKernelMode(KernelMode::AIV_MODE);
ICPU_RUN_KF(add_custom, blockDim, x, y, z); // use this macro for cpu debug
// 输出数据写出
WriteFile("./output/output_z.bin", z, outputByteSize);
// 调用GmFree释放申请的资源
AscendC::GmFree((void *)x);
AscendC::GmFree((void *)y);
AscendC::GmFree((void *)z);
```

步骤3 编写NPU侧运行算子的调用程序。

完成算子核函数**NPU侧运行验证的步骤如下：**

图 3-6 NPU 侧运行验证步骤




```
    // AscendCL初始化
    CHECK_ACL(aclInit(nullptr));
    // 运行管理资源申请
    int32_t deviceId = 0;
    CHECK_ACL(aclrtSetDevice(deviceId));
    aclrtStream stream = nullptr;
    CHECK_ACL(aclrtCreateStream(&stream));
    // 分配Host内存
    uint8_t *xHost, *yHost, *zHost;
    uint8_t *xDevice, *yDevice, *zDevice;
    CHECK_ACL(aclrtMallocHost((void**)(&xHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&yHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&zHost), outputByteSize));
    // 分配Device内存
    CHECK_ACL(aclrtMalloc((void**)&xDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&yDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&zDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // Host内存初始化
    ReadFile("./input/input_x.bin", inputByteSize, xHost, inputByteSize);
    ReadFile("./input/input_y.bin", inputByteSize, yHost, inputByteSize);
    CHECK_ACL(aclrtMemcpy(xDevice, inputByteSize, xHost, inputByteSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    CHECK_ACL(aclrtMemcpy(yDevice, inputByteSize, yHost, inputByteSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    // 用内核调用符<<<>>>调用核函数完成指定的运算,add_custom_do中封装了<<<>>>调用
    add_custom_do(blockDim, nullptr, stream, xDevice, yDevice, zDevice);
    CHECK_ACL(aclrtSynchronizeStream(stream));
    // 将Device上的运算结果拷贝回Host
    CHECK_ACL(aclrtMemcpy(zHost, outputByteSize, zDevice, outputByteSize,
```



```
ACL_MEMCPY_DEVICE_TO_HOST));
    WriteFile("./output/output_z.bin", zHost, outputByteSize);
    // 释放申请的资源
    CHECK_ACL(aclrtFree(xDevice));
    CHECK_ACL(aclrtFree(yDevice));
    CHECK_ACL(aclrtFree(zDevice));
    CHECK_ACL(aclrtFreeHost(xHost));
    CHECK_ACL(aclrtFreeHost(yHost));
    CHECK_ACL(aclrtFreeHost(zHost));
    // AscendCL去初始化
    CHECK_ACL(aclrtDestroyStream(stream));
    CHECK_ACL(aclrtResetDevice(deviceId));
    CHECK_ACL(aclFinalize());
```

步骤4 执行一键式编译运行脚本，编译和运行应用程序。

脚本执行方式如下，***<soc_version>***表示算子运行的AI处理器型号，<run_mode>表示算子以cpu模式或npu模式运行。

```
bash run.sh -r <run_mode> -v <soc_version>
```

1. 执行脚本前需要配置环境变量ASCEND_INSTALL_PATH，配置为CANN软件的安装路径，示例如下，请根据实际安装路径进行修改：
   ```
   export ASCEND_INSTALL_PATH=$HOME/Ascend/ascend-toolkit/latest
   ```
2. 完成CPU和NPU侧运行验证。
   - CPU模式下执行如下命令，命令中的***<soc_version>***请替换为实际的AI处理器型号。
     ```
     bash run.sh -r cpu -v <soc_version>
     ```
     运行结果如下，当前使用numpy接口计算了输出数据和真值数据的绝对误差和相对误差，误差在容忍偏差范围内，视为精度符合要求，输出"test pass"字样。
     ```
     test pass
     ```
   - NPU模式下执行如下命令，命令中的***<soc_version>***请替换为实际的AI处理器型号。
     ```
     bash run.sh -r npu -v <soc_version>
     ```
     运行结果如下，当前使用numpy接口计算了输出数据和真值数据的绝对误差和相对误差，误差在容忍偏差范围内，视为精度符合要求，输出"test pass"字样。
     ```
     test pass
     ```

     说明

     AI处理器的型号<*soc_version*>请通过如下方式获取：
     - 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

     该样例支持以下型号：
     - **Atlas 推理系列产品**
     - **Atlas 训练系列产品**
     - **Atlas A2训练系列产品/Atlas 800I A2推理产品**

----结束

接下来的引导

如果您对教程中的多核并行、流水编程等概念不了解，导致阅读过程有些吃力，可以参考**5 编程模型**学习基本概念，再来回顾本教程；如果您已经了解相关概念，并跑通了该样例，您可以参考**7.2 矢量编程**了解Ascend C矢量编程中的更多细节。

3.3 基于自定义算子工程的算子开发

本节以一个简单算子为例，带您体验从算子工程创建、代码编写、编译部署到运行验证的开发全流程，让您对算子开发工程有个宏观的认识，此处我们以输入是动态shape的Add算子实现为例，为了与内置Add算子区分，定义算子类型为AddCustom。

工程创建

CANN软件包中提供了工程创建工具msOpGen，开发者可以输入算子原型定义文件生成Ascend C算子开发工程。

步骤1 编写AddCustom算子的原型定义json文件。

假设AddCustom算子的原型定义文件命名为add_custom.json，存储路径为: $HOME/sample，文件内容如下：

```
[
  {
    "op": "AddCustom",
    "input_desc": [
      {
        "name": "x",
        "param_type": "required",
        "format": [
          "ND"
        ],
        "type": [
          "fp16"
        ]
      },
      {
        "name": "y",
        "param_type": "required",
        "format": [
          "ND"
        ],
        "type": [
          "fp16"
        ]
      }
    ],
    "output_desc": [
      {
        "name": "z",
        "param_type": "required",
        "format": [
          "ND"
        ],
        "type": [
          "fp16"
        ]
      }
    ]
  }
]
```

步骤2 使用msOpGen工具生成AddCustom算子的开发工程。

```
${INSTALL_DIR}/python/site-packages/bin/msopgen gen -i $HOME/sample/add_custom.json -c ai_core-<soc_version> -lan cpp -out   $HOME/sample/AddCustom
```

- **${INSTALL_DIR}**为CANN软件安装后文件存储路径，请根据实际环境进行替换。
- -i：算子原型定义文件add_custom.json所在路径。

- -c：ai_core-<*soc_version*>代表算子在AI Core上执行，<*soc_version*>为昇腾AI处理器的型号。

  说明

  AI处理器的型号<*soc_version*>请通过如下方式获取：

  - 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

  基于同系列的AI处理器型号创建的算子工程，其基础功能（基于该工程进行算子开发、编译和部署）通用。

- -lan： 参数cpp代表算子基于Ascend C编程框架，使用C++编程语言开发。

步骤3 命令执行完后，会在$HOME/sample目录下生成算子工程目录AddCustom，工程中包含算子实现的模板文件，编译脚本等，如下所示：

```
AddCustom
├── build.sh         // 编译入口脚本
├── cmake
│   ├── config.cmake
│   ├── util         // 算子工程编译所需脚本及公共编译文件存放目录
├── CMakeLists.txt   // 算子工程的CMakeLists.txt
├── CMakePresets.json // 编译配置项
├── framework        // 算子插件实现文件目录，单算子模型文件的生成不依赖算子适配插件，无需关注
├── op_host                  // host侧实现文件
│   ├── add_custom_tiling.h    // 算子tiling定义文件
│   ├── add_custom.cpp         // 算子原型注册、shape推导、信息库、tiling实现等内容文件
│   ├── CMakeLists.txt
├── op_kernel                // kernel侧实现文件
│   ├── CMakeLists.txt
│   ├── add_custom.cpp       // 算子核函数实现文件
├── scripts                  // 自定义算子工程打包相关脚本所在目录
```

说明

上述目录结构中的粗体文件为后续算子开发过程中需要修改的文件，其他文件无需修改。

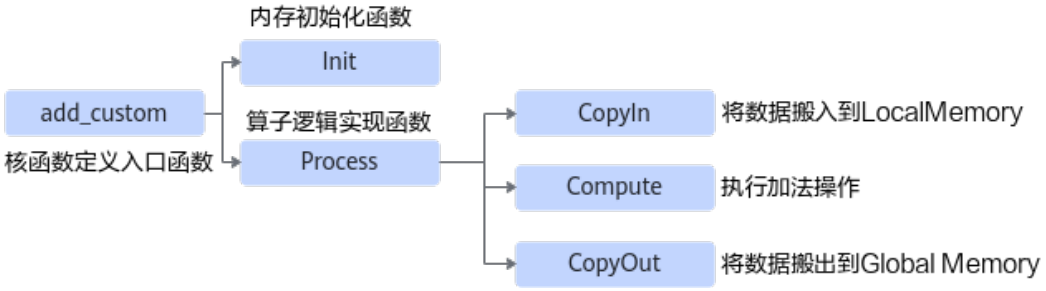
----结束

算子核函数实现

在工程存储目录的“AddCustom/op_kernel/add_custom.cpp”文件中实现算子的核函数，完整的样例代码您可以在**add_custom.cpp**中查看，下面介绍关键实现代码。

算子核函数实现代码的内部调用关系示意图如下：

图 3-7 核函数调用关系图



由此可见除了Init函数完成初始化外，Process中完成了对**流水任务：**“搬入、计算、搬出”的调用，开发者可以重点关注三个流水任务的实现。

步骤1 首先，进行**核函数的定义，**并在核函数中调用算子类的Init和Process函数。

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR
workspace, GM_ADDR tiling)
{
    // 获取Host侧传入的Tiling参数
    GET_TILING_DATA(tiling_data, tiling);
    // 初始化算子类
    KernelAdd op;
    // 算子类的初始化函数，完成内存初始化相关工作
    op.Init(x, y, z, tiling_data.totalLength, tiling_data.tileNum);
    // 完成算子实现的核心逻辑
    op.Process();
}
```

步骤2 定义KernelAdd算子类，其具体成员及成员函数实现如下。

```
#include "kernel_operator.h"
constexpr int32_t BUFFER_NUM = 2;
class KernelAdd {
public:
    __aicore__ inline KernelAdd() {}
    // 初始化函数，完成内存初始化相关操作
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z, uint32_t totalLength, uint32_t
tileNum)
    {
        // 使用获取到的TilingData计算得到singleCoreSize(每个核上总计算数据大小)、tileNum（每个核上分块个
数）、singleTileLength（每个分块大小）等变量
        this->blockLength = totalLength / AscendC::GetBlockNum();
        this->tileNum = tileNum;
        this->tileLength = this->blockLength / tileNum / BUFFER_NUM;

        // 获取当前核的起始索引
        xGm.SetGlobalBuffer((__gm__ DTYPE_X*)x + this->blockLength * AscendC::GetBlockIdx(), this-
>blockLength);
        yGm.SetGlobalBuffer((__gm__ DTYPE_Y*)y + this->blockLength * AscendC::GetBlockIdx(), this-
>blockLength);
        zGm.SetGlobalBuffer((__gm__ DTYPE_Z*)z + this->blockLength * AscendC::GetBlockIdx(), this-
>blockLength);
        // 通过Pipe内存管理对象为输入输出Queue分配内存
        pipe.InitBuffer(inQueueX, BUFFER_NUM, this->tileLength * sizeof(DTYPE_X));
        pipe.InitBuffer(inQueueY, BUFFER_NUM, this->tileLength * sizeof(DTYPE_Y));
        pipe.InitBuffer(outQueueZ, BUFFER_NUM, this->tileLength * sizeof(DTYPE_Z));
    }
    // 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操
作
    __aicore__ inline void Process()
    {
        int32_t loopCount = this->tileNum * BUFFER_NUM;
        for (int32_t i = 0; i < loopCount; i++) {
            CopyIn(i);
            Compute(i);
            CopyOut(i);
        }
    }


private:
    // 搬入函数，完成CopyIn阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyIn(int32_t progress)
    {
        // 从Queue中分配输入Tensor
        AscendC::LocalTensor<DTYPE_X> xLocal = inQueueX.AllocTensor<DTYPE_X>();
        AscendC::LocalTensor<DTYPE_Y> yLocal = inQueueY.AllocTensor<DTYPE_Y>();
         // 将GlobalTensor数据拷贝到LocalTensor
        AscendC::DataCopy(xLocal, xGm[progress * this->tileLength], this->tileLength);
        AscendC::DataCopy(yLocal, yGm[progress * this->tileLength], this->tileLength);
        // 将LocalTesor放入VECIN（代表矢量编程中搬入数据的逻辑存放位置）的Queue中
        inQueueX.EnQue(xLocal);
        inQueueY.EnQue(yLocal);
    }
```



```
    // 计算函数，完成Compute阶段的处理，被核心Process函数调用
    __aicore__ inline void Compute(int32_t progress)
    {
        // 将Tensor从队列中取出，用于后续计算
        AscendC::LocalTensor<DTYPE_X> xLocal = inQueueX.DeQue<DTYPE_X>();
        AscendC::LocalTensor<DTYPE_Y> yLocal = inQueueY.DeQue<DTYPE_Y>();
        // 从Queue中分配输出Tensor
        AscendC::LocalTensor<DTYPE_Z> zLocal = outQueueZ.AllocTensor<DTYPE_Z>();
        // 调用Add接口进行计算
        AscendC::Add(zLocal, xLocal, yLocal, this->tileLength);
        // 将计算结果LocalTensor放入到VecOut的Queue中
        outQueueZ.EnQue<DTYPE_Z>(zLocal);
        // 释放输入Tensor
        inQueueX.FreeTensor(xLocal);
        inQueueY.FreeTensor(yLocal);
    }
    // 搬出函数，完成CopyOut阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyOut(int32_t progress)
    {
        // 从VecOut的Queue中取出输出Tensor
        AscendC::LocalTensor<DTYPE_Z> zLocal = outQueueZ.DeQue<DTYPE_Z>();
        // 将输出Tensor拷贝到GlobalTensor中
        AscendC::DataCopy(zGm[progress * this->tileLength], zLocal, this->tileLength);
        // 将不再使用的LocalTensor释放
        outQueueZ.FreeTensor(zLocal);
    }


private:
    //Pipe内存管理对象
    AscendC::TPipe pipe;
    //输入数据Queue队列管理对象，QuePosition为VECIN
    AscendC::TQue<AscendC::QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;
    //输出数据Queue队列管理对象，QuePosition为VECOUT
    AscendC::TQue<AscendC::QuePosition::VECOUT, BUFFER_NUM> outQueueZ;
    //管理输入输出Global Memory内存地址的对象，其中xGm, yGm为输入，zGm为输出
    AscendC::GlobalTensor<DTYPE_X> xGm;
    AscendC::GlobalTensor<DTYPE_Y> yGm;
    AscendC::GlobalTensor<DTYPE_Z> zGm;
    // 每个核上总计算数据大小
    uint32_t blockLength;
    // 每个核上总计算数据分块个数
    uint32_t tileNum;
    // 每个分块大小
    uint32_t tileLength;
};
```

----结束

Host 侧算子实现

核函数开发并验证完成后，下一步就是进行Host侧的实现，对应“AddCustom/op_host”目录下的add_custom_tiling.h文件与add_custom.cpp文件。下面简要介绍下两个文件的关键实现，完整的样例代码可参见**add_custom_tiling.h**与**add_custom.cpp**。

步骤1 修改“add_custom_tiling.h”文件，在此文件中增加粗体部分的代码，进行Tiling参数的定义。

```
#ifndef ADD_CUSTOM_TILING_H
#define ADD_CUSTOM_TILING_H
#include "register/tilingdata_base.h"
namespace optiling {
BEGIN_TILING_DATA_DEF(TilingData)
  // AddCustom算子使用了2个tiling参数：totalLength与tileNum
  TILING_DATA_FIELD_DEF(uint32_t, totalLength);     // 总计算数据量
  TILING_DATA_FIELD_DEF(uint32_t, tileNum);         // 每个核上总计算数据分块个数
END_TILING_DATA_DEF;
```



```
// 注册tiling数据到对应的算子
REGISTER_TILING_DATA_CLASS(AddCustom, TilingData)
}
#endif // ADD_CUSTOM_TILING_H
```

步骤2 修改“add_custom.cpp”文件，进行Tiling的实现。

修改“TilingFunc”函数，实现Tiling上下文的获取，并通过上下文获取输入输出shape信息，并根据shape信息设置TilingData、序列化保存TilingData，并设置TilingKey。

```
namespace optiling {
const uint32_t BLOCK_DIM = 8;
const uint32_t TILE_NUM = 8;
static ge::graphStatus TilingFunc(gert::TilingContext* context)
{
    TilingData tiling;
    uint32_t totalLength = context->GetInputShape(0)->GetOriginShape().GetShapeSize();
    context->SetBlockDim(BLOCK_DIM);
    tiling.set_totalLength(totalLength);
    tiling.set_tileNum(TILE_NUM);
    tiling.SaveToBuffer(context->GetRawTilingData()->GetData(), context->GetRawTilingData()-
>GetCapacity());
    context->GetRawTilingData()->SetDataSize(tiling.GetDataSize());
    size_t *currentWorkspace = context->GetWorkspaceSizes(1);
    currentWorkspace[0] = 0;
    return ge::GRAPH_SUCCESS;
}
} // namespace optiling
```

步骤3 在“add_custom.cpp”文件中实现AddCustom算子的shape推导。

Add算子的输出shape等于输入shape，所以直接将输入shape赋给输出shape，当前msOpGen工具生成的代码“InferShape”函数无需修改。

步骤4 修改“add_custom.cpp”文件中的算子原型注册，此函数为入口函数。

```
namespace ops {
class AddCustom : public OpDef {
public:
    explicit AddCustom(const char* name) : OpDef(name)
    {
        // Add算子的第一个输入
        this->Input("x")
            .ParamType(REQUIRED)    // 代表输入必选
            .DataType({ ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32 })   // 输入支持的数据类型
            .Format({ ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND });   // 输入支持的数据格式
        // Add算子的第二个输入
        this->Input("y")
            .ParamType(REQUIRED)
            .DataType({ ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32 })
            .Format({ ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND });
        this->Output("z")
            .ParamType(REQUIRED)
            .DataType({ ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32 })
            .Format({ ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND });
        // 关联InferShape函数
        this->SetInferShape(ge::InferShape);
        // 关联Tiling函数
        this->AICore()
            .SetTiling(optiling::TilingFunc);
        // 注册算子支持的AI处理器型号，请替换为实际支持的AI处理器型号
        this->AICore().AddConfig("ascend*xxx*");
    }
};
// 结束算子注册
```



```
OP_ADD(AddCustom);
} // namespace ops
```

----结束

算子工程编译部署

编译AddCustom工程，生成自定义算子安装包，并将其安装到算子库中。

步骤1 编译自定义算子工程，构建生成自定义算子包。

1. 修改CMakePresets.json中**ASCEND_CANN_PACKAGE_PATH**为CANN软件的安装目录，例如：/usr/local/Ascend/ascend-toolkit/latest。

```
{
    ......
    "configurePresets": [
        {
                ......
                "ASCEND_CANN_PACKAGE_PATH": {
                    "type": "PATH",
                    "value": "/usr/local/Ascend/latest"
                },
                ......
        }
    ]
}
```

2. 在算子工程AddCustom目录下执行如下命令，进行算子工程编译。

```
./build.sh
```

编译成功后，会在当前目录下创建build_out目录，并在build_out目录下生成自定义算子安装包**custom_opp_*<target os>*_*<target architecture>*.run**，例如“custom_opp_ubuntu_x86_64.run”。

步骤2 自定义算子安装包部署。

在自定义算子包所在路径下，执行如下命令，安装自定义算子包。

```
./custom_opp_<target os>_<target architecture>.run
```

命令执行成功后，自定义算子包中的相关文件将部署至当前环境的OPP算子库的vendors/customize目录中，如果用户部署多个自定义算子包，可通过如下命令指定路径安装：

```
./custom_opp_<target os>_<target architecture>.run --install-path=<path>
```

说明：如果部署算子包时通过配置--install-path参数指定了算子包的安装目录，则在使用自定义算子前，需要执行source *<path>*/vendors/*<vendor_name>/bin/set_env.bash*命令，set_env.bash脚本中将自定义算子包的安装路径追加到环境变量ASCEND_CUSTOM_OPP_PATH中，使自定义算子在当前环境中生效。

查看部署后的目录结构，如下所示：

```
├── opp    // 算子库目录
│   ├── built-in     // 内置算子所在目录
│   ├── vendors     // 自定义算子所在目录
│      ├── config.ini
│      └── vendor_name1   // 自定义算子所在目录，若不指定路径安装，默认为“customize”
│          ├── framework     //自定义算子插件库
│          ├── op_impl
│          │   └── ai_core
│          │       └── tbe
│          │           ├── config
│          │           │   └── ${soc_version}     //昇腾AI处理器类型
│          │           │       └── aic-${soc_version}-ops-info.json     //自定义算子信息库文件
│          │           ├── vendor_name1_impl     //自定义算子实现代码文件
│          │           │   └── dynamic
```



```
│           │           │       ├── xx.cpp
│           │           │       └── xx.py
│           │           ├── kernel     //自定义算子二进制文件
│           │           │   └── ${soc_version}     //昇腾AI处理器类型
│           │           │   └── config
│           │           └── op_tiling
│           │               ├── lib
│           │               └── liboptiling.so
│           └── op_proto     //自定义算子原型库所在目录
│               ├── inc
│               │   └── op_proto.h
│               └── lib
│       ├── vendor_name2   // 存储厂商vendor_name2部署的自定义算子
```

----结束

算子 ST 测试

CANN开发套件包中提供了ST测试工具“msOpST”，用于生成算子的ST测试用例并在硬件环境中执行。

本节仅以AddCustom算子为例，介绍ST测试工具的关键执行流程。

步骤1 创建算子ST测试用例定义文件“AddCustom_case.json”，例如存储到跟算子工程目录“AddCustom”同级别的“AddCustom_st”路径下。

“AddCustom_case.json”文件的样例如下，开发者可基于此文件定制修改。

```
[
  {
    "case_name": "Test_AddCustom_001",
    "op": "AddCustom",
    "input_desc": [
      {
        "format": [
          "ND"
        ],
        "type": [
          "float16"
        ],
        "shape": [8,2048],
        "data_distribute": [
          "uniform"
        ],
        "value_range": [
          [
            0.1,
            1.0
          ]
        ],
        "name": "x"
      },
      {
        "format": [
          "ND"
        ],
        "type": [
          "float16"
        ],
        "shape": [8,2048],
        "data_distribute": [
          "uniform"
        ],
        "value_range": [
          [
            0.1,
            1.0
          ]
```



```
            ],
            "name": "y"
        }
    ],
    "output_desc": [
        {
            "format": [
                "ND"
            ],
            "type": [
                "float16"
            ],
            "shape": [8,2048],
            "name": "z"
        }
    ]
  }
]
```

步骤2 配置ST测试用例执行时依赖的环境变量。

如下为设置环境变量的示例。${INSTALL_DIR}表示CANN软件安装目录，例如，$HOME/Ascend/ascend-toolkit/latest。*{arch-os}*为运行环境的架构和操作系统，*arch*表示操作系统架构，*os*表示操作系统，例如x86_64-linux或aarch64-linux。

```
export DDK_PATH=${INSTALL_DIR}
export NPU_HOST_LIB=${INSTALL_DIR}/{arch-os}/devlib
```

提示：请根据CANN软件包实际安装路径对以上环境变量进行修改。

步骤3 进入msOpST工具所在目录，执行如下命令生成并执行测试用例。

1. 进入msOpST工具所在目录。
   ```
   cd $HOME/Ascend/ascend-toolkit/latest/python/site-packages/bin
   ```
2. 生成测试用例文件并执行。
   ```
   ./msopst run -i $HOME/AddCustom_st/AddCustom_case.json -soc <soc_version> -out $HOME/AddCustom_st
   ```
   - -i：算子测试用例定义文件（*.json）的路径，可配置为绝对路径或者相对路径。
   - -soc：昇腾AI处理器的型号，请根据实际环境进行替换。
   - -out：生成文件所在路径。

   此命令执行完成后，会输出类似如下打屏结果：

   ```
   ---------------------------------------------------------------------------
   - test case count: 1
   - success count: 1
   - failed count: 0
   ---------------------------------------------------------------------------
   2023-08-28 20:20:40 (25058) - [INFO] Process finished!
   2023-08-28 20:20:40 (25058) - [INFO] The st report saved in:  xxxx/AddCustom_st/20230828202015/st_report.json.
   ```

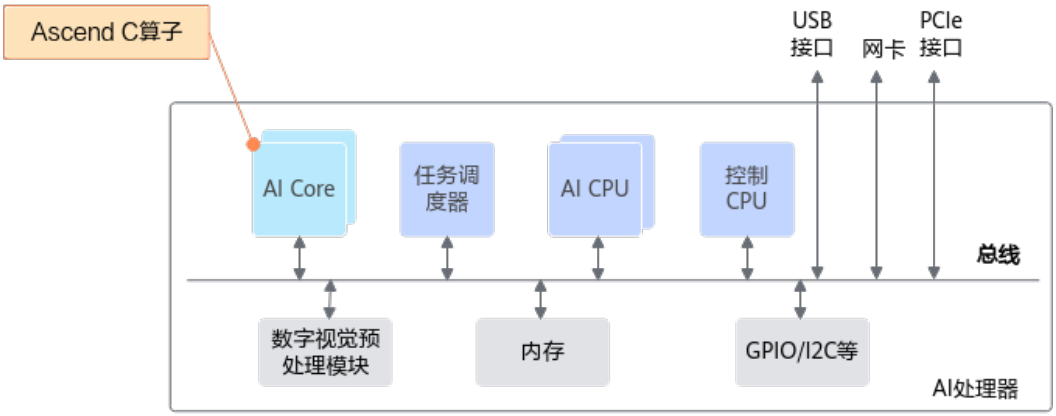
   您也可以查看上述屏显信息提示的“st_report.json”文件，查看详细运行结果。

----结束

4 硬件架构

4.1 基本架构

如下图所示，基于Ascend C开发的算子运行在AI Core上，对于初学者需要对硬件架构有基本认识，下文的编程模型基于硬件架构的抽象进行介绍，了解该内容能够更好的理解编程模型；对于需要完成高性能编程的深度开发者，更需要了解硬件架构相关知识，最佳实践中很多内容都以本章为基础进行介绍。



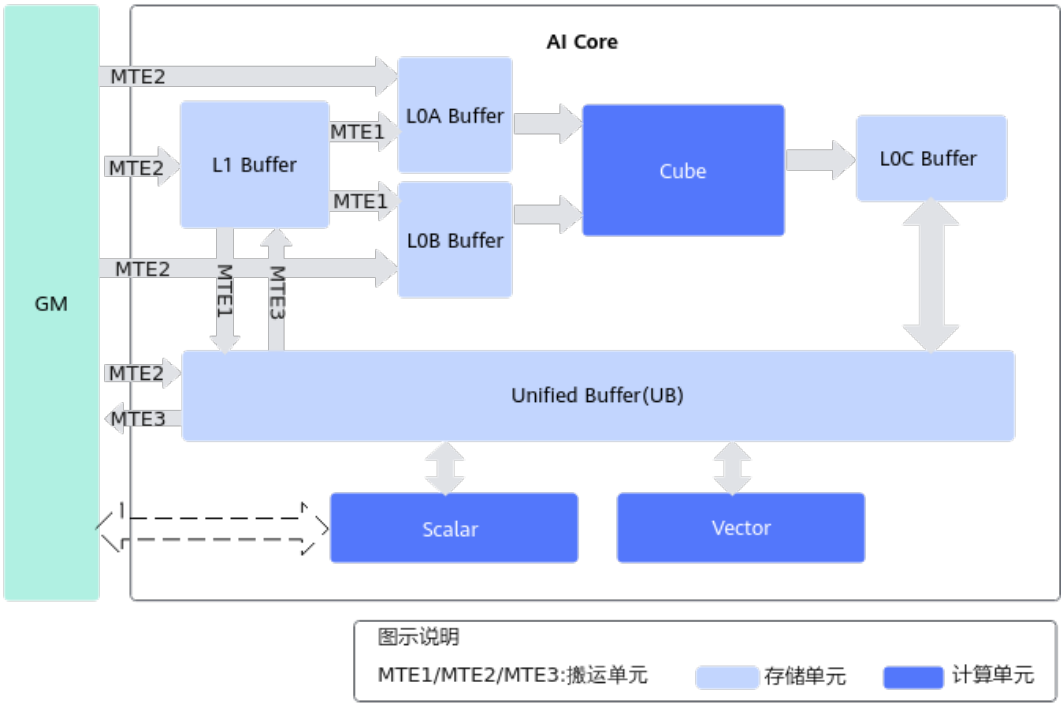
AI Core负责执行标量、向量和张量相关的计算密集型算子，包括三种基础**计算单元**：Cube（矩阵）计算单元、Vector（向量）计算单元和Scalar（标量）计算单元，同时还包含**存储单元**（包括硬件存储和用于数据搬运的搬运单元）和**控制单元**。硬件架构根据Cube计算单元和Vector计算单元是否同核部署分为**耦合架构**和**分离架构**两种。如下产品型号对应的处理器架构模式如下：

- Atlas 推理系列产品：耦合架构
- Atlas 训练系列产品：耦合架构

- Atlas A2训练系列产品/Atlas 800I A2推理产品：分离架构
- Atlas 200/500 A2推理产品：分离架构

耦合架构

耦合架构是指Cube计算单元和Vector计算单元同核部署，架构图如下图所示。下图中列出了计算架构中的存储单元和计算单元，箭头表示数据处理流向，MTE1/MTE2/MTE3代表搬运单元。

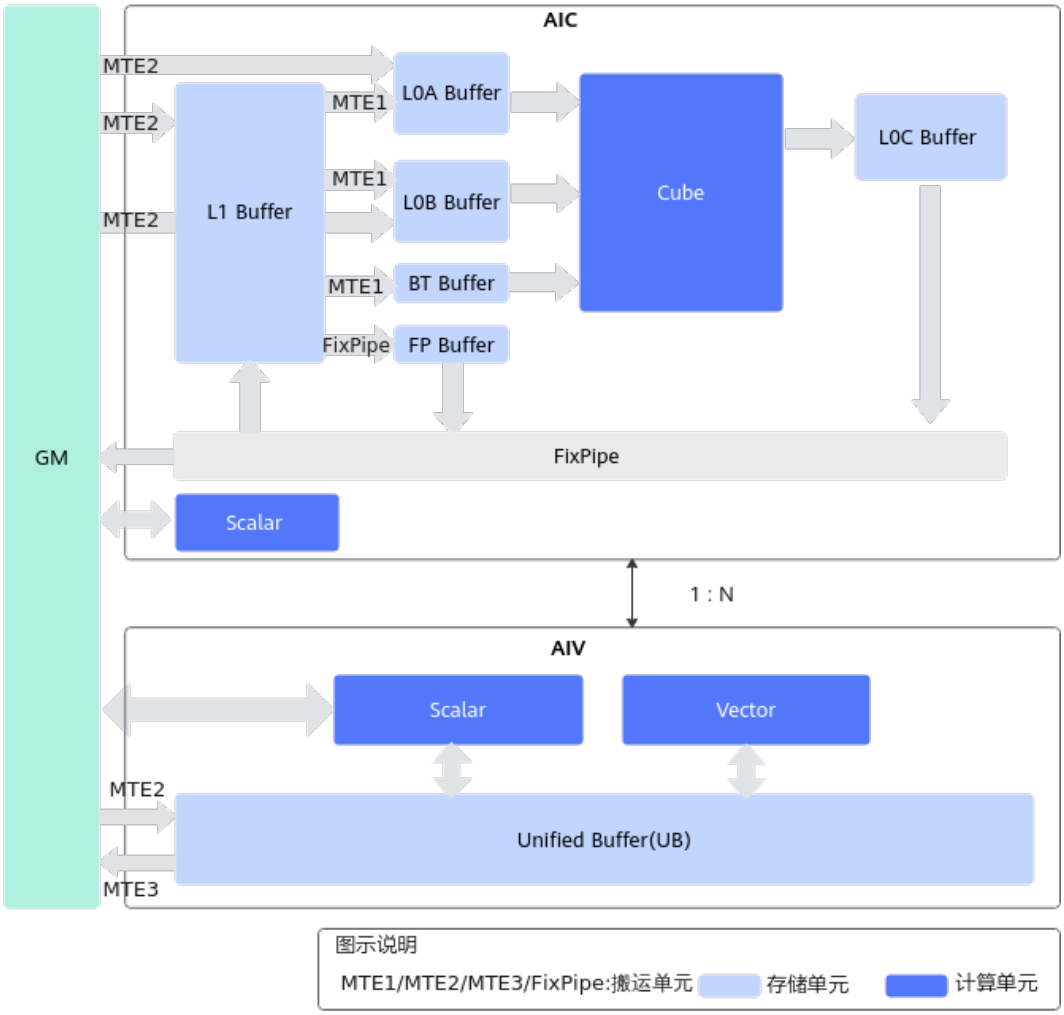


注：图中的虚线箭头表明

- Atlas 训练系列产品：不支持Scalar直接读写GM数据。
- Atlas 推理系列产品：支持Scalar直接读写GM数据。

分离架构

如下图所示，分离架构将AI Core拆成矩阵计算（AI Cube，AIC）和向量计算（AI Vector，AIV）两个独立的核，每个核都有自己的Scalar单元，能独立加载自己的代码段，从而实现矩阵计算与向量计算的解耦，在系统软件的统一调度下互相配合达到计算效率优化的效果。AIV与AIC之间通过Global Memory进行数据传递，比耦合架构增加了两个Buffer：BT Buffer(BiasTable Buffer，存放Bias)和FP Buffer(Fixpipe Buffer，存放量化参数、Relu参数等)。



- AIC架构
  - 包含5个并行执行单元（搬运单元和计算单元）：MTE1，MTE2，MTE3，Cube，Scalar
  - 包含7个内存单元：GM（核外），L1，L0A，L0B，L0C，BiasTable Buffer，Fixpipe Buffer
- AIV架构
  - 包含4个并行执行单元：MTE2，MTE3，Vector，Scalar
  - 包含2个内存单元：GM（核外），UB
- 典型计算数据流
  - Vector计算：GM-UB- [Vector]-UB-GM
  - Cube计算：
    - GM-L1-L0A/L0B-Cube-L0C-FixPipe-GM
    - GM-L1-L0A/L0B-Cube-L0C-FixPipe-L1

4.2 计算单元

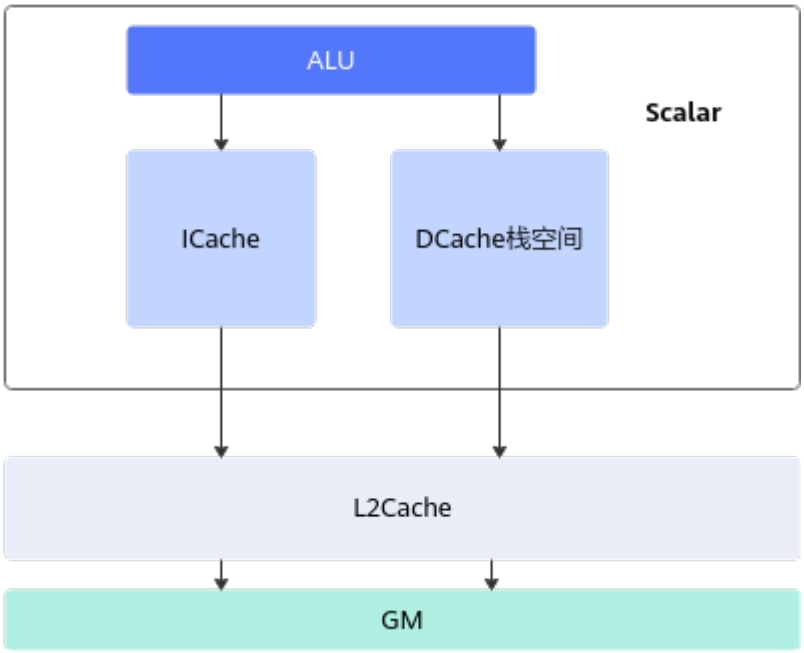
计算单元是AI Core中提供强大算力的核心单元，包括三种基础**计算单元**：Cube（矩阵）计算单元、Vector（向量）计算单元和Scalar（标量）计算单元，完成AI Core中不同类型的数据计算。

Scalar

Scalar负责各类型的标量数据运算和程序的流程控制。功能上可以看做一个小CPU，完成整个程序的循环控制、分支判断、Cube/Vector等指令的地址和参数计算以及基本的算术运算，并且可以通过在事件同步模块中插入同步符的方式来控制AI Core中其他功能性单元的执行流水。相对于Host CPU，Scalar的计算能力较弱，重点用于发射指令，性能调优时尽量减少if/else及变量运算。

如下图所示：Scalar执行标量运算指令时，执行标准的ALU(Arithmetic Logic Unit)语句，ALU需要的代码段和数据段（栈空间）都来自于GM，ICache用于缓存代码段，缓存大小与硬件规格相关，比如为16K或32K，以2K为单位加载；DCache用于缓存数据段，大小也与硬件规格相关，比如为16K，以cacheline（64Byte）为单位加载。考虑到核内访问效率最高，应尽量保证代码段和数据段被缓存在ICache和DCache，避免核外访问； 同时根据数据加载单位不同，编程时可以考虑单次加载数据大小，来提升加载效率。例如在DCache加载数据时，当数据内存首地址与cacheline（64Byte）对齐时，加载效率最高。

图 4-1 Scalar 对指令和数据的访问



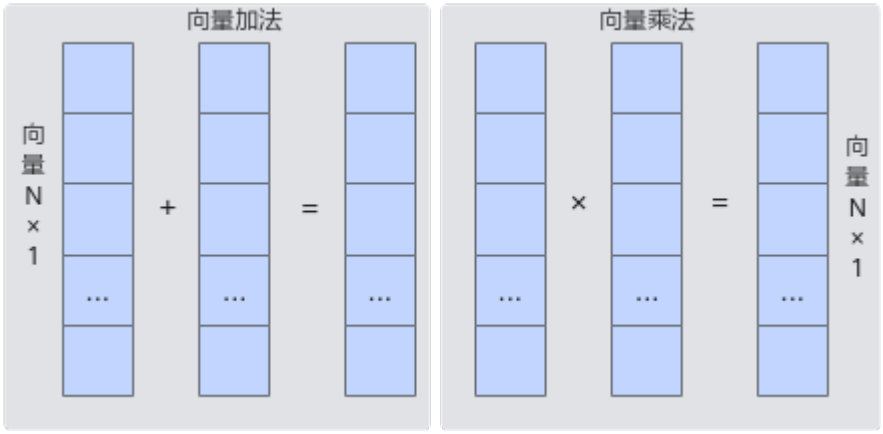
说明

硬件提供L2Cache用于缓存访问GM的数据（包括代码段、数据段），以此加快访问速度，提高访问效率。核外L2Cache以cacheline为单位加载数据，根据硬件规格不同，cacheline大小不同（128/256/512Byte等）。

Vector

Vector负责执行向量运算。向量计算单元执行向量指令，类似于传统的单指令多数据（Single Instruction Multiple Data，SIMD）指令，每个向量指令可以完成多个操作数的同一类型运算。如下图所示，向量计算单元可以快速完成两个FP16类型的向量相加或者相乘。向量指令支持多次迭代执行，也支持对带有间隔的向量直接进行运算。

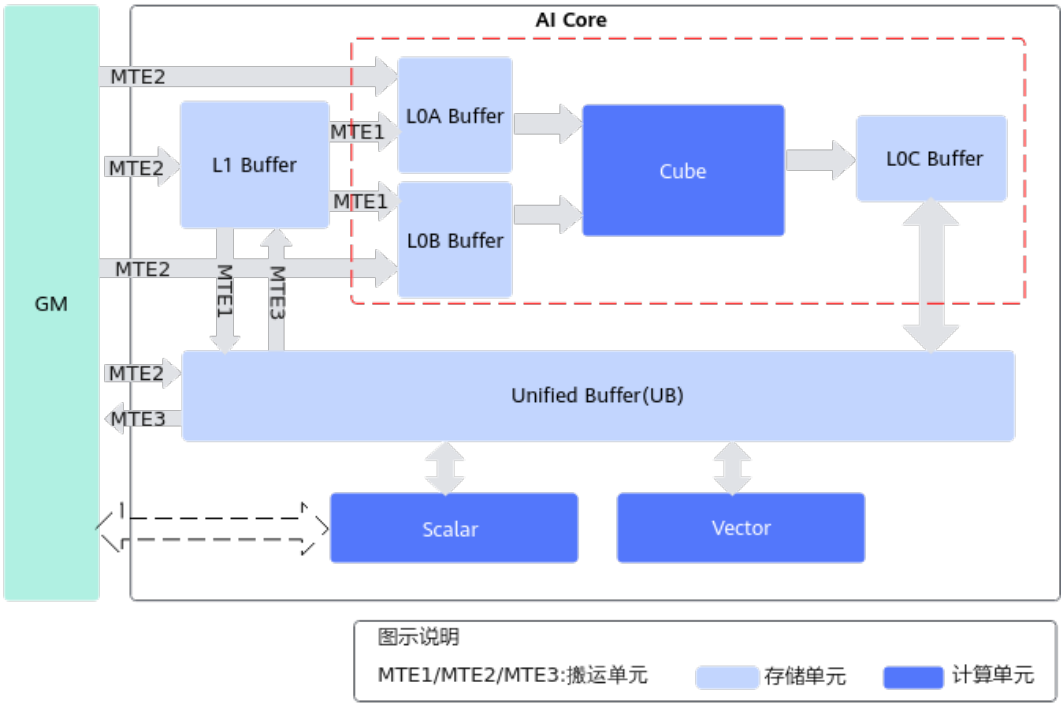
图 4-2 向量运算



Vector所有计算的源数据以及目标数据都要求存储在Unified Buffer中，**并要求首地址和操作长度都满足32Byte**对齐。

Cube

Cube计算单元负责执行矩阵运算。Cube一次执行可以完成的A矩阵（M*K）与B矩阵（K*N）的矩阵乘。如下图所示红色虚线框划出了Cube计算单元及其访问的存储单元，其中左矩阵A来源于L0A，右矩阵B来源于L0B，L0C存储矩阵乘的结果和中间结果。



4.3 存储单元

AI处理器中的计算资源要想发挥强劲算力，必要条件是保证输入数据能够及时准确地出现在计算单元中，需要精心设计存储系统，保证计算单元所需的数据供应。

AI Core中包含多级**内部存储**，AI Core需要把**外部存储**中的数据加载到内部存储中，才能完成相应的计算。AI Core的主要内部存储包括：L1 Buffer（L1缓冲区），L0 Buffer（L0缓冲区），Unified Buffer（统一缓冲区）等。为了配合AI Core中的数据传输和搬运，AI Core中还包含MTE（Memory Transfer Engine，存储转换引擎）搬运单元，在搬运过程中可执行随路数据格式/类型转换。

AI Core的内部存储列表如表4-1所示。

表 4-1 存储单元介绍

存储单元	描述
L1 Buffer	L1缓冲区，通用内部存储，是AI Core内比较大的一块数据中转区，可暂存AI Core中需要反复使用的一些数据从而减少从总线读写的次数。
L0A Buffer / L0B Buffer	Cube指令的输入。
L0C Buffer	Cube指令的输出，但进行累加计算的时候，也是输入的一部分。
Unified Buffer	统一缓冲区，向量和标量计算的输入和输出。
BT Buffer	BiasTable Buffer，存放Bias。
FP Buffer	Fixpipe Buffer，存放量化参数、Relu参数等。

表 4-2 搬运单元介绍

搬运单元	描述
MTE1	负责如下通路的数据搬运： ● L1->L0A/L0B ● L1->UB（只有耦合架构支持） ● L1->BT Buffer（只有分离架构支持）
MTE2	负责如下通路的数据搬运： ● GM->{L1, L0A/B}，在该通路下，基于分形大小搬运，搬运时满足CacheLine（512B）对齐，性能更优。 ● GM->UB，基于cacheline大小搬运性能更优。
MTE3	负责如下通路的数据搬运：UB -> GM。
FixPipe	只有分离架构支持，负责如下通路的数据搬运，搬运过程中可以完成随路数据格式/类型转换： ● L0C->{GM/L1} ● L1->FP Buffer

说明

- 不同类型的AI处理器，存储单元大小不同，开发者可通过GetCoreMemSize接口获取。
- 所有通过搬运单元读写GM的数据都缺省被缓存在L2Cache，以此加快访问速度，提高访问效率。核外L2Cache以cacheline为单位加载数据，根据硬件规格不同，cacheline大小不同（128/256/512Byte等）。

4.4 控制单元

控制单元为整个计算过程提供了指令控制，负责整个AI Core的运行。AI Core包含的控制单元如图4-3所示，每个模块的具体介绍请参考表4-3。

图 4-3 控制单元

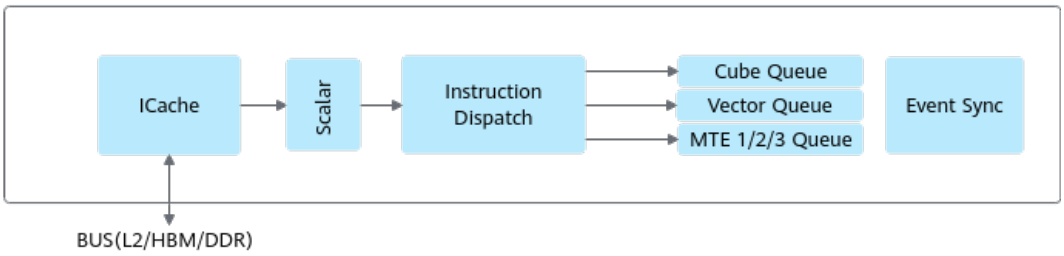


表 4-3 控制单元及相关的指令队列介绍

控制单元/指令队列	描述
标量计算单元（Scalar）	Scalar计算单元。
矩阵运算队列（Cube Queue）	Cube指令队列。同一个队列里的指令顺序执行，不同队列之间可以并行执行。
向量运算队列（Vector Queue）	Vector指令队列。同一个队列里的指令顺序执行，不同队列之间可以并行执行。
存储转换队列（MTE Queue）	MTE指令队列。同一个队列里的指令顺序执行，不同队列之间可以并行执行。
事件同步模块（Event Sync）	用于控制不同队列指令（也叫做不同指令流水）之间的依赖和同步的模块。

多条指令从系统内存通过总线接口进入到AI Core的指令缓存模块（Instruction Cache）中，后续的指令执行过程，根据指令的类型，有两种可能：

- 如果指令是Scalar指令，指令会被Scalar单元直接执行。
- 其他指令，指令会被调度到5个独立的分类队列（Vector指令队列、Cube指令队列、MTE1/MTE2/MTE3指令队列），然后再分配到某个执行单元执行。

同一个指令执行队列中的指令是按照进入队列的顺序进行执行的，不同指令执行队列之间可以并行执行，通过多个指令执行队列的并行执行可以提升整体执行效率。对于并行执行过程中可能出现的数据依赖，通过事件同步模块插入同步指令来控制流水线的同步，提供PipeBarrier、SetFlag/WaitFlag两种指令，保证队列内部以及队列之间按照逻辑关系执行。

- PipeBarrier本身是一条指令，用于在队列内部约束执行顺序。其作用是，保证前序队列中所有数据的读写工作全部完成，后序指令才能执行。
- SetFlag/WaitFlag为两条指令，在SetFlag/WaitFlag的指令中，可以指定一对指令队列的关系，表示两个队列之间完成一组“锁”机制，其作用方式为：
  - SetFlag：当前序指令的所有读写操作都完成之后，当前指令开始执行，并将硬件中的对应标志位设置为1。
  - WaitFlag：当执行到该指令时，如果发现对应标志位为0，该队列的后续指令将一直被阻塞；如果发现对应标志位为1，则将对应标志位设置为0，同时后续指令开始执行。

Ascend C API提供了用于同步控制的接口同步控制，开发者可以使用这类API来自行完成同步控制。需要注意的是，通常情况下，开发者基于**5 编程模型**中介绍的编程模型和范式进行编程时不需要关注同步，编程模型帮助开发者完成了同步控制；使用编程模型和范式是我们推荐的编程方式，自行同步控制可能会带来一定的编程复杂度。

但是我们仍然希望开发者可以理解同步的基本原理，便于后续更好的理解设计并行计算程序；同时少数情况需要开发者手动插入同步，您可以通过什么时候需要开发者手动插入同步来了解具体内容。

5 编程模型

5.1 SPMD 模型

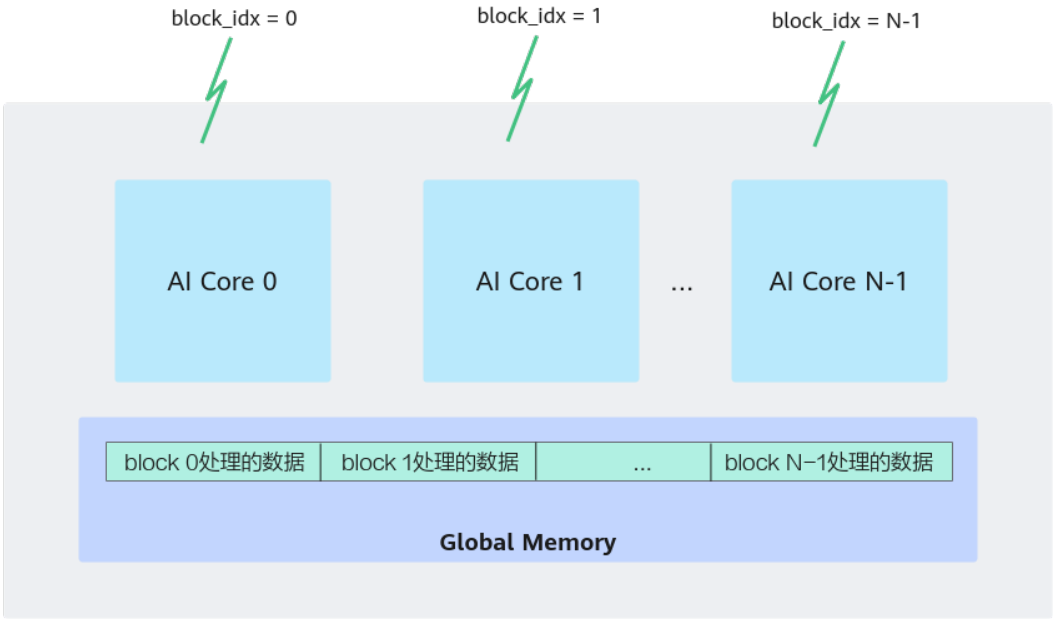
Ascend C算子编程是SPMD（Single-Program Multiple-Data）编程，SPMD是一种常用的并行计算的方法，是提高计算速度的有效手段。

假设，从输入数据到输出数据需要经过3个阶段任务的处理（T1、T2、T3）。如下图所示，SPMD模式下，系统会启动一组进程，并行处理待处理的数据：首先待处理数据会被切分成多个数据分片，切分后的数据分片随后被分发给不同进程处理，每个进程接收到一个或多个数据分片，并独立地对这些分片进行3个任务的处理。

图 5-1 SPMD 数据并行示意图

具体到Ascend C编程模型中的应用，是将需要处理的数据拆分并同时在多个计算核心（类比于上文介绍中的多个进程）上运行，从而获取更高的性能。多个AI Core共享相同的指令代码，每个核上的运行实例唯一的区别是block_idx不同，每个核通过不同的block_idx来识别自己的身份。block的概念类似于上文中进程的概念，block_idx就是标识进程唯一性的进程ID。并行计算过程如下图所示。

图 5-2 SPMD 并行计算示意图



下面的代码片段取自于Ascend C Add算子的实现代码，算子被调用时，所有的计算核心都执行相同的实现代码，入口函数的入参也是相同的。每个核上处理的数据地址需要在起始地址上增加**GetBlockIdx()***BLOCK_LENGTH（每个block处理的数据长度）的偏移来获取。这样也就实现了多核并行计算的数据切分。

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd() {}
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z)
    {
        // get start index for current core, core parallel
        xGm.SetGlobalBuffer((__gm__ half*)x + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
        yGm.SetGlobalBuffer((__gm__ half*)y + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
        zGm.SetGlobalBuffer((__gm__ half*)z + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
        // pipe alloc memory to queue, the unit is Bytes
        pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half));
        pipe.InitBuffer(inQueueY, BUFFER_NUM, TILE_LENGTH * sizeof(half));
        pipe.InitBuffer(outQueueZ, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    }
    ...
}


// 实现核函数
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    // 初始化算子类，算子类提供算子初始化和核心处理等方法
    KernelAdd op;
    // 初始化函数，获取该核函数需要处理的输入输出地址，同时完成必要的内存初始化工作
    op.Init(x, y, z);
    // 核心处理函数，完成算子的数据搬运与计算等核心逻辑
    op.Process();
}
```


5.2 核函数

从5.1 SPMD模型可以得知，使用Ascend C进行编程时，我们编写一份算子实现代码，算子被调用时，将启动N个运行实例，在N个核上运行。本节将介绍算子实现的入口函数。

核函数（Kernel Function）是Ascend C算子设备侧实现的入口。在核函数中，需要为在一个核上执行的代码规定要进行的数据访问和计算操作，当核函数被调用时，多个核都执行相同的核函数代码，具有相同的参数，并行执行。

Ascend C允许用户使用核函数这种C/C++函数的语法扩展来管理设备端的运行代码，用户在核函数中进行算子类对象的创建和其成员函数的调用，由此实现该算子的所有功能。核函数是主机端和设备端连接的桥梁，本章将具体介绍核函数的用法。

核函数定义

```
extern "C" __global__ __aicore__ void add_custom(__gm__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t* z);
```

以上是一个核函数声明的例子，编写核函数时需要遵循以下规则。

- **使用函数类型限定符**

  除了需要按照C/C++函数声明的方式定义核函数之外，还要为核函数加上额外的函数类型限定符，包含__global__和__aicore__。

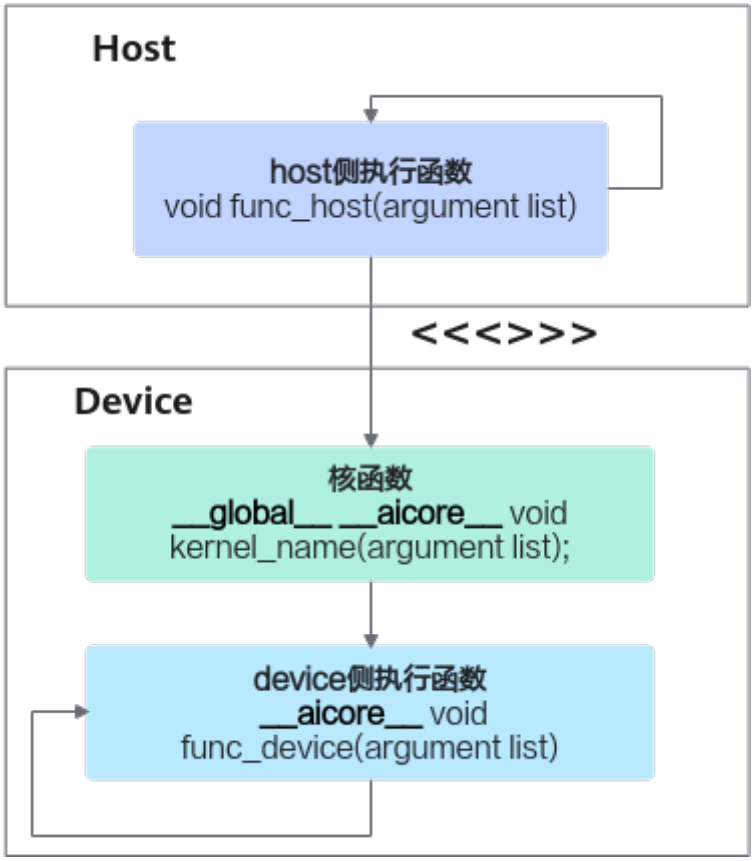
  使用__global__函数类型限定符来标识它是一个核函数，可以被<<<>>>调用；使用__aicore__函数类型限定符来标识该核函数在设备端AI Core上执行：

  ```
  __global__ __aicore__ void kernel_name(argument list);
  ```

  编程中使用到的函数可以分为三类：核函数（device侧执行）、host侧执行函数、device侧执行函数（除核函数之外的）。三者的调用关系如下图所示：

  - host侧执行函数可以调用同类的host执行函数，也就是通用C/C++编程中的函数调用；也可以通过<<<>>>调用核函数。
  - device侧执行函数（除核函数之外的）可以调用同类的device侧执行函数。
  - 核函数可以调用device侧执行函数（除核函数之外的）。

图 5-3 核函数（device 侧执行）、host 侧执行函数、device 侧执行函数（除核函数之外的）调用关系



- **使用变量类型限定符**

  指针入参变量需要增加变量类型限定符__gm__，表明该指针变量指向Global Memory上某处内存地址。

- **其他规则或建议**

  a. 规则：核函数必须具有void返回类型。

  b. 规则：仅支持入参为指针或C/C++内置数据类型（Primitive data types），如：half* s0、float* s1、int32_t c。

  c. 建议：为了统一表达，建议使用GM_ADDR宏来修饰入参，GM_ADDR宏定义如下：

  ```
  #define GM_ADDR __gm__ uint8_t*
  ```

  使用GM_ADDR修饰入参的样例如下：

  ```
  extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
  ```

  这里统一使用uint8_t类型的指针，在后续的使用中需要将其转化为实际的指针类型。

核函数调用

核函数的调用语句是C/C++函数调用语句的一种扩展。本节仅描述最基础的调用方式，实际进行算子开发时，您可以通过该方式调用，也可以通过接入CANN框架的方式调用，更多细节请参考**9 工程化算子开发**。

常见的函数调用方式是如下的形式：

```
function_name(argument list);
```


核函数使用内核调用符<<<...>>>这种语法形式，来规定核函数的执行配置：

```
kernel_name<<<blockDim, l2ctrl, stream>>>(argument list);
```

内核调用符仅可在NPU侧编译时调用，CPU侧编译无法识别该符号。

执行配置由3个参数决定：

- blockDim，规定了核函数将会在几个核上执行。每个执行该核函数的核会被分配一个逻辑ID，即block_idx，可以在核函数的实现中调用GetBlockIdx来获取block_idx；

  说明

  blockDim是逻辑核的概念，取值范围为[1,65535]。为了充分利用硬件资源，一般设置为物理核的核数或其倍数。对于耦合架构和分离架构，blockDim在运行时的意义和设置规则有一些区别，具体说明如下：

  - 耦合架构：由于其Vector、Cube单元是集成在一起的，blockDim用于设置启动多个AICore核实例执行，不区分Vector、Cube。AI Core的核数可以通过GetCoreNumAiv或者GetCoreNumAic获取。
  - 分离架构
    - 针对仅包含Vector计算的算子，blockDim用于设置启动多少个Vector（AIV）实例执行，比如某款AI处理器上有40个Vector核，建议设置为40。
    - 针对仅包含Cube计算的算子，blockDim用于设置启动多少个Cube（AIC）实例执行，比如某款AI处理器上有20个Cube核，建议设置为20。
    - 针对Vector/Cube融合计算的算子，启动时，按照AIV和AIC组合启动，blockDim用于设置启动多少个组合执行，比如某款AI处理器上有40个Vector核和20个Cube核，一个组合是2个Vector核和1个Cube核，建议设置为20，此时会启动20个组合，即40个Vector核和20个Cube核。**注意：该场景下，设置的blockDim逻辑核的核数不能超过物理核（2个Vector核和1个Cube核组合为1个物理核）的核数。**
    - AIC/AIV的核数分别通过GetCoreNumAic和GetCoreNumAiv接口获取。
- l2ctrl，保留参数，暂时设置为固定值nullptr，开发者无需关注；
- stream，类型为aclrtStream，stream用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序在device上执行。stream创建等管理接口请参考Stream管理。

如下名为add_custom的核函数，实现两个矢量的相加，调用示例如下：

```
// blockDim设置为8表示在8个核上调用了add_custom核函数，每个核都会独立且并行地执行该核函数，该核函数的参数列表为x，y，z。
add_custom<<<8, nullptr, stream>>>(x, y, z);
```

核函数的调用是异步的，核函数的调用结束后，控制权立刻返回给主机端，可以调用以下aclrtSynchronizeStream函数来强制主机端程序等待所有核函数执行完毕。

```
aclError aclrtSynchronizeStream(aclrtStream stream);
```

核函数示例

下面提供核函数实现和调用的样例代码片段，完整样例请参考**Add算子示例**。

```
// 实现核函数
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    // 初始化算子类，算子类提供算子初始化和核心处理等方法
    KernelAdd op;
    // 初始化函数，获取该核函数需要处理的输入输出地址，同时完成必要的内存初始化工作
    op.Init(x, y, z);
    // 核心处理函数，完成算子的数据搬运与计算等核心逻辑
    op.Process();
```

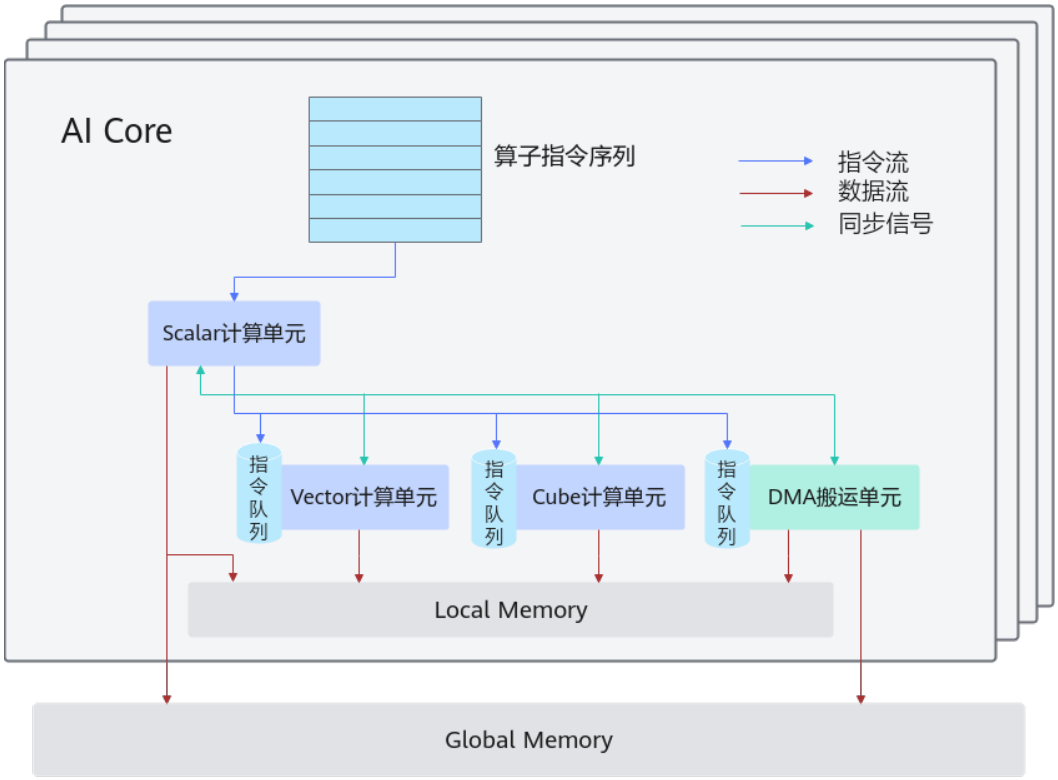


```
}

// 调用核函数
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y, uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
```

5.3 硬件架构抽象

Ascend C基于硬件抽象架构进行编程， 进行屏蔽不同硬件之间的差异。

图 5-4 硬件架构抽象



AI Core中包含**计算单元、存储单元、搬运单元**等核心组件。

- 计算单元包括了三种基础计算资源：Cube计算单元、Vector计算单元和Scalar计算单元。
- 存储单元包括内部存储和外部存储：
  - AI Core的内部存储，统称为Local Memory，对应的数据类型为LocalTensor。由于不同芯片间硬件资源不固定，可以为UB、L1、L0A、L0B等。
  - AI Core能够访问的外部存储称之为Global Memory，对应的数据类型为GlobalTensor。
- DMA（Direct Memory Access）搬运单元：负责在Global Memory和Local Memory之间搬运数据。

AI Core内部核心组件及组件功能详细说明如下表。

表 5-1 AI Core 内部核心组件

组件分类	组件名称	组件功能
计算单元	Scalar	执行地址计算、循环控制等标量计算工作，并把向量计算、矩阵计算、数据搬运、同步指令发射给对应单元执行。
	Vector	负责执行向量运算。
	Cube	负责执行矩阵运算。
存储单元	Local Memory	AI Core的内部存储。
搬运单元	DMA（Direct Memory Access）	负责在Global Memory和Local Memory之间搬运数据，包含搬运单元MTE2（Memory Transfer Engine，数据搬入单元），MTE3（数据搬出单元）等。

开发者在理解硬件架构的抽象时，需要重点关注如下**异步指令流**、**同步信号流**、**计算数据流**三个过程：

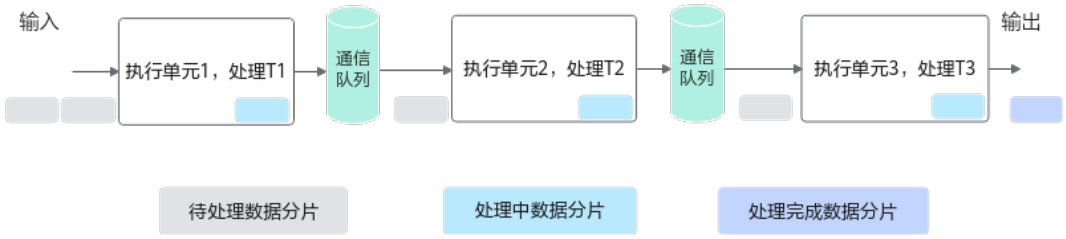
- AI Core内部的异步并行计算过程：Scalar计算单元读取指令序列，并把向量计算、矩阵计算、数据搬运指令发射给对应单元的指令队列，向量计算单元、矩阵计算单元、数据搬运单元异步的并行执行接收到的指令。该过程可以参考**图1**中蓝色箭头所示的指令流。
- 不同的指令间有可能存在依赖关系，为了保证不同指令队列间的指令按照正确的逻辑关系执行，Scalar计算单元也会给对应单元下发同步指令。各单元之间的同步过程可以参考**图1**中的绿色箭头所示的同步信号流。
- AI Core内部数据处理的基本过程：DMA搬入单元把数据搬运到Local Memory，Vector/Cube计算单元完成数据计算，并把计算结果写回Local Memory，DMA搬出单元把处理好的数据搬运回Global Memory。该过程可以参考**图1**中的红色箭头所示的数据流。

5.4 编程范式

编程范式描述了算子实现的固定流程，基于编程范式进行编程，可以快速搭建算子实现的代码框架。

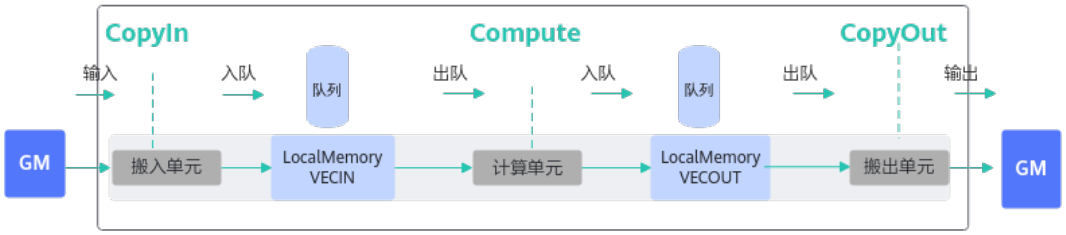
根据**5.3 硬件架构抽象**可以了解到，AI Core内部的执行单元异步并行地执行接收到的指令。如下图所示，从输入数据到输出数据需要经过3个阶段任务的处理（T1、T2、T3），多个执行单元并行处理，每个执行单元只会专注于一个任务的处理，会处理所有的数据分片。可以看出，流水线并行和工业生产中的流水线是类似的，每一个执行单元都可以看成是流水线上的节点，通过流水线并行的方式来提高计算效率：执行单元1完成对某个数据分片的处理后，将其加入到通信队列，执行单元2空闲时就会从队列中取出数据继续处理；可以类比为生产流水线中的工人只完成某一项固定工序，完成后就交由下一项工序负责人继续处理。

图 5-5 流水线并行示意图



Ascend C编程范式就是这样一种流水线式的编程范式，把算子核内的处理程序，分成多个**流水任务**，通过队列（Queue）完成**任务间通信和同步**，并通过统一的**资源管理**模块（Pipe）来统一管理内存、事件等资源。

Vector 编程范式



如上图所示，Vector编程范式把算子的实现流程分为3个基本任务：CopyIn，Compute，CopyOut。

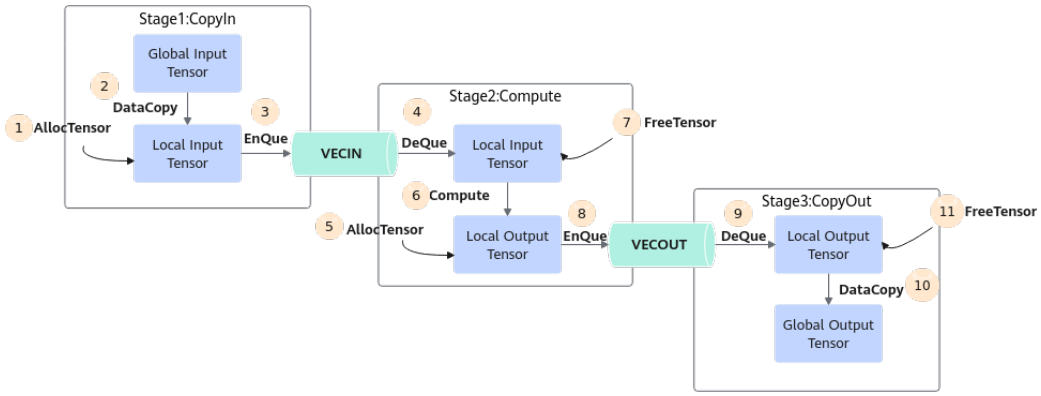
- **CopyIn**负责搬入操作：将输入数据从Global Memory搬运到Local Memory（VECIN用于表达矢量计算搬入数据的存放位置），完成搬运后执行入队列操作；
- **Compute**负责矢量指令计算操作：完成队列出队后，从Local Memory获取数据并计算，计算完成后执行入队操作；
- **CopyOut**负责搬出操作：完成队列出队后，将计算结果从Local Memory（VECOUT用于表达矢量计算搬出数据的存放位置）搬运到GM。

上文中提到的VECIN/VECOUT是TPostition的概念。Ascend C管理不同层级的物理内存时，用一种抽象的逻辑位置（TPosition）来表达各级别的存储，代替了片上物理存储的概念，达到隐藏硬件架构的目的。除了VECIN/VECOUT，矢量编程中还会使用到VECCALC，一般在定义临时变量时使用此位置。这些TPostition与物理内存的映射关系如下表。

表 5-2 TPosition 与物理内存映射关系

TPosition	物理内存
GM	Global Memory
VECIN	Unified Buffer
VECOUT	Unified Buffer
VECCALC	Unified Buffer

从编程的角度来讲，具体流程（如下文的伪代码）和流程图如下：

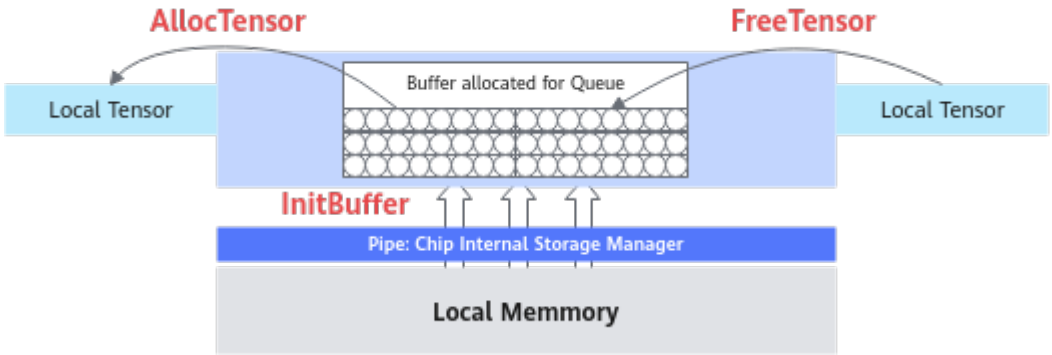



```
AscendC::TPipe pipe;   //创建全局的资源管理
AscendC::TQue<AscendC::QuePosition::VecIn, 1> queIn;  //创建CopyIn阶段的队列
AscendC::TQue<AscendC::QuePosition::VecOut, 1> queOut; //创建CopyOut阶段的队列
// Init 阶段:
pipe.InitBuffer(queIn, 2, 1024);  // 开启double buffer,将待处理的数据一分为二,实现流水并行
for-loop {
   //CopyIn 阶段{
   auto tensor = queIn.AllocTensor<half>();     //从Que上申请资源, 长度1024
   AscendC::DataCopy(tensor, gm, len);                  //搬运数据从GM到VECIN
   queIn.EnQue(tensor);
   }
   //Compute阶段{
   auto tensor = queIn.DeQue<half>();
   auto tensorOut = queOut.AllocTensor<half>();
   AscendC::Abs(tensorOut, tensor, 1024);
   queIn.FreeTensor(tensor);
   queOut.EnQue(tensorOut);
   }
   //CopyOut 阶段{
   auto tensor = queOut.DeQue<half>();
   AscendC::DataCopy(gmOut, tensor, 1024);
   queOut.FreeTensor(tensor);
   }
}
```

任务间数据传递使用到的内存、事件等资源统一由管理模块Pipe进行管理。如下所示的内存管理示意图，TPipe通过InitBuffer接口对外提供Queue内存初始化功能，开发者可以通过该接口为指定的Queue分配内存。

Queue队列内存初始化完成后，需要使用内存时，通过调用AllocTensor来为LocalTensor分配内存，当创建的LocalTensor完成相关计算无需再使用时，再调用FreeTensor来回收LocalTensor的内存。

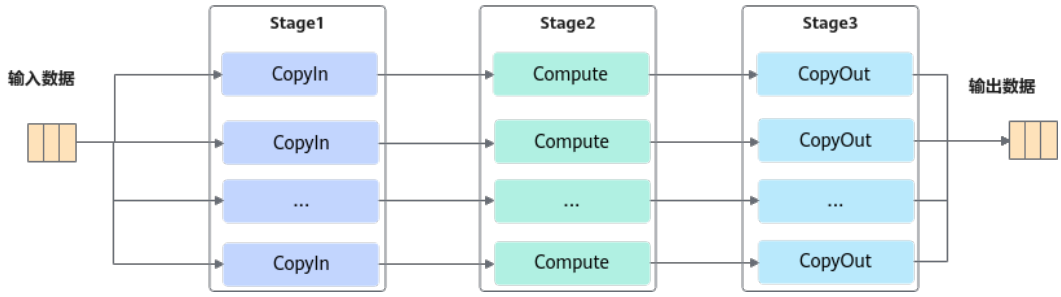
图 5-6 内存管理示意图



编程过程中使用到的临时变量内存同样通过Pipe进行管理。临时变量可以使用TBuf数据结构来申请指定TPosition上的存储空间。使用TBuf申请的内存空间只能参与计算，无法执行Queue队列的入队出队操作。具体的接口使用说明请参考TBuf。

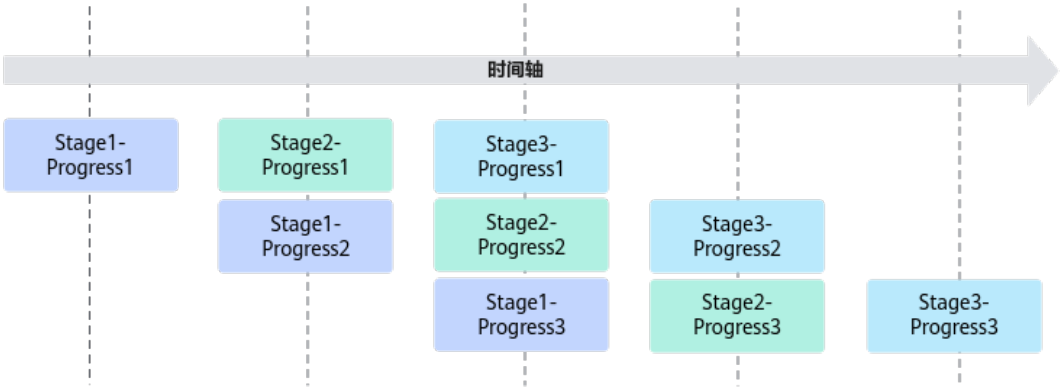
按照上述编程范式进行编程即可实现单核上数据的并行处理。需要处理的数据被切分成n片，每个并行任务（Stage1、2、3）需要依次完成n个数据切片的处理。Stage间的箭头表达数据间的依赖关系，比如Stage1（CopyIn）处理完第一个数据分片之后，Stage2（Compute）才能对该分片进行处理。

图 5-7 流水任务示意图



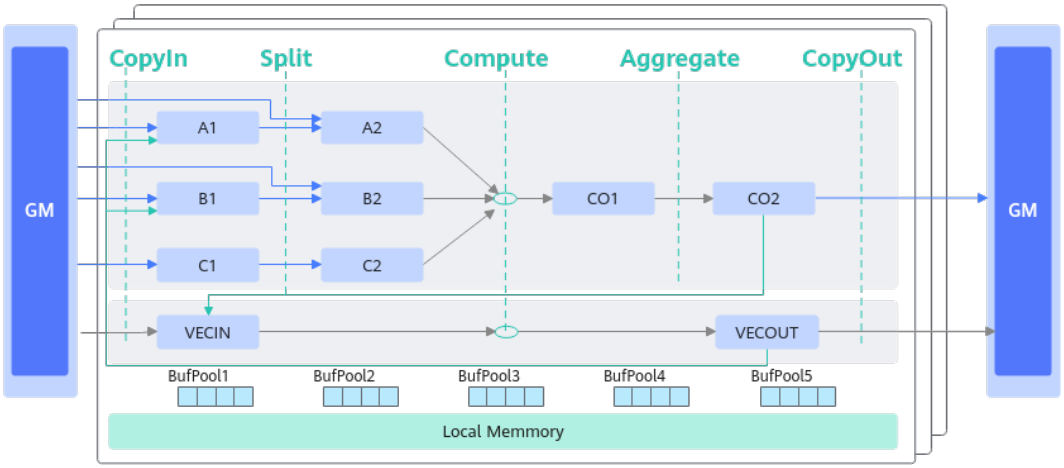
上图中的流水任务运行起来的示意图如下，Progress1、2、3代表处理的数据分片，从运行图中可以看出，对于同一片数据，Stage1、Stage2、Stage3之间的处理具有依赖关系，需要串行处理；不同的数据切片，同一时间点，可以有多个任务在并行处理，由此达到任务并行、提升性能的目的。

图 5-8 流水任务运行示意图



Cube 编程范式

Cube计算的典型数据流图如下所示：



和矢量编程范式一样，同样也使用逻辑位置（TPosition）来表达数据流，Cube编程范式中主要使用的逻辑位置定义如下：

- 搬入数据的存放位置：A1，用于存放整块A矩阵，可类比CPU多级缓存中的二级缓存；
- 搬入数据的存放位置：B1，用于存放整块B矩阵，可类比CPU多级缓存中的二级缓存；
- 搬入数据的存放位置：A2，用于存放切分后的小块A矩阵，可类比CPU多级缓存中的一级缓存；
- 搬入数据的存放位置：B2，用于存放切分后的小块B矩阵，可类比CPU多级缓存中的一级缓存；
- 结果数据的存放位置：CO1，用于存放小块结果C矩阵，可理解为Cube Out；
- 结果数据的存放位置：CO2，用于存放整块结果C矩阵，可理解为Cube Out；
- 搬入数据的存放位置：VECIN，用于矢量计算，实际业务在数据搬入Vector计算单元时使用此位置；
- 搬入数据的存放位置：VECCALC，用于矢量计算，实际业务一般在计算需要临时变量时使用此位置；
- 搬出数据的存放位置：VECOUT，用于矢量计算，实际业务在将Vector计算单元结果搬出时使用此位置。

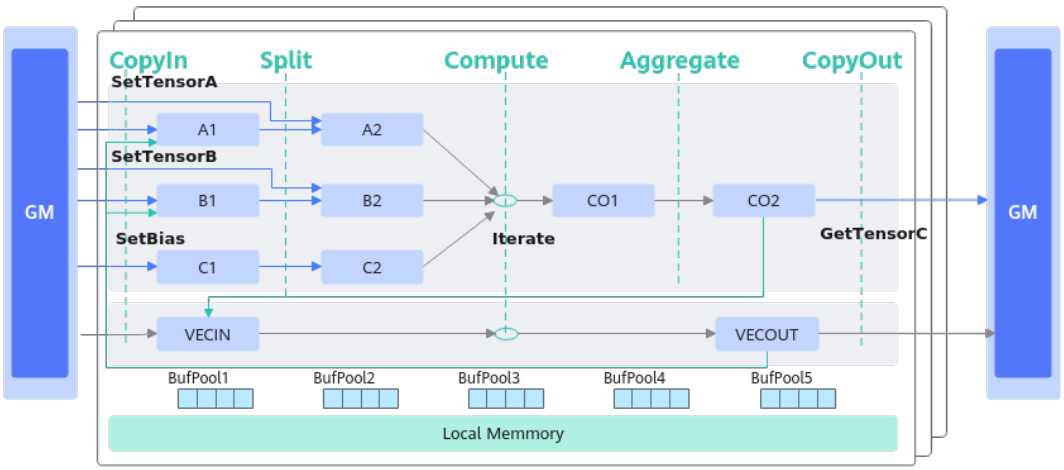
上述TPosition与物理内存的映射关系如下：

表 5-3 TPosition 与物理内存映射关系

TPosition	物理内存
GM	Global Memory
VECIN	Unified Buffer
VECCALC	Unified Buffer
VECOUT	Unified Buffer
A1	L1 Buffer

TPosition	物理内存
A2	L0A Buffer
B1	L1 Buffer
B2	L0B Buffer
C1	Atlas 训练系列产品，Unified Buffer。 Atlas推理系列产品AI Core，Unified Buffer。 Atlas A2训练系列产品/Atlas 800I A2推理产品，L1 Buffer。
C2	Atlas 训练系列产品，L0C Buffer。 Atlas推理系列产品AI Core，L0C Buffer。 Atlas A2训练系列产品/Atlas 800I A2推理产品，BT Buffer。
CO1	L0C Buffer
CO2	Atlas 训练系列产品，Unified Buffer。 Atlas推理系列产品AI Core，Unified Buffer。 Atlas A2训练系列产品/Atlas 800I A2推理产品，Global Memory。

Cube计算流程同样也可以理解为CopyIn、Compute、CopyOut这几个阶段，因为流程相对复杂，Matmul高阶API提供对此的高阶封装，编程范式如下：



具体流程可参考如下示例：

```
// 创建Matmul对象 创建对象时需要传入A、B、C、Bias的参数类型信息， 类型信息通过MatmulType来定义，包括：内存逻辑位置、数据格式、数据类型。
typedef MatmulType<TPosition::GM, CubeFormat::ND, half> aType;
typedef MatmulType<TPosition::GM, CubeFormat::ND, half> bType;
typedef MatmulType<TPosition::GM, CubeFormat::ND, float> cType;
```



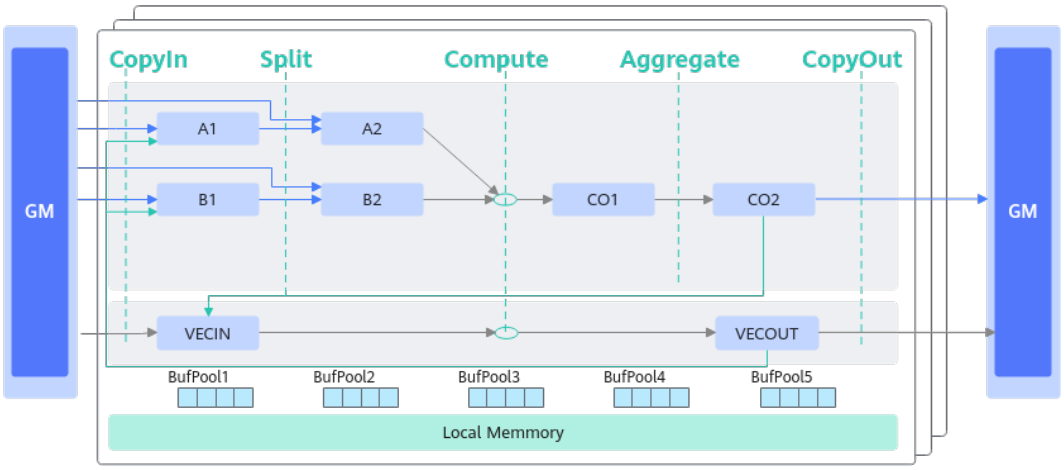
```
typedef MatmulType<TPosition::GM, CubeFormat::ND, float> biasType;
Matmul<aType, bType, cType, biasType> mm;

REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm, &tiling); // 初始化
// CopyIn阶段：完成从GM到LocalMemory的搬运
mm.SetTensorA(gm_a);    // 设置左矩阵A
mm.SetTensorB(gm_b);    // 设置右矩阵B
mm.SetBias(gm_bias);    // 设置Bias
// Compute阶段：完成矩阵乘计算
while (mm.Iterate()) {
    // CopyOut阶段：完成从LocalMemory到GM的搬运
    mm.GetTensorC(gm_c);
}
// 结束矩阵乘操作
mm.End();
```

融合算子编程范式

支持Vector与Cube混合计算的算子称之为融合算子。Ascend C提供**融合算子的编程范式**，方便开发者基于该范式表达融合算子的数据流，快速实现自己的融合算子。

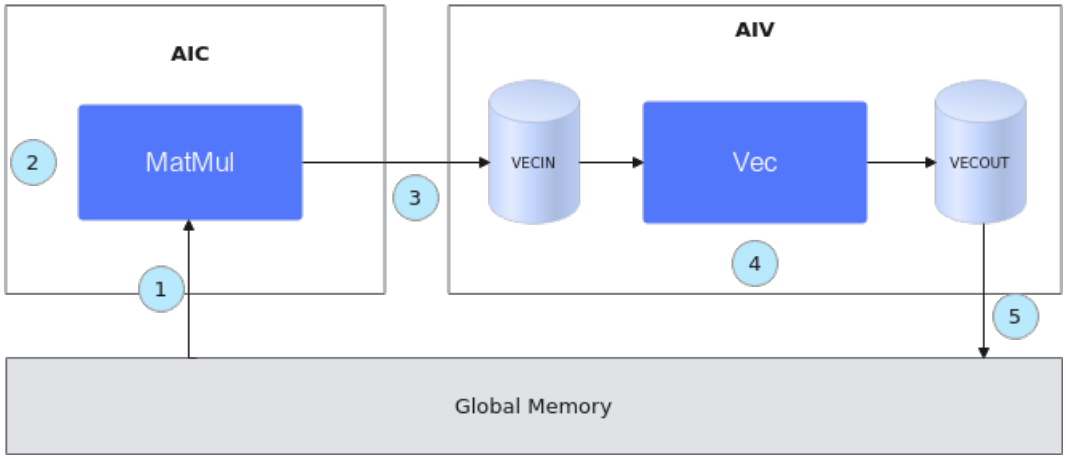
融合算子数据流指融合算子的输入输出在各存储位置间的流向。以一个典型的Cube和Vector融合算子为例，逻辑位置间的数据流向如下图所示（为了简化描述，没有列出bias）：

- Cube的输出可以作为Vector的输入：CO2->VECIN
- Vector的输出可以作为Cube的输入：VECOUT->A1->A2、VECOUT->B1->B2



基于Matmul高阶API的融合算子编程范式，对上述数据流简化表达如下：

图 5-9 融合算子编程范式



1. 初始化一个MatMul对象，将输入数据从Global Memory搬运到Cube核上。
2. 进行MatMul内部的计算。
3. 将MatMul的计算结果搬运到Vector核上。
4. 进行Vector矢量计算。
5. 将输出结果搬运到Global Memory上。

整个过程的示例代码如下（伪代码）：

```
template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::Process()
{
    // 步骤1：初始化一个MatMul对象，将输入数据从Global Memory搬运到Cube核上。
    uint32_t computeRound = 0;
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), matmulObj);
    matmulObj.Init(&tiling);
    matmulObj.SetTensorA(aGlobal);
    matmulObj.SetTensorB(bGlobal);
    matmulObj.SetBias(biasGlobal);

    while (matmulObj.template Iterate<true>()) { // 步骤2：进行MatMul内部的计算。
        // 步骤3：将MatMul的计算结果搬运到Vector核上。
        reluOutLocal = reluOutQueue_.AllocTensor<cType>();
        matmulObj.template GetTensorC<true>(reluOutLocal, false, true);
       // 步骤4：进行Vector矢量计算。
        AscendC::LeakyRelu(reluOutLocal, reluOutLocal, (cType)alpha, tiling.baseM * tiling.baseN);
        reluOutQueue_.EnQue(reluOutLocal);
        // 步骤5：将输出结果搬运到Global Memory上
        reluOutQueue_.DeQue<cType>();
        ...
        AscendC::DataCopy(cGlobal[startOffset], reluOutLocal, copyParam);
        reluOutQueue_.FreeTensor(reluOutLocal);

        computeRound++;
    }
    matmulObj.End();
}
```

编程模型背后的奥秘

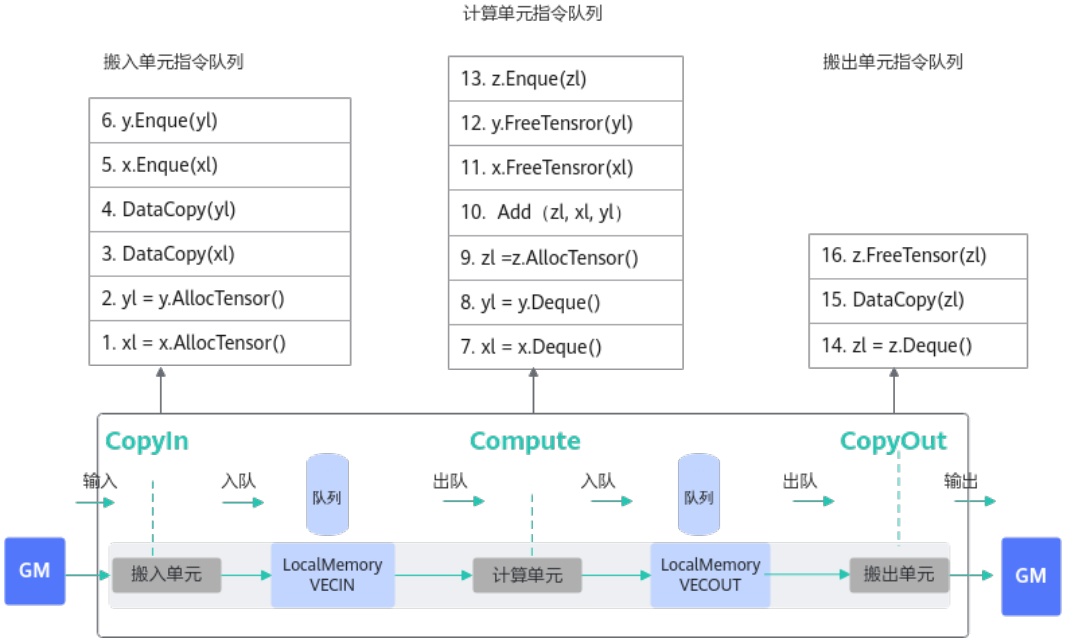
由上文可知，Ascend C的并行编程范式核心要素是：一组并行计算任务、通过队列实现任务之间的通信和同步、开发者自主表达对并行计算任务和资源的调度。本节介绍编程模型的实现原理，作为扩展阅读，便于开发者更好的理解编程模型的设计思路和优势，对于后续的深度开发也会有所帮助。

每个并行任务Stage的编程范式如下：

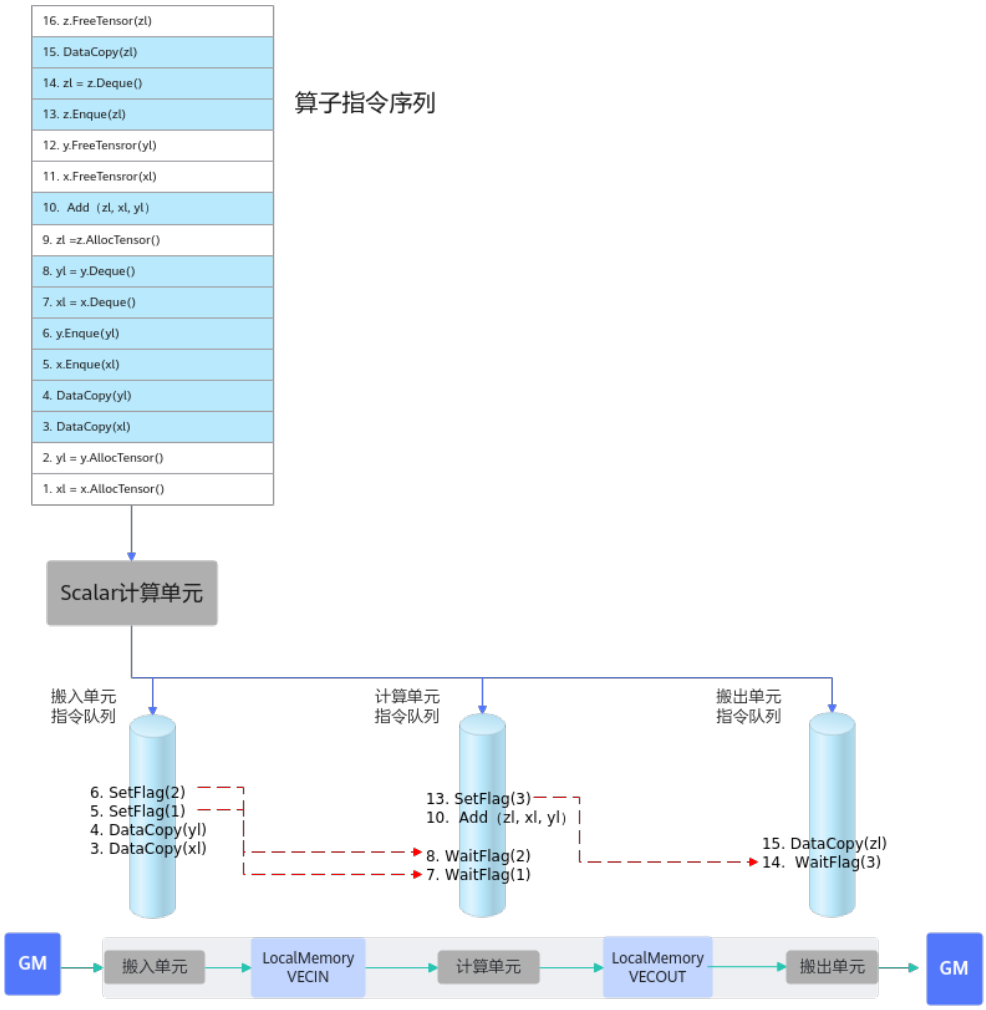
1. 获取Local Memory的内存，调用**AllocTensor**申请内存，或者从上游队列**DeQue**一块内存数据。
2. 完成计算或者数据搬运。
3. 把上一步处理好的数据调用**EnQue**入队。
4. 调用**FreeTensor**释放不再需要的内存。

以最简单的矢量编程范式为例，在调用上述接口时，实际上会给各执行单元下发一些指令，如下图所示：

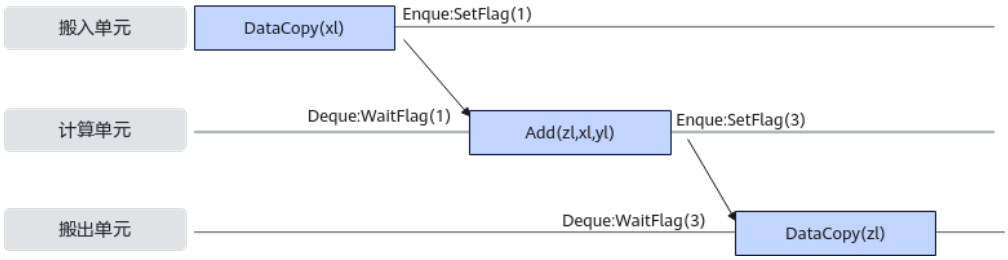
图 5-10 Vector 编程范式指令队列示例



- EnQue/DeQue的具体处理流程：
  a. 标量执行单元读取算子指令序列
  b. 把这些指令发射到对应的执行单元的指令队列
  c. 各个执行单元并行执行这些指令序列
  d. EnQue/DeQue解决对内存的写后读问题
     - EnQue调用会发射同步指令set，发送信号激活wait
     - DeQue调用会发射同步指令wait，等待数据写入完成
     - wait需要等到set信号才能执行否则阻塞

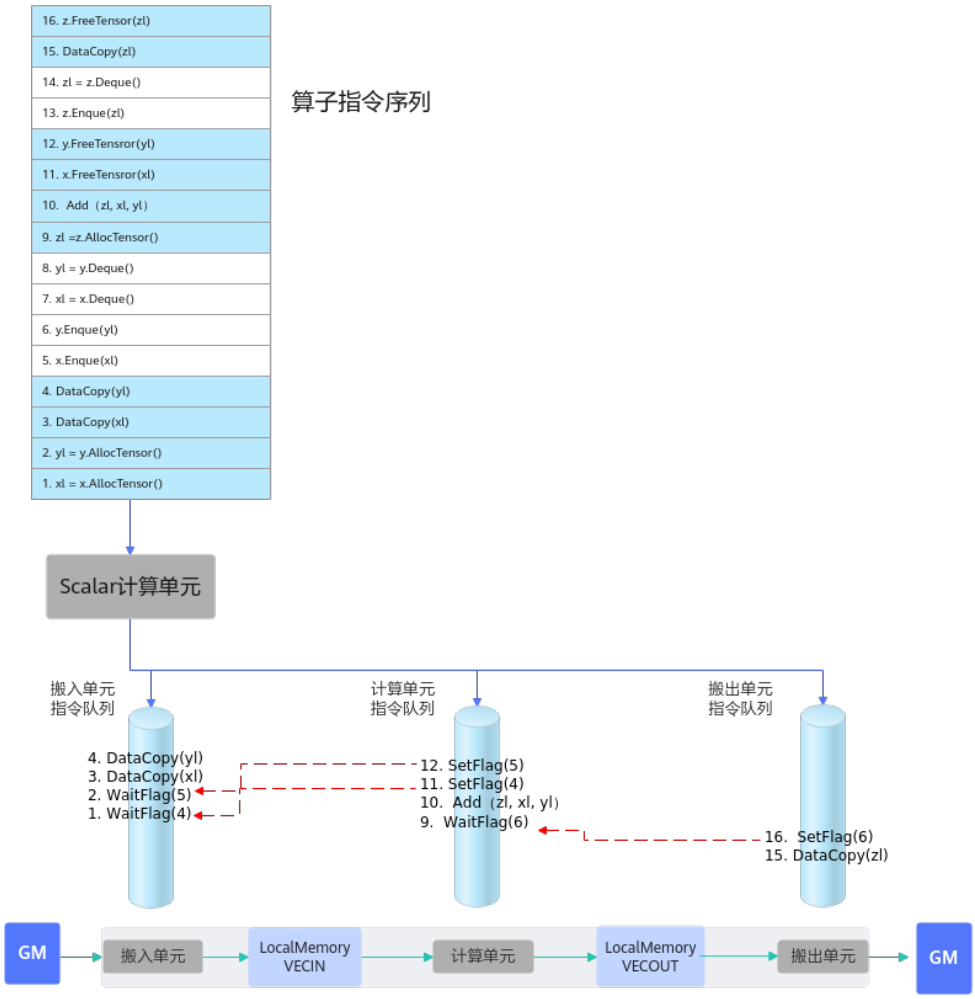


由此可以看出，EnQue/DeQue主要解决了存在数据依赖时，并行执行单元的写后读同步控制问题。

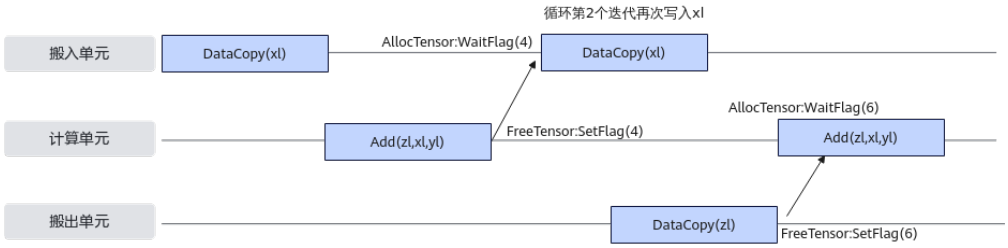


- AllocTensor/FreeTensor的具体处理流程
  a. 标量执行单元读取算子指令序列
  b. 把这些指令发射到对应的执行单元的指令队列
  c. 各个执行单元并行执行这些指令序列
  d. AllocTensor/FreeTensor，解决对内存的读后写问题
     - AllocTensor调用会发射同步指令wait等待内存被读完成
     - FreeTensor调用会发射同步指令set，通知内存释放，可以重复写

- wait需要等到set信号才能执行否则阻塞



由此可以看出，AllocTensor/FreeTensor主要解决了存在数据依赖时，并行执行单元的读后写同步控制问题。



通过上文的详细说明，可以看出异步并行程序需要考虑复杂的同步控制，而Ascend C编程模型将这些流程进行了封装，同时对外界面上通过EnQue/DeQue/AllocTensor/FreeTensor这种开发者熟悉的资源控制方式来体现，同时达到了简化编程和易于理解的目的。

6 编程 API

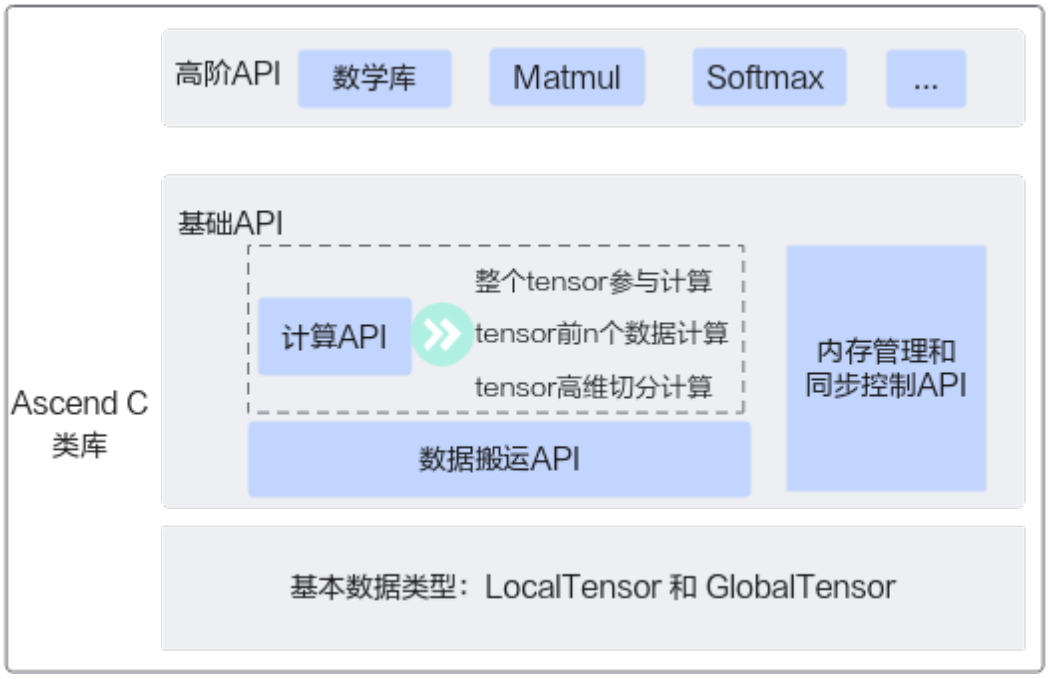
6.1 接口概述

Ascend C算子采用标准C++语法和一组类库API进行编程，您可以根据自己的需求选择合适的API。Ascend C编程类库API示意图如下所示，Ascend C API的操作数都是Tensor类型：GlobalTensor和LocalTensor；类库API分为基础API和高阶API。

基础API：实现对硬件能力的抽象，开放芯片的能力，保证完备性和兼容性。标注为ISASI（Instruction Set Architecture Special Interface，硬件体系结构相关的接口）类别的API，不能保证跨硬件版本兼容。

高阶API：实现一些常用的计算算法，用于提高编程开发效率，通常会调用多种基础API实现。高阶API包括数学库、Matmul、Softmax等API。高阶API可以保证兼容性。

图 6-1 Ascend C 编程类库 API 示意图



对于基础API，主要分为以下几类：

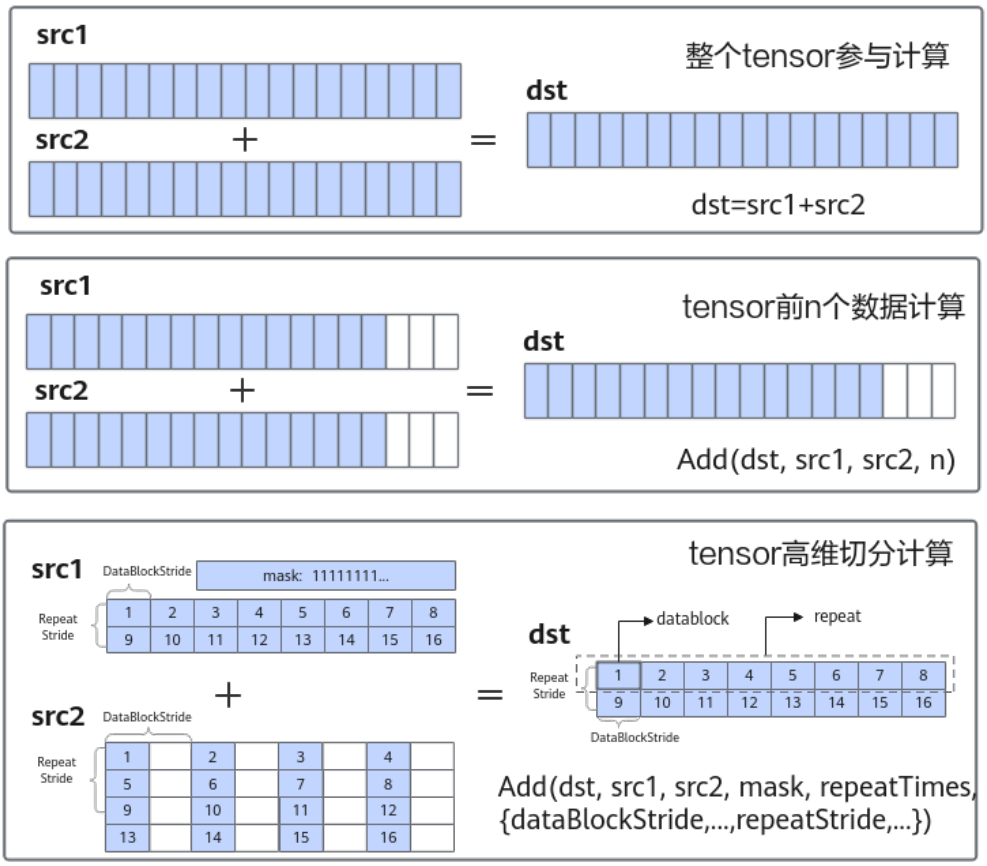
- **计算API**，包括标量计算API、向量计算API、矩阵计算API，分别实现调用Scalar计算单元、Vector计算单元、Cube计算单元执行计算的功能。
- **数据搬运API**，计算API基于Local Memory数据进行计算，所以数据需要先从Global Memory搬运至Local Memory，再使用计算API完成计算，最后从Local Memory搬出至Global Memory。执行搬运过程的接口称之为数据搬移API，比如DataCopy接口。
- **内存管理API**，用于分配管理内存，比如AllocTensor、FreeTensor接口。
- **同步控制API**，完成任务间的通信和同步，比如EnQue、DeQue接口。不同的API指令间有可能存在依赖关系，从**5.3 硬件架构抽象**可知，不同的指令异步并行执行，为了保证不同指令队列间的指令按照正确的逻辑关系执行，需要向不同的组件发送同步指令。同步控制API内部即完成这个发送同步指令的过程，开发者无需关注内部实现逻辑，使用简单的API接口即可完成。
- **ISASI（Instruction Set Architecture Special API），**硬件体系结构相关的API，该类接口不能保证跨硬件版本兼容。

对于基础API中的**计算API，**根据对数据操作方法的不同，分为以下**几种计算方式：**

- **整个tensor参与计算**：通过运算符重载的方式实现，支持“+, -, *, /, |, &, <, >, <=, >=, ==, !=”，实现计算的简化表达。例如：
  ```
  dst=src1+src2
  ```
- **tensor前n个数据计算**：针对源操作数的连续n个数据进行计算并连续写入目的操作数，解决一维tensor的连续计算问题。例如：
  ```
  Add(dst, src1, src2, n);
  ```
- **tensor高维切分计算：**功能灵活的计算API，充分发挥硬件优势，支持对每个操作数的dataBlockStride，repeatStride，mask等参数的操作。dataBlockStride，repeatStride，mask等参数的详细介绍请参见**6.2.1.1 通用参数说明**。

下图以矢量加法为例，展示了几种计算方式的特点。

图 6-2 计算 API 几种计算方式的特点



说明

Ascend C API所在头文件目录为：

- 基础API：*{install_path}/include*/ascendc/basic_api/interface
- 高阶API：（注意，如下目录头文件中包含的接口如果未在资料中声明，属于间接调用接口，开发者无需关注）
  - *{install_path}/include*/ascendc/highlevel_api/lib
  - *{install_path}*/include/tiling

其中{install_path}表示CANN软件安装目录。

6.2 基础 API

6.2.1 矢量计算

6.2.1.1 通用参数说明

说明

- 本章节对**矢量计算基础API**中的**tensor高维切分计算**接口做解释说明。如果您不需要使用此类接口，可略过该章节。
- 下文中的repeatTimes、dataBlockStride、repeatStride、mask为通用描述，其命名不一定与具体指令中的参数命名完全对应。

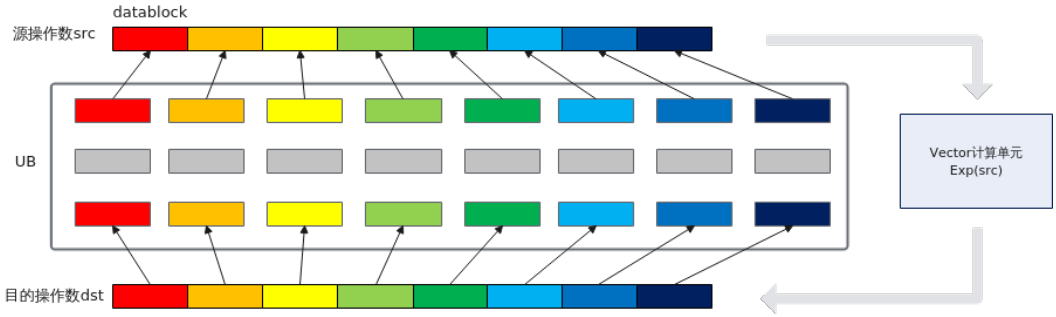
  比如，单次迭代内不同datablock间地址步长dataBlockStride参数，在单目指令中，对应为dstBlkStride、srcBlkStride参数；在双目指令中，对应为dstBlkStride、src0BlkStride、src1BlkStride参数。

  您可以在具体接口的参数说明中，找到参数含义的描述。

使用**tensor高维切分计算**API可充分发挥硬件优势，支持开发者控制指令的**迭代执行**和操作数的**地址间隔**，功能更加灵活。

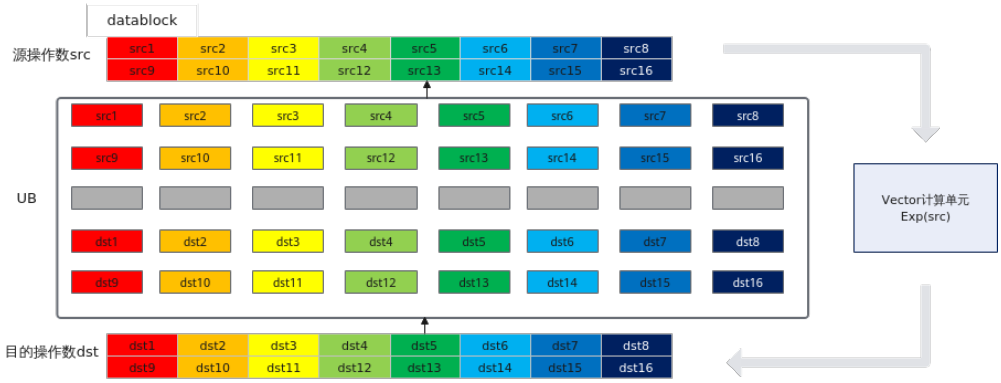
矢量计算通过Vector计算单元完成，矢量计算的源操作数和目的操作数均通过Unified Buffer（UB）来进行存储。Vector计算单元每个迭代会从UB中取出**8个datablock**（每个datablock数据块内部地址连续，长度32Byte），进行计算，并写入对应的8个datablock中。下图为单次迭代内的8个datablock进行Exp计算的示意图。

图 6-3 单次迭代内的 8 个 datablock 进行 Exp 计算示意图



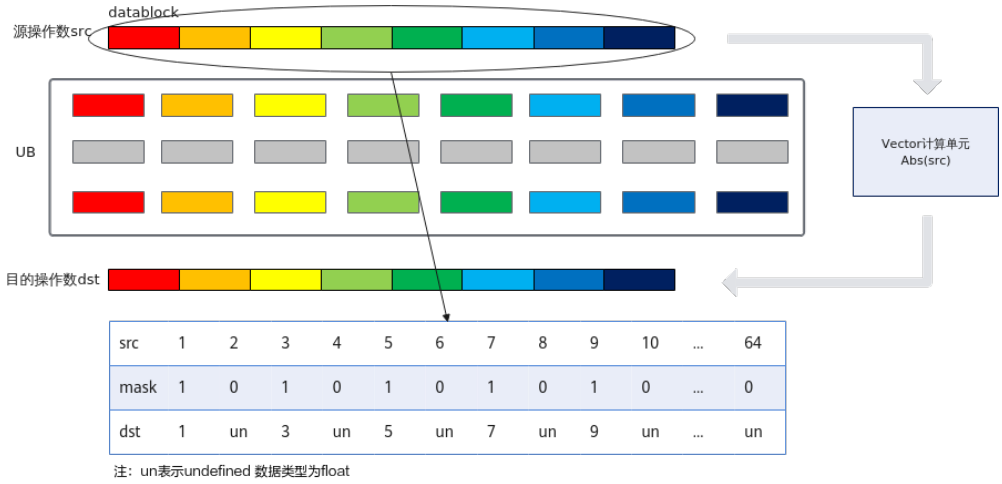
- 矢量计算API支持开发者通过**repeatTimes**来配置迭代次数，从而控制指令的多次迭代执行。假设**repeatTimes**设置为2，矢量计算单元会进行2个迭代的计算，可计算出2 * 8（每个迭代8个datablock） * 32Byte（每个datablock32Byte） = 512Byte的结果。如果数据类型为half，则计算了256个元素。下图展示了2次迭代Exp计算的示意图。**repeatTimes不能超过255。**

图 6-4 2 次迭代 Exp 计算



- 针对同一个迭代中的数据，可以通过**mask**参数进行掩码操作来控制实际参与计算的个数。下图为进行Abs计算时通过mask逐比特模式按位控制哪些元素参与计算的示意图，1表示参与计算，0表示不参与计算。

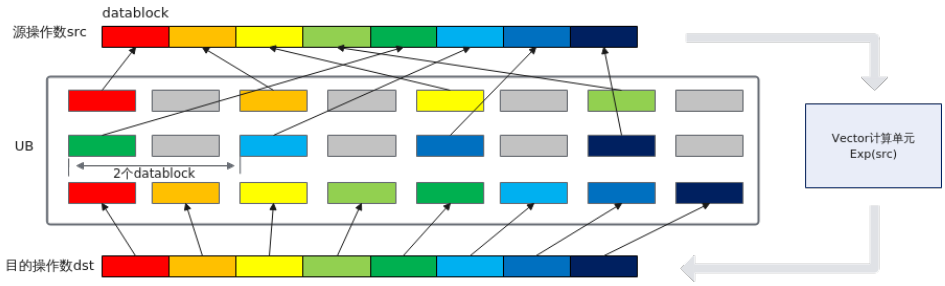
  图 6-5 通过 **mask** 参数进行掩码操作示意图



- 矢量计算单元还支持带间隔的向量计算，通过**dataBlockStride**（单次迭代内不同datablock间地址步长）和**repeatStride**（相邻迭代间相同datablock的地址步长）来进行配置。
  - **dataBlockStride**

    如果需要控制单次迭代内，数据处理的步长，可以通过设置同一迭代内不同datablock的地址步长dataBlockStride来实现。下图给出了单次迭代内非连续场景的示意图，示例中源操作数的dataBlockStride配置为2，表示单次迭代内不同datablock间地址步长(起始地址之间的间隔)为2个datablock。

    图 6-6 单次迭代内非连续场景的示意图

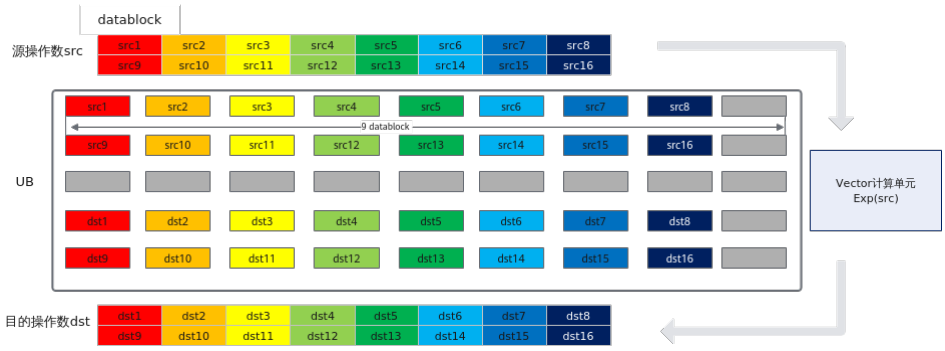


  - repeatStride

    当repeatTimes大于1，需要多次迭代完成矢量计算时，您可以根据不同的使用场景合理设置相邻迭代间相同datablock的地址步长repeatStride的值。

    下图给出了多次迭代间非连续场景的示意图，示例中源操作数和目的操作数的**repeatStride**均配置为9，表示相邻迭代间相同datablock起始地址之间的间隔为9个datablock。相同datablock是指datablock在迭代内的位置相同，比如下图中的src1和src9处于相邻迭代，在迭代内都是第一个datablock的位置，其间隔即为**repeatStride**的数值。

图 6-7 多次迭代间非连续场景的示意图

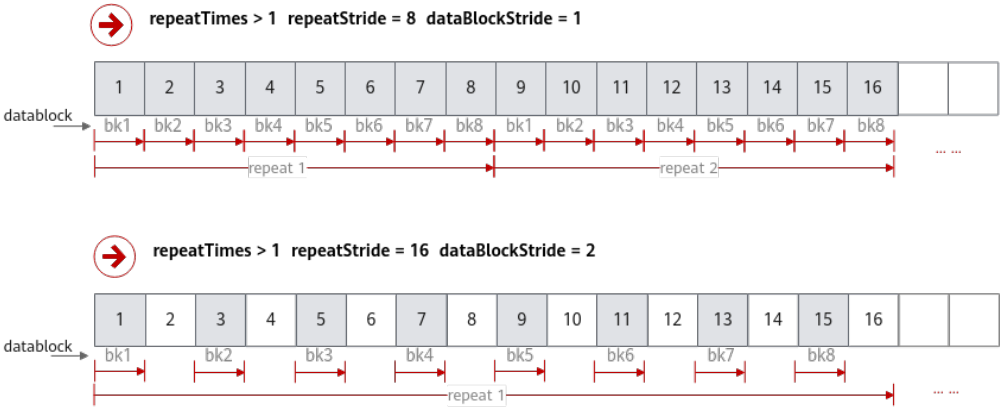


下文中给出了dataBlockStride、repeatStride、mask的详细配置说明和示例。

dataBlockStride（同一迭代内不同 datablock 的地址步长）

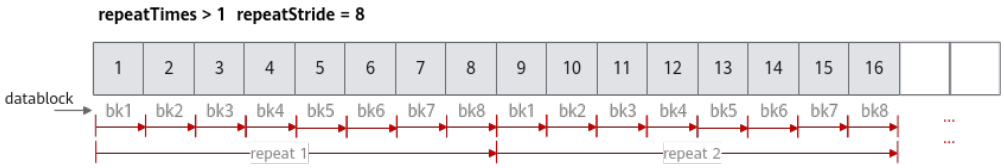
- 连续计算，dataBlockStride设置为1，对同一迭代内的8个datablock数据连续进行处理。
- 非连续计算，dataBlockStride值大于1（如取2），同一迭代内不同datablock之间在读取数据时出现一个datablock的间隔，如下图所示。

图 6-8 dataBlockStride 不同取值举例

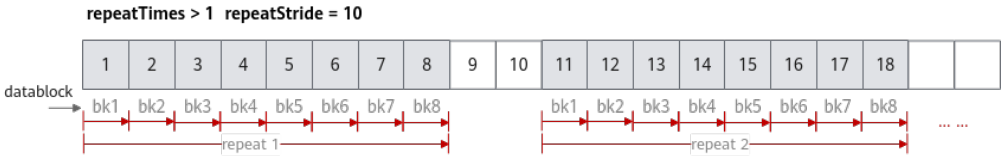


repeatStride（相邻迭代间相同 datablock 的地址步长）

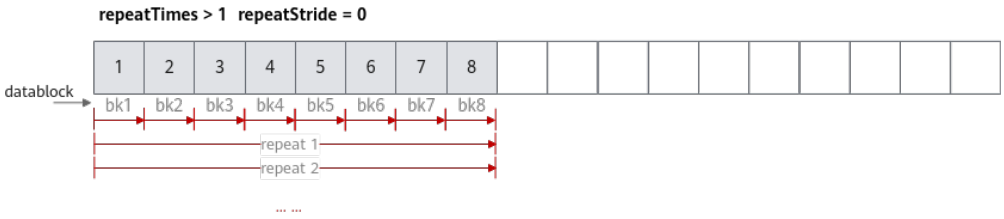
- **连续计算场景：**假设定义一个Tensor供目的操作数和源操作数同时使用（即地址重叠），repeatStride取值为8。此时，矢量计算单元第一次迭代读取连续8个datablock，第二轮迭代读取下一个连续的8个datablock，通过多次迭代即可完成所有输入数据的计算。



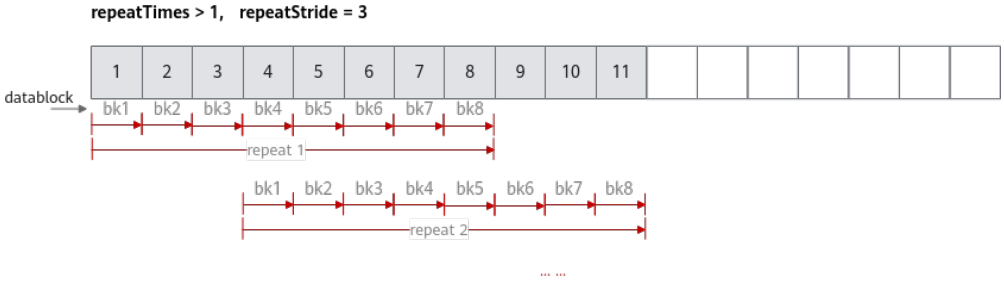
- **非连续计算场景：**repeatStride取值大于8（如取10）时，则相邻迭代间矢量计算单元读取的数据在地址上不连续，出现2个datablock的间隔。



- **反复计算场景：**repeatStride取值为0时，矢量计算单元会对首个连续的8个datablock进行反复读取和计算。



- **部分重复计算：**repeatStride取值大于0且小于8时，相邻迭代间部分数据会被矢量计算单元重复读取和计算，此种情形一般场景不涉及。



mask 参数

mask用于控制每次迭代内参与计算的元素。可通过连续模式和逐bit模式两种方式进行设置。

- 连续模式：表示前面连续的多少个元素参与计算。数据类型为uint64_t。取值范围和操作数的数据类型有关，数据类型不同，每次迭代内能够处理的元素个数最大值不同（当前数据类型单次迭代时能处理的元素个数最大值为：256 / sizeof(数据类型)）。当操作数的数据类型占bit位16位时（如half/uint16_t），mask∈[1, 128]；当操作数为32位时（如float/int32_t），mask∈[1, 64]。

  具体样例如下：

```
// int16_t数据类型单次迭代能处理的元素个数最大值为256/sizeof(int16_t) = 128，mask = 64，mask∈[1, 128]，所以是合法输入
// repeatTimes = 1, 共128个元素，单次迭代能处理128个元素，故repeatTimes = 1
// dstBlkStride, src0BlkStride, src1BlkStride = 1, 单次迭代内连续读取和写入数据
// dstRepStride, src0RepStride, src1RepStride = 8, 迭代间的数据连续读取和写入
uint64_t mask = 64;
AscendC::Add(dstLocal, src0Local, src1Local, mask, 1, { 1, 1, 1, 8, 8, 8 });
```

  结果示例如下：

```
输入数据(src0Local): [1 2 3 ... 64 ...128]
输入数据(src1Local): [1 2 3 ... 64 ...128]
输出数据(dstLocal): [2 4 6 ... 128 undefined...undefined]
```

```
// int32_t数据类型单次迭代能处理的元素个数最大值为256/sizeof(int32_t) = 64，mask = 64，mask∈[1, 64]，所以是合法输入
// repeatTimes = 1, 共64个元素，单次迭代能处理64个元素，故repeatTimes = 1
// dstBlkStride, src0BlkStride, src1BlkStride = 1, 单次迭代内连续读取和写入数据
// dstRepStride, src0RepStride, src1RepStride = 8, 迭代间的数据连续读取和写入
uint64_t mask = 64;
AscendC::Add(dstLocal, src0Local, src1Local, mask, 1, { 1, 1, 1, 8, 8, 8 });
```


结果示例如下：

```
输入数据(src0Local): [1 2 3 ... 64]
输入数据(src1Local): [1 2 3 ... 64]
输出数据(dstLocal): [2 4 6 ... 128]
```

- 逐bit模式：可以按位控制哪些元素参与计算，bit位的值为1表示参与计算，0表示不参与。参数类型为长度为2的uint64_t类型数组。

  参数取值范围和操作数的数据类型有关，数据类型不同，每次迭代内能够处理的元素个数最大值不同。当操作数为16位时，mask[0]、mask[1]∈[0, 2^{64}-1]，且mask[0]和mask[1]不可同时为0；当操作数为32位时，mask[1]为0，mask[0]∈（0, 2^{64}-1]。

  具体样例如下:

```
// 数据类型为int16_t
uint64_t mask[2] = {6148914691236517205, 6148914691236517205};
// repeatTimes = 1, 共128个元素，单次迭代能处理128个元素，故repeatTimes = 1。
// dstBlkStride, src0BlkStride, src1BlkStride = 1, 单次迭代内连续读取和写入数据。
// dstRepStride, src0RepStride, src1RepStride = 8, 迭代间的数据连续读取和写入。
AscendC::Add(dstLocal, src0Local, src1Local, mask, 1, { 1, 1, 1, 8, 8, 8 });
```

  结果示例如下：

```
输入数据(src0Local): [1 2 3 ... 64 ...127 128]
输入数据(src1Local): [1 2 3 ... 64 ...127 128]
输出数据(dstLocal): [2 un 6 ... un ...254 undefined]
```

  mask过程如下：

  mask={6148914691236517205, 6148914691236517205}（注：6148914691236517205表示64位二进制数0b010101....01，mask按照低位到高位的顺序排布）

src0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	64	...	127	128
src1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	64	...	127	128
mask	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	...	0	...	1	0
dst	2	un	6	un	10	un	14	un	18	un	22	un	26	un	30	un	34	un	38	un	...	un	...	254	un

注：un表示undefined

```
// 数据类型为int32_t
uint64_t mask[2] = {6148914691236517205, 0};
// repeatTimes = 1, 共64个元素，单次迭代能处理64个元素，故repeatTimes = 1。
// dstBlkStride, src0BlkStride, src1BlkStride = 1, 单次迭代内连续读取和写入数据。
// dstRepStride, src0RepStride, src1RepStride = 8, 迭代间的数据连续读取和写入。
AscendC::Add(dstLocal, src0Local, src1Local, mask, 1, { 1, 1, 1, 8, 8, 8 });
```

  结果示例如下：

```
输入数据(src0Local): [1 2 3 ... 63 64]
输入数据(src1Local): [1 2 3 ... 63 64]
输出数据(dstLocal): [2 un 6 ... 126 undefined]
```

  mask过程如下：

  mask={6148914691236517205, 0}（注：6148914691236517205表示64位二进制数0b010101....01）

src0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	63	64
src1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...	63	64
mask	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	...	1	0
dst	2	un	6	un	10	un	14	un	18	un	22	un	26	un	30	un	34	un	38	un	...	126	un

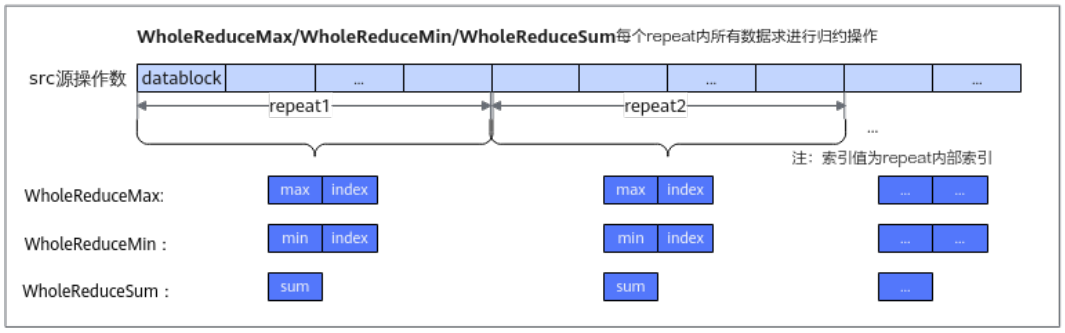
注：un表示undefined

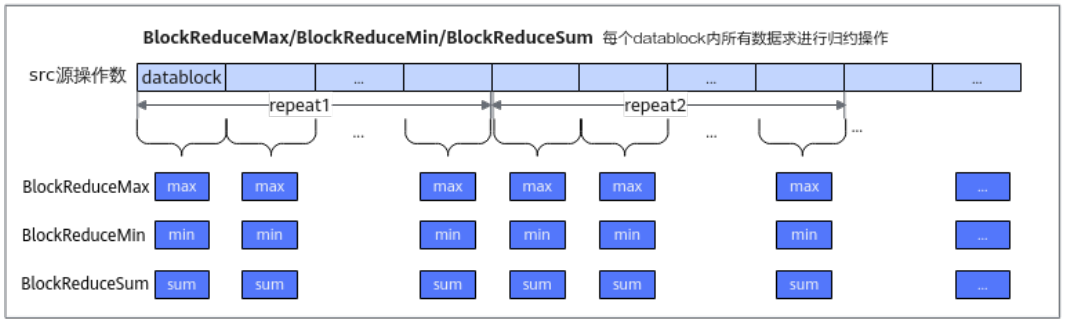
6.2.1.2 归约指令

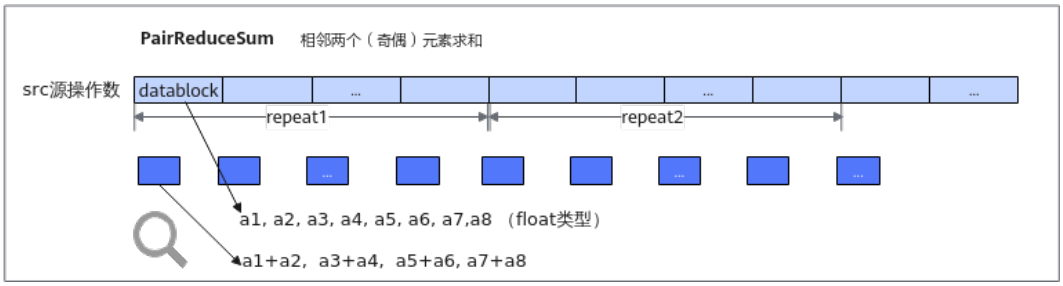
归约指令将数据集合简化为单一值或者更小的集合。按照归约操作的数据范围的不同，归约指令分为以下几种，可参考**归约指令示意图**：

- ReduceMax/ReduceMin/ReduceSum：对所有的输入数据做归约操作，得到最大值和最大值索引/最小值和最小值索引/数据总和。
- WholeReduceMax/WholeReduceMin/WholeReduceSum：对每个repeat内的输入数据做归约操作，得到每个repeat内的最大值和最大值索引/最小值和最小值索引/数据总和。返回索引时返回的是repeat内部索引。
- BlockReduceMax/BlockReduceMin/BlockReduceSum：对每个datablock内的输入数据做归约操作，得到每个datablock内的最大值/最小值/数据总和。
- PairReduce：相邻两个（奇偶）元素求和，例如（a1, a2, a3, a4, a5, a6...），归约后结果为（a1+a2, a3+a4, a5+a6, ... ）。

图 6-9 归约指令示意图







针对归约指令，和其他的基础API一样也提供了**tensor高维切分计算**接口，可充分发挥硬件优势，支持开发者控制指令的**迭代执行**和操作数的**地址间隔**，功能更加灵活。但具体参数的单位和约束与**基础API**略有不同，下文将对这些差异点进行介绍。

- **repeatTimes**：迭代次数，开发者通过repeatTimes来配置迭代次数，从而控制指令的多次迭代执行。
  - ReduceMax/ReduceMin/ReduceSum实际上是对WholeReduceMax/WholeReduceMin/WholeReduceSum的封装，对于repeatTimes超过255的情况，在API内部也进行了处理。所以repeatTimes支持更大的取值范围，保证不超过int32_t最大值的范围即可。
  - WholeReduceMax/WholeReduceMin/WholeReduceSum/BlockReduceMax/BlockReduceMin/BlockReduceSum/PairReduce和其他基础API一样，repeatTimes要求不超过255。
- **mask**：用于控制每次迭代内参与计算的元素，mask参数的使用方法和基础API通用的使用方法一致。
- **repeatStride**：表示相邻迭代间的地址步长。
  - ReduceMax/ReduceMin/ReduceSum指令的目的操作数会归约成一个最大值/最小值/总和，所以其目的操作数不支持配置repeatStride。仅源操作数支持repeatStride，其含义、单位（datablock）和基础API通用说明一致。
  - WholeReduceMax/WholeReduceMin/WholeReduceSum/BlockReduceMax/BlockReduceMin/BlockReduceSum/PairReduce源操作数和目的操作数都支持配置repeatStride，源操作数repeatStride的含义、单位（datablock）和基础API通用说明一致。目的操作数repeatStride的含义、单位和基础API通用说明有差异，因为归约后，目的操作数的长度会变短，比如WholeReduceSum归约后每个repeat会合并为一个值，所以迭代之间的间隔不能在使用一个datablock为单位，而以一个repeat归约后的长度为单位。
- **dataBlockStride**：表示单次迭代内datablock的地址步长。
  - ReduceMax/ReduceMin/ReduceSum指令的目的操作数会归约成一个最大值/最小值/总和，所以其目的操作数不支持配置dataBlockStride。源操作数也不支持dataBlockStride。
  - WholeReduceMax/WholeReduceMin/WholeReduceSum/BlockReduceMax/BlockReduceMin/BlockReduceSum/PairReduce源操作数支持配置dataBlockStride，源操作数dataBlockStride的含义、单位（datablock）和基础API通用说明一致。目的操作数不支持dataBlockStride，因为归约后，目的操作数的长度会变短，比如WholeReduceSum归约后每个repeat会合并为一个值，不再有迭代内datablock和地址间隔的概念。

6.2.1.3 掩码操作

掩码操作API用于开发者在矢量计算API外部手动配置Normal模式和Counter模式，并设置对应模式下的mask掩码。

什么是 Normal 模式和 Counter 模式

Normal模式为默认模式，支持开发者配置迭代次数，**通用说明**中介绍的配置方式为Normal模式。为了简化开发者操作，提供Counter模式，不需要开发者去感知迭代次数、处理非对齐尾块的操作，可直接传入计算数据量，实际迭代次数由Vector计算单元自动推断。

- Counter模式使用更方便，不需要计算尾块，但是不具备单次迭代内的mask能力这种高级功能；

- Normal模式具备单次迭代内的mask能力，但使用不如Counter模式方便，需要开发者感知迭代次数、额外进行尾块的计算。

默认情况下，高维切分计算API内部使用的是Normal模式，tensor前n个数据连续计算API内部使用的是Counter模式。

什么场景需要手动设置 Normal 模式和 Counter 模式

既然，高维切分计算API/tensor前n个数据连续计算API内部已经使用了Normal模式/Counter模式，为什么还需要开发者手动配置呢?

- Counter模式
  - 对于连续计算，如果数据量相同时，可以统一设置为Counter模式，并设置参与计算的数据量，无需在API内部反复设置，省去了在API反复设置的过程，会有一定的性能优势。
  - 使用高维切分计算API配套Counter模式使用时，比tensor前n个数据计算增加了可间隔的计算，支持dataBlockStride、repeatStride参数。
- Normal模式

  对于带mask参数（单次迭代内控制参与计算的个数）的非连续计算，如果mask参数相同，可以统一设置为Normal模式，并设置mask参数，无需在API内部反复设置，省去了在API反复设置的过程，会有一定的性能优势。

如何手动设置 Normal 模式

Normal模式的设置和使用流程如下：

1. 调用SetMaskNorm接口设置Normal模式。
2. 调用SetVectorMask接口设置mask掩码。Normal模式下，mask参数用来控制单次迭代内参与计算的元素个数。
3. 调用tensor高维切分计算API。
   - isSetMask模板参数需要设置为false，表示在矢量计算接口外部设置mask；接口入参中的mask值设置为MASK_PLACEHOLDER，用于占位，无实际含义。
   - Normal模式下，根据使用场景正确配置repeatTimes、dataBlockStride、repeatStride参数。
4. 使用ResetMask恢复mask的值为默认值。

如何手动设置 Counter 模式

1. 调用SetMaskCount接口设置Counter模式。
2. 调用SetVectorMask接口设置mask掩码。Counter模式下，mask参数表示整个矢量计算参与计算的元素个数。
3. 调用矢量计算API
   - 针对tensor高维切分计算API。
     - isSetMask模板参数需要设置为false，表示在矢量计算接口外部设置mask；接口入参中的mask值设置为MASK_PLACEHOLDER，用于占位，无实际含义。
     - 根据使用场景正确配置dataBlockStride、repeatStride参数。repeatTimes传入固定值即可，建议统一设置为1，该值不生效。

  - 针对tensor前n个数据计算API（当前仅部分API支持isSetMask模板参数）。isSetMask模板参数需要设置为false，表示在矢量计算接口外部设置mask；接口入参中的calCount不生效，建议设置成1。
4. 设置为Counter模式的场景需要在矢量计算使用完之后调用SetMaskNorm将mask模式恢复为Normal模式。
5. 使用ResetMask恢复mask的值为默认值。

6.2.2 数据搬运

数据搬运接口，包括普通数据搬运、增强数据搬运、切片数据搬运、随路格式转换。

- 普通数据搬运接口，适用于连续和不连续数据搬运。
- 增强数据搬运接口，相比于普通数据搬运接口，在CO1->CO2通路时，通过设置DataCopyEnhancedParams参数增加随路计算，其他通路情况下，DataCopyEnhancedParams不生效，等同于普通数据搬运的功能。
- 切片数据搬运接口，主要适用于非连续vector数据搬运。
- 随路格式转换接口，适用于在搬运时进行格式转换。

6.2.3 内存管理与同步控制

Ascend C编程范式，把算子核内的处理程序，分成多个流水任务，通过队列（Queue）完成**任务间通信和同步**，并通过统一的**资源管理**模块（Pipe）来统一管理内存、事件等资源。

Ascend C提供一组内存管理与同步控制API，开发者使用这一组API即可完成任务间同步和内存管理。

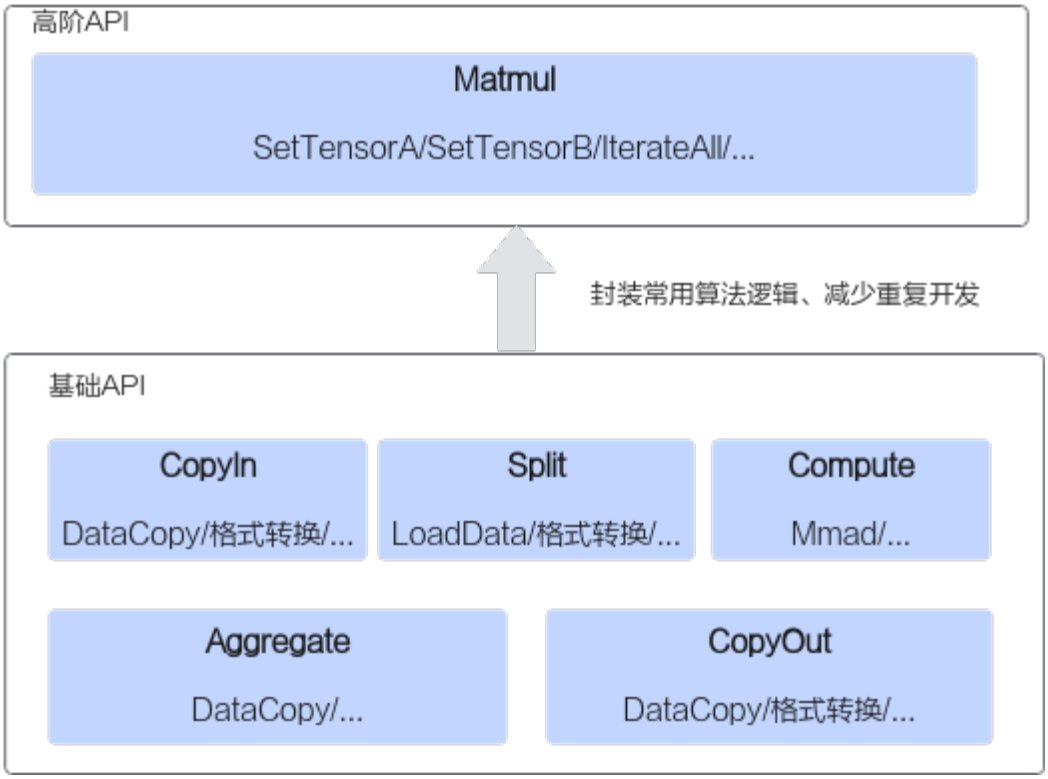
核心的API包括：

- AllocTensor：从Queue中分配Tensor，Tensor所占大小为InitBuffer时设置的每块内存长度。
- FreeTensor：释放Queue中的指定Tensor，供Queue后续使用。
- EnQue：将Tensor push到队列Queue。
- DeQue：将Tensor从队列Queue中取出，用于后续处理。

6.3 高阶 API

高阶API一般是基于单核对常见算法的抽象和封装，用于提高编程开发效率，通常会调用多种基础API实现。高阶API包括数学库、Matmul、Softmax等API。

如下图所示，实现一个矩阵乘操作，使用基础API需要的步骤较多，需要关注格式转换、数据切分等逻辑；使用高阶API则无需关注这些逻辑，直接传入输入矩阵，调用接口获取输出即可。



注意，在程序中调用高阶API的Tiling接口或者使用高阶API的Tiling结构体参数时，需要引入依赖的头文件，示例如下：

```
#include "tiling/tiling_api.h"
```


7 算子实现

7.1 概述

Ascend C的算子实现主要包含两个部分：

- Host侧Tiling实现

  由于NPU中AI Core内部存储无法完全容纳算子输入输出的所有数据，需要每次搬运一部分输入数据进行计算然后搬出，再搬运下一部分输入数据进行计算，这个过程就称之为Tiling。切分数据的算法称为Tiling算法或者Tiling策略。根据算子的shape等信息来确定数据切分算法相关参数（比如每次搬运的块大小，以及总共循环多少次）的计算程序，称之为Tiling实现，也叫Tiling函数（Tiling Function）。由于Tiling实现中完成的均为标量计算，AI Core并不擅长，所以我们将其独立出来放在Host侧CPU上执行。

- Device侧Kernel实现

  Kernel实现即算子核函数实现，在Kernel函数内部通过解析Host侧传入的Tiling结构体获取Tiling信息，根据Tiling信息控制数据搬入搬出Local Memory的流程；通过调用计算、数据搬运、内存管理、任务同步API，实现算子逻辑。其核心逻辑基本上都为计算密集型任务，需要在NPU上执行。

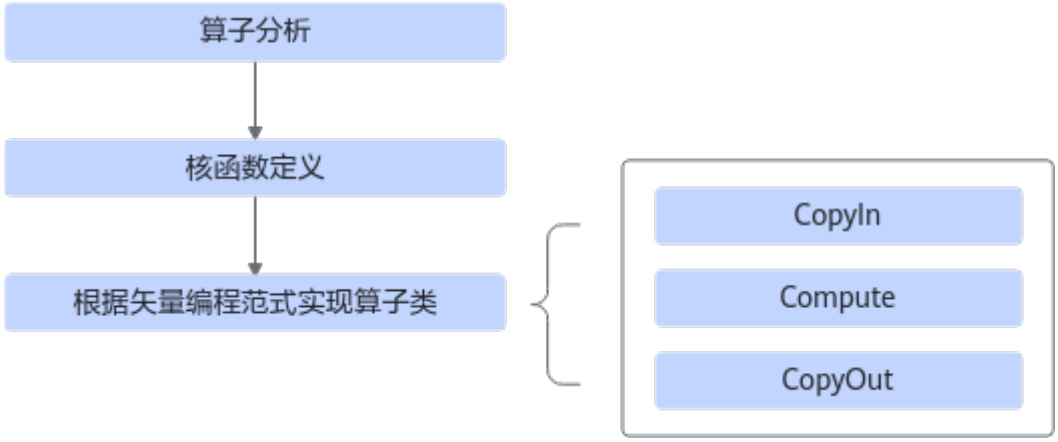
本章介绍了矢量编程、矩阵编程、融合算子编程三种典型场景下的算子Tiling、Kernel实现，是对上文中三种典型编程范式的具体应用，同时也介绍了编程的更多细节、API的使用方法等。

7.2 矢量编程

7.2.1 算子实现

基于Ascend C方式实现矢量算子的流程如下图图7-1所示。

图 7-1 矢量算子实现流程



- 算子分析：分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。
- 核函数定义：定义Ascend C算子入口函数。
- 根据矢量编程范式实现算子类：完成核函数的内部实现。

下文以ElemWise(Add)算子为例，对上述步骤进行详细介绍。本样例中介绍的算子完整代码参见add_custom.cpp。

算子分析

在开发算子代码之前需要分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。

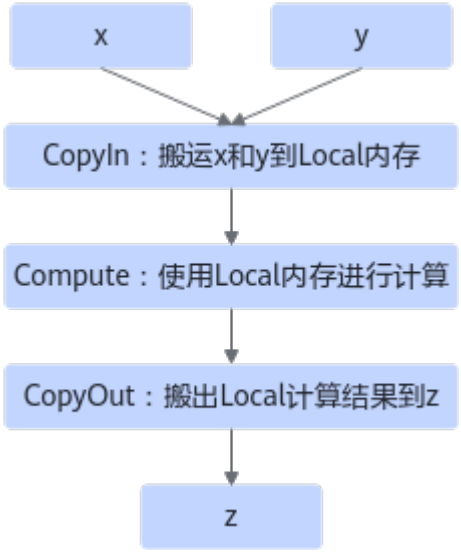
步骤1 明确算子的数学表达式及计算逻辑。

Add算子的数学表达式为：

```
z = x + y
```

计算逻辑是：Ascend C提供的矢量计算接口的操作元素都为LocalTensor，输入数据需要先搬运进片上存储，然后使用计算接口完成两个输入参数相加，得到最终结果，再搬出到外部存储上。Ascend C Add算子的计算逻辑如下图所示。

图 7-2 算子计算逻辑



步骤2 明确输入和输出。

- Add算子有两个输入：x与y，输出为z。
- 本样例中算子的输入支持的数据类型为half（float16），算子输出的数据类型与输入数据类型相同。
- 算子输入支持shape（8，2048），输出shape与输入shape相同。
- 算子输入支持的format为：ND。

步骤3 确定核函数名称和参数。

- 您可以自定义核函数名称，本样例中核函数命名为add_custom。
- 根据对算子输入输出的分析，确定核函数有3个参数x，y，z；x，y为输入在Global Memory上的内存地址，z为输出在Global Memory上的内存地址。

步骤4 确定算子实现所需接口。

- 实现涉及外部存储和内部存储间的数据搬运，查看Ascend C API参考中的数据搬移接口，需要使用DataCopy来实现数据搬移。
- 本样例只涉及矢量计算的加法操作，查看Ascend C API参考中的矢量计算接口矢量计算，初步分析可使用双目指令Add接口Add实现x+y。
- 计算中使用到的Tensor数据结构，使用Queue队列进行管理，会使用到EnQue、DeQue等接口。

----结束

通过以上分析，得到Ascend C Add算子的设计规格如下：

表 7-1 Ascend C Add 算子设计规格

算子类型（OpType）	Add			
算子输入输出	**name**	**shape**	**data type**	**format**
	x（输入）	(8, 2048)	half	ND

	y（输入）	(8, 2048)	half	ND
	z（输出）	(8, 2048)	half	ND
核函数名称	add_custom			
使用的主要接口	DataCopy：数据搬移接口			
	Add：矢量双目指令接口			
	EnQue、DeQue等接口：Queue队列管理接口			
算子实现文件名称	add_custom.cpp			

核函数定义

根据**核函数定义**中介绍的规则进行核函数的定义。

步骤1 函数原型定义

本样例中，函数名为add_custom（核函数名称可自定义），根据**算子分析**中对算子输入输出的分析，确定有3个参数x，y，z，其中x，y为输入内存，z为输出内存。根据**核函数定义**核函数的规则介绍，函数原型定义如下所示：使用__global__函数类型限定符来标识它是一个核函数，可以被<<<>>>调用；使用__aicore__函数类型限定符来标识该核函数在设备端aicore上执行；为方便起见，统一使用GM_ADDR宏修饰入参，GM_ADDR宏定义请参考**核函数定义**。

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
}
```

步骤2 调用算子类的Init和Process函数。

算子类的Init函数，完成内存初始化相关工作，Process函数完成算子实现的核心逻辑，具体介绍参见**算子类实现**。

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

步骤3 对核函数的调用进行封装，得到add_custom_do函数，便于主程序调用。#ifndef ASCENDC_CPU_DEBUG表示该封装函数仅在编译运行NPU侧的算子时会用到，编译运行CPU侧的算子时，可以直接调用add_custom函数。根据**核函数调用**章节，调用核函数时，除了需要传入参数x，y，z，还需要传入blockDim（核函数执行的核数），l2ctrl（保留参数，设置为nullptr），stream（应用程序中维护异步操作执行顺序的stream）来规定核函数的执行配置。

```
#ifndef ASCENDC_CPU_DEBUG
// call of kernel function
void add_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* x, uint8_t* y, uint8_t* z)
{
    add_custom<<<blockDim, l2ctrl, stream>>>(x, y, z);
}
#endif
```

----结束

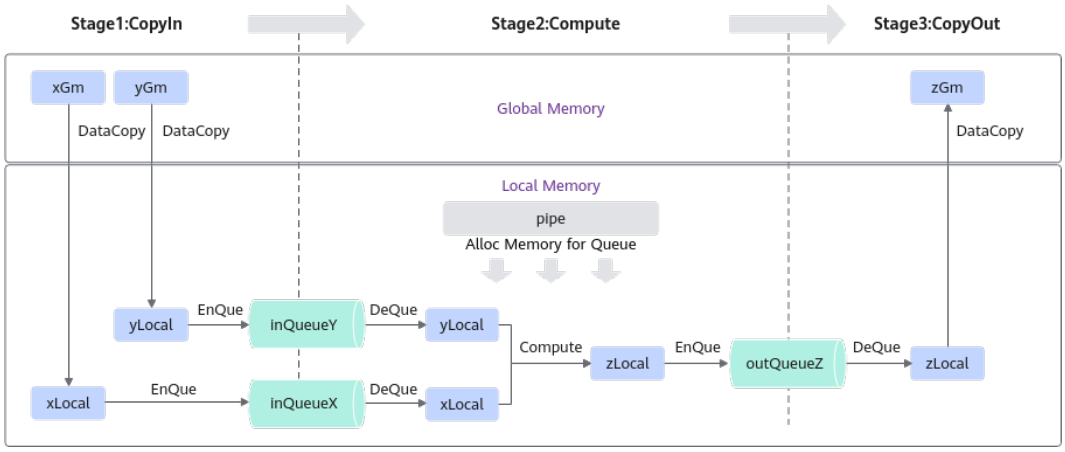
算子类实现

根据上一节介绍，核函数中会调用算子类的Init和Process函数，本节具体讲解如何基于编程范式实现算子类。

根据矢量编程范式对Add算子的实现流程进行设计的思路如下，矢量编程范式请参考**Vector编程范式**，设计完成后得到的Add算子实现流程图参见**图7-3**：

- Add算子的实现流程分为3个基本任务：CopyIn，Compute，CopyOut。CopyIn任务负责将Global Memory上的输入Tensor xGm和yGm搬运至Local Memory，分别存储在xLocal，yLocal，Compute任务负责对xLocal，yLocal执行加法操作，计算结果存储在zLocal中，CopyOut任务负责将输出数据从zLocal搬运至Global Memory上的输出Tensor zGm中。
- CopyIn，Compute任务间通过VECIN队列inQueueX，inQueueY进行通信和同步，Compute，CopyOut任务间通过VECOUT队列outQueueZ进行通信和同步。
- 任务间交互使用到的内存、临时变量使用到的内存统一使用pipe内存管理对象进行管理。

图 7-3 Add 算子实现流程



算子类中主要实现上述流程，包含对外开放的初始化Init函数和核心处理函数Process，Process函数中会对上图中的三个基本任务进行调用；同时包括一些算子实现中会用到的私有成员，比如上图中的Global Tensor和VECIN、VECOUT队列等。KernelAdd算子类具体成员如下：

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd() {}
    // 初始化函数，完成内存初始化相关操作
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z){}
    // 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成矢量算子的三级流水操作
    __aicore__ inline void Process(){}

private:
    // 搬入函数，完成CopyIn阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyIn(int32_t progress){}
    // 计算函数，完成Compute阶段的处理，被核心Process函数调用
    __aicore__ inline void Compute(int32_t progress){}
    // 搬出函数，完成CopyOut阶段的处理，被核心Process函数调用
    __aicore__ inline void CopyOut(int32_t progress){}

private:
```



```
    AscendC::TPipe pipe;  //Pipe内存管理对象
    AscendC::TQue<AscendC::QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;  //输入数据Queue队列管理对象，QuePosition为VECIN
    AscendC::TQue<AscendC::QuePosition::VECOUT, BUFFER_NUM> outQueueZ;  //输出数据Queue队列管理对象，QuePosition为VECOUT
    AscendC::GlobalTensor<half> xGm;  //管理输入输出Global Memory内存地址的对象，其中xGm, yGm为输入，zGm为输出
    AscendC::GlobalTensor<half> yGm;
    AscendC::GlobalTensor<half> zGm;
};
```

初始化函数主要完成以下内容：

- 设置输入输出Global Tensor的Global Memory内存地址。

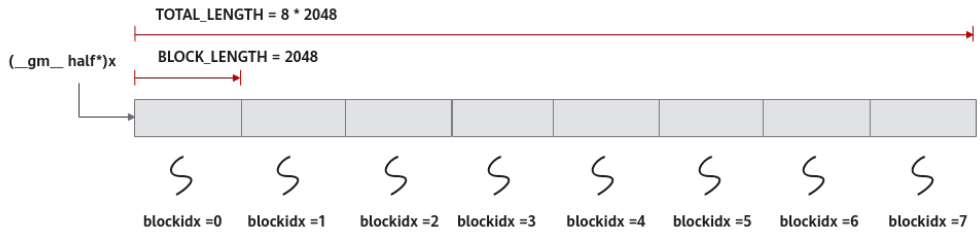
  本样例中使用多核并行计算，即把数据进行分片，分配到多个核上进行处理。Ascend C核函数是在一个核上的处理函数，所以只处理部分数据，需要在初始化函数中获取该核函数需要处理的输入输出在Global Memory上的内存偏移地址，并将该偏移地址设置在Global Tensor中。

  以获取输入x在Global Memory上的内存偏移地址为例：

  ```
  xGm.SetGlobalBuffer((__gm__ half*)x + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
  ```

  本样例中的分配方案是：数据整体长度TOTAL_LENGTH为8 * 2048，平均分配到8个核上运行，每个核上处理的数据大小BLOCK_LENGTH为2048。x + BLOCK_LENGTH * GetBlockIdx()即为单核处理程序中x在Global Memory上的内存偏移地址，获取偏移地址后，使用GlobalTensor类的SetGlobalBuffer接口设定该核上Global Memory的起始地址以及长度。具体示意图请参考图7-4。

  图 7-4 多核并行处理示意图

- 通过Pipe内存管理对象为输入输出Queue分配内存。

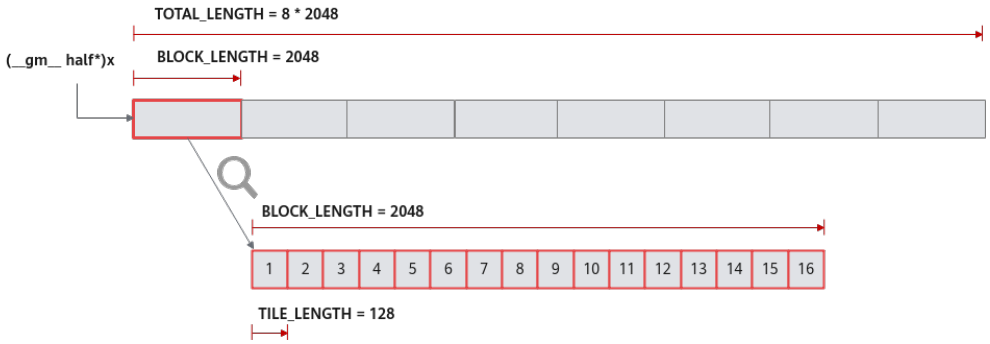
  比如，为输入x的Queue分配内存，可以通过如下代码段实现：

  ```
  pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half))
  ```

  对于单核上的处理数据，可以进行数据切块（Tiling），在本示例中，仅作为参考，将数据切分成8块（并不意味着8块就是性能最优）。切分后的每个数据块再次切分成2块，即可开启double buffer，实现流水线之间的并行。

  这样单核上的数据（2048个数）被切分成16块，每块TILE_LENGTH（128）个数据。上文代码表示Pipe为inQueueX分配了两块大小为TILE_LENGTH * sizeof(half)个字节的内存块，每个内存块能容纳TILE_LENGTH（128）个half类型数据。数据切分示意图如图7-5所示。

图 7-5 单核数据切分示意图



具体的初始化函数代码如下：

```
constexpr int32_t TOTAL_LENGTH = 8 * 2048;                            // total length of data
constexpr int32_t USE_CORE_NUM = 8;                                   // num of core used
constexpr int32_t BLOCK_LENGTH = TOTAL_LENGTH / USE_CORE_NUM;         // length computed of each core
constexpr int32_t TILE_NUM = 8;                                       // split data into 8 tiles for each core
constexpr int32_t BUFFER_NUM = 2;                                     // tensor num for each queue
constexpr int32_t TILE_LENGTH = BLOCK_LENGTH / TILE_NUM / BUFFER_NUM; // separate to 2 parts, due to double buffer
__aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
    // get start index for current core, core parallel
    xGm.SetGlobalBuffer((__gm__ half*)x + BLOCK_LENGTH * AscendC::GetBlockIdx(), BLOCK_LENGTH);
    yGm.SetGlobalBuffer((__gm__ half*)y + BLOCK_LENGTH * AscendC::GetBlockIdx(), BLOCK_LENGTH);
    zGm.SetGlobalBuffer((__gm__ half*)z + BLOCK_LENGTH * AscendC::GetBlockIdx(), BLOCK_LENGTH);
    // pipe alloc memory to queue, the unit is Bytes
    pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(inQueueY, BUFFER_NUM, TILE_LENGTH * sizeof(half));
    pipe.InitBuffer(outQueueZ, BUFFER_NUM, TILE_LENGTH * sizeof(half));
}
```

基于矢量编程范式，将核函数的实现分为3个基本任务：CopyIn，Compute，CopyOut。Process函数中通过如下方式调用这三个函数。

```
__aicore__ inline void Process()
{
    // loop count need to be doubled, due to double buffer
    constexpr int32_t loopCount = TILE_NUM * BUFFER_NUM;
    // tiling strategy, pipeline parallel
    for (int32_t i = 0; i < loopCount; i++) {
        CopyIn(i);
        Compute(i);
        CopyOut(i);
    }
}
```

根据编程范式上面的算法分析，将整个计算拆分成三个Stage，用户单独编写每个Stage的代码，三阶段流程示意图参见图7-3，具体流程如下：

步骤1 Stage1：CopyIn函数实现。

1. 使用DataCopy接口将GlobalTensor数据拷贝到LocalTensor。
2. 使用EnQue将LocalTensor放入VecIn的Queue中。

```
__aicore__ inline void CopyIn(int32_t progress)
{
    // alloc tensor from queue memory
    AscendC::LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
    AscendC::LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
    // copy progress_th tile from global tensor to local tensor
    AscendC::DataCopy(xLocal, xGm[progress * TILE_LENGTH], TILE_LENGTH);
```

```
    AscendC::DataCopy(yLocal, yGm[progress * TILE_LENGTH], TILE_LENGTH);
    // enque input tensors to VECIN queue
    inQueueX.EnQue(xLocal);
    inQueueY.EnQue(yLocal);
}
```

步骤2 Stage2：Compute函数实现。

1. 使用DeQue从VecIn中取出LocalTensor。
2. 使用Ascend C接口Add完成矢量计算。
3. 使用EnQue将计算结果LocalTensor放入到VecOut的Queue中。
4. 使用FreeTensor释放不再使用的LocalTensor。

```
__aicore__ inline void Compute(int32_t progress)
{
    // deque input tensors from VECIN queue
    AscendC::LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    AscendC::LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    AscendC::LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    // call Add instr for computation
    AscendC::Add(zLocal, xLocal, yLocal, TILE_LENGTH);
    // enque the output tensor to VECOUT queue
    outQueueZ.EnQue<half>(zLocal);
    // free input tensors for reuse
    inQueueX.FreeTensor(xLocal);
    inQueueY.FreeTensor(yLocal);
}
```

步骤3 Stage3：CopyOut函数实现。

1. 使用DeQue接口从VecOut的Queue中取出LocalTensor。
2. 使用DataCopy接口将LocalTensor拷贝到GlobalTensor上。
3. 使用FreeTensor将不再使用的LocalTensor进行回收。

```
__aicore__ inline void CopyOut(int32_t progress)
{
    // deque output tensor from VECOUT queue
    AscendC::LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
    // copy progress_th tile from local tensor to global tensor
    AscendC::DataCopy(zGm[progress * TILE_LENGTH], zLocal, TILE_LENGTH);
    // free output tensor for reuse
    outQueueZ.FreeTensor(zLocal);
}
```

----结束

运行验证

核函数即算子kernel程序开发完成后，即可编写host侧的核函数调用程序，实现从host侧的APP程序调用算子，进行运行验证。包括CPU侧和NPU侧两种运行验证方法：

- **CPU侧运行验证**主要通过ICPU_RUN_KF CPU调测宏等CPU调测库提供的接口来完成；
- **NPU侧运行验证**主要通过使用Kernel Launch接口或者<<<>>>内核调用符，以及AscendCL API提供的运行时接口来完成。

具体步骤请参考**8.2.1 核函数运行验证简介**。

7.2.2 非对齐场景

数据搬运和 Vector 计算的对齐要求

须知

进行数据搬运和Vector计算时，对于搬运的数据长度和操作数的起始地址有如下的对齐要求：

- 使用DataCopy接口进行数据搬运，搬运的数据长度和操作数的起始地址必须保证32字节对齐。
- 进行Vector计算时，操作数的起始地址必须保证32字节对齐。

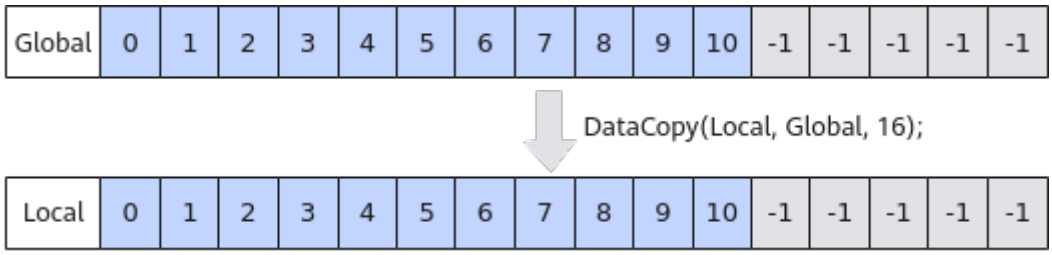
下文描述中的Global代指Global Memory，Local代指Local Memory。

下面是一些非对齐搬运和计算的例子。

- **非对齐搬入**

  当需要从Global拷贝11个half数值到Local时，为保证搬运的数据长度32字节对齐，使用DataCopy将拷贝16个half(32B)数据到Local上，Local[11]~Local[15]被写成无效数据-1。

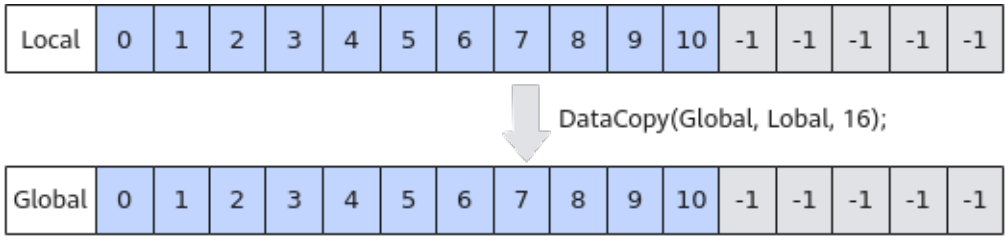
  图 7-6 非对齐搬入内存

- **非对齐搬出**

  当需要从Local拷贝11个half数值到Global时，为保证搬运的数据长度32字节对齐，使用DataCopy将拷贝16个half(32B)数据到Global上，Global[11]~Global[15]被写成无效数据-1。

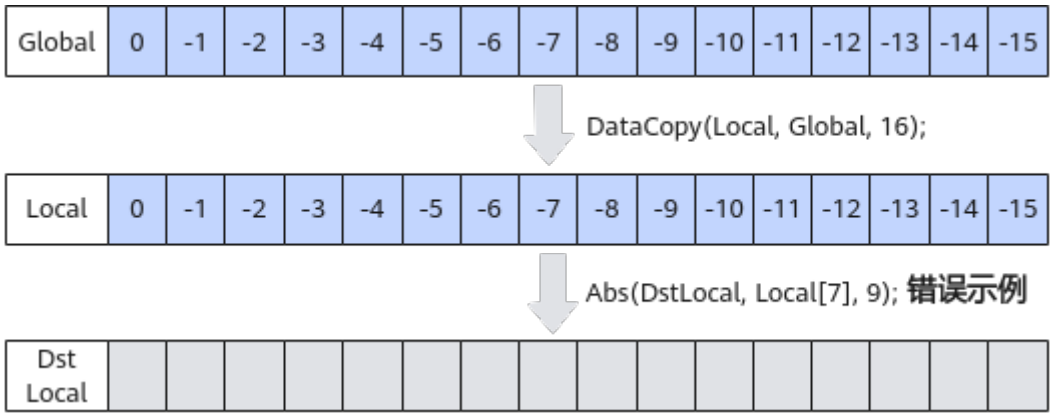
  图 7-7 非对齐搬出内存

- **矢量计算起始地址非32字节对齐的错误示例**

  矢量计算时需要保证起始地址32字节对齐，如下的示例中，从Local[7]，即LocalTensor的第8个数开始计算，起始地址不满足32字节对齐，是错误示例。

图 7-8 矢量计算起始地址非 32 字节对齐的错误示例



非对齐处理方案

DataCopyPad接口提供非对齐搬运的功能，如果基于该接口支持的产品开发算子（参见支持的型号），则可以直接使用该接口解决非对齐场景下的搬运问题。但部分型号不支持DataCopyPad接口，则需要参考如下的方案处理。

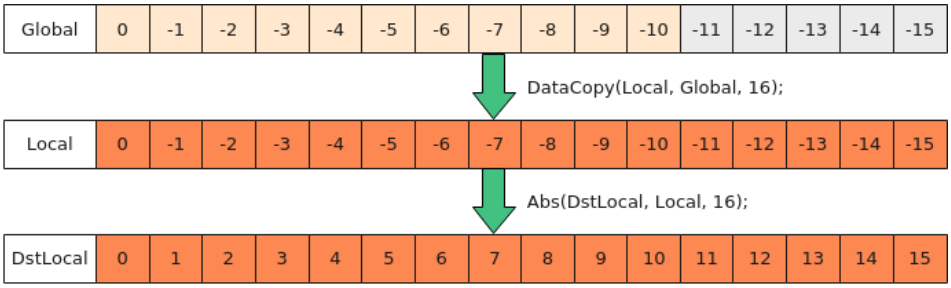
-
- Global上逐行搬运长度不对齐数据到Local中，导致Local中每行都存在冗余数据
  - 冗余数据参与计算

    如下图所示，将前11个half数据进行Abs计算，冗余数据可以参与计算，不影响最终结果，该种方式主要用于elemwise计算，这里步骤为：

    i. 使用DataCopy搬运16个half数据到Local中；

    ii. 直接使用Abs做整块计算，可以不用计算尾块大小。

    图 7-9 冗余数据参与计算

  - 使用mask掩掉脏数据，一般用于轴归约计算等

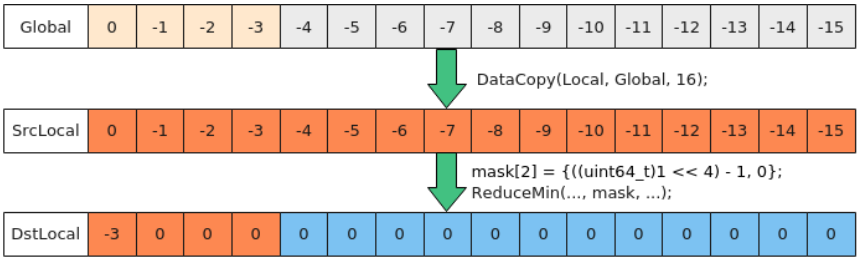
    如下图所示，假设输入数据的Shape为16*4，将输入数据搬入到UB后每行数据前4个half数据为有效数据，其余为冗余数据。为只对前4个half数据进行ReduceMin计算，且保证冗余数据不参与到计算，可以通过设置ReduceMin API的mask参数的方法掩盖掉脏数据。这里步骤为：

    i. 使用DataCopy搬运16个half数据到Local中；

    ii. 对归约计算的目的操作数DstLocal清零，如使用Duplicate等；

    iii. 进行归约操作，将ReduceMin的mask模式设置为前4个数据有效，来掩掉对冗余数据区域的处理。

图 7-10 冗余数据不参与计算



- 搬入Local中，逐行调用基础API Duplicate，脏数据位置填充0值

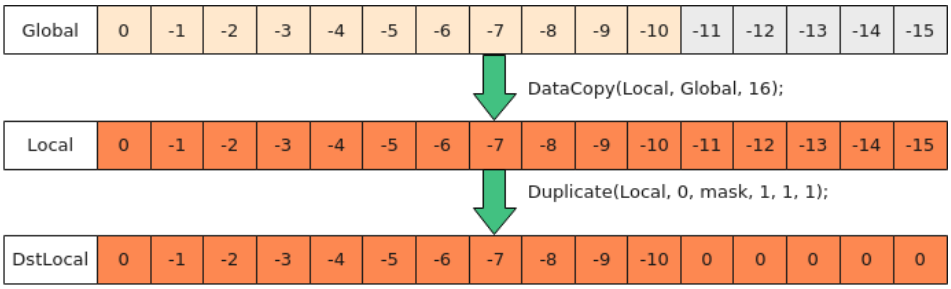
  如下图所示，对于搬入后的非对齐数据，逐行进行Duplicate清零处理，步骤为：

  i. 使用DataCopy搬运16个half数据到Local；

  ii. 使用高阶API Duplicate，按照如下方式设置mask值，控制仅后5个元素位置有效，将冗余数据填充为0。

```
uint64_t mask0 = ((uint64_t)1 << 16) - ((uint64_t)1 << 11);
uint64_t mask[2] = {mask0, 0};
```

图 7-11 逐行填充 0 值



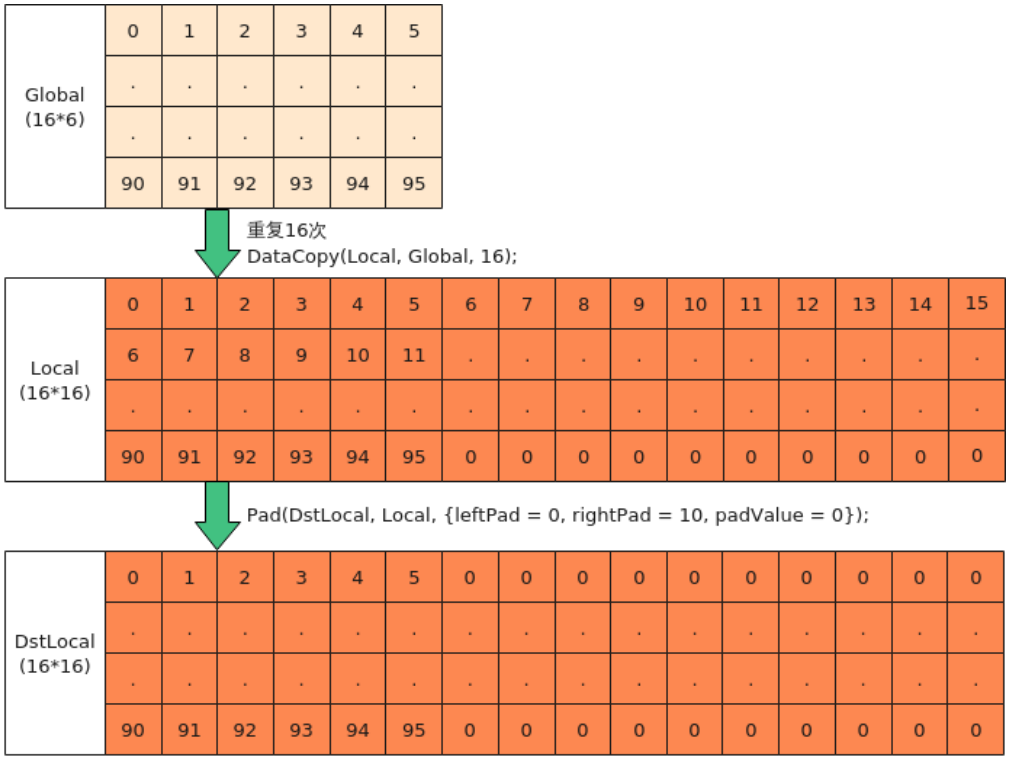
- Global搬运非对齐数据到Local， 逐行搬入后，Pad成0值

  如下图所示，将Local内16*16大小的数据块进行脏数据清零，逐行清零性能会很差，可以使用Pad一次性清零，步骤为：

  a. 将16*6的数据从GM上逐行搬入UB后，每行6个有效数据；

  b. 与Pad接口的场景2相同，将脏数据位置填充为0；

图 7-12 使用 Pad 接口补齐



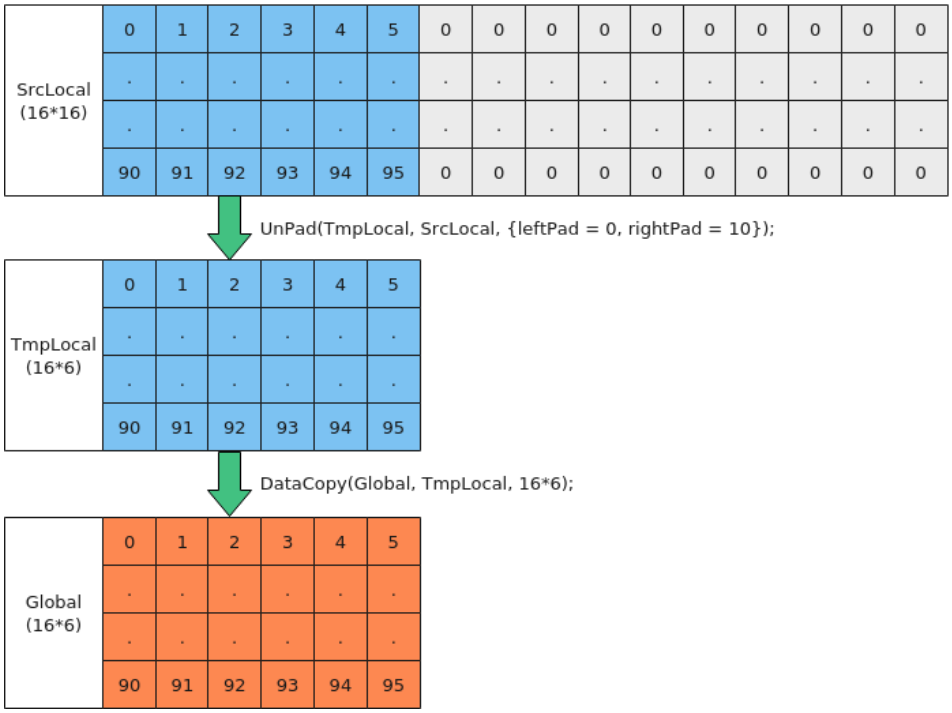
- Local非对齐拷贝出Global，拷贝长度大于32B
  - Local中存在冗余数据，如果有效数据为32B整除，使用UnPad接口去除冗余数据并完整搬出

    如下图所示，Local内存为16*16， 需要将16*6的有效内存搬到Global中，步骤如下：

    i. 要搬出的有效数据16*6为32B对齐，可以使用UnPad高阶API去除冗余值；

    ii. 使用DataCopy搬运出连续的16*6个half数据到Global中。

图 7-13 使用 UnPad 去除冗余值



- 使用GatherMask处理

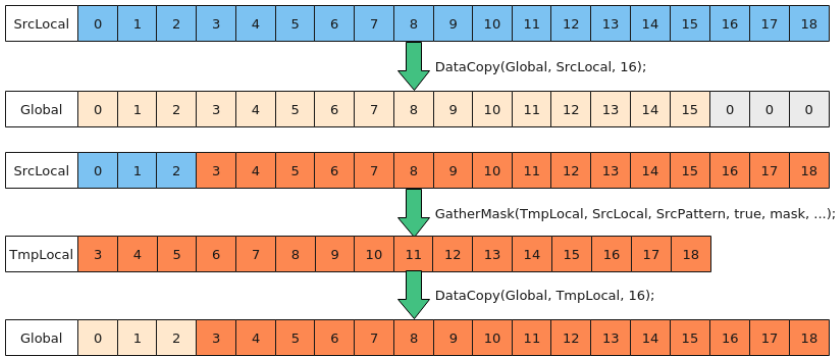
  如下图所示，为搬出19个half数据到Global中，使用GatherMask处理，步骤如下：

  i. 完整拷贝前16个half(32B)数据到Global中；

  ii. 使用GatherMask接口，将SrcLocal[4]~[19]的数Gather到TmpLocal中，TmpLocal从对齐地址开始；

  iii. 从TmpLocal中搬运Gather的数据(32B整数倍)到Global中。

图 7-14 使用 GatherMask 借位搬运



- Local非对齐拷贝出Global， 拷贝长度小于32B

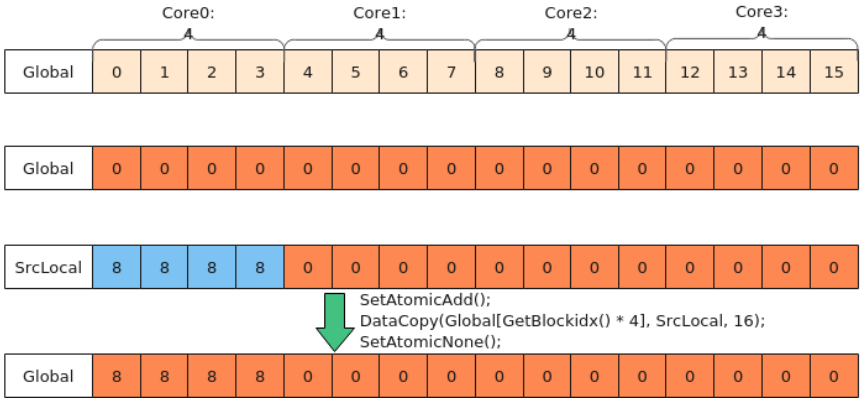
  如下图所示，将一段数据分多核拷出， 每个核拷贝出4个数

  a. 将目标Global完整清零，可以通过在Host清零或者在Kernel侧用UB覆盖的方式处理；

  b. 将本核内的Local数据，除了要搬出的4个有效数，其余冗余部分清零(使用Duplicate)；

c. 使用atomic累加的方式拷贝到Global，因为冗余数据已被清成0值，所以不会出现数据踩踏。

图 7-15 使用 atomic 累加的方式处理拷贝长度小于 32B 的场景



7.2.3 更多场景

动态 shape 场景

在7.2.1 算子实现章节，已经介绍了简单的固定shape矢量算子的kernel侧实现，算子的shape、数据类型都是固定的；在实际的算子开发场景中，这些信息是支持动态变化的，场景会更加灵活和复杂。下文重点进行动态shape与固定shape差异点的介绍。

最主要的区别是：动态Shape场景下，输入的Shape是未知的。一些与输入Shape相关的变量（比如每次搬运的块大小等），需要通过Tiling计算出来，然后传递到kernel侧，kernel侧使用该参数进行后续的计算。

- 7.2.1 算子实现章节中固定shape的算子样例中，TILE_NUM（每个核上总计算数据分块个数）、BLOCK_LENGTH（每个核上总计算数据大小）、TILE_LENGTH（每个分块大小）等是固定的数值。

```
constexpr int32_t TOTAL_LENGTH = 8 * 2048;                            // total length of data
constexpr int32_t USE_CORE_NUM = 8;                                   // num of core used
constexpr int32_t BLOCK_LENGTH = TOTAL_LENGTH / USE_CORE_NUM;         // length computed of each core
constexpr int32_t TILE_NUM = 8;                                       // split data into 8 tiles for each core
constexpr int32_t BUFFER_NUM = 2;                                     // tensor num for each queue
constexpr int32_t TILE_LENGTH = BLOCK_LENGTH / TILE_NUM / BUFFER_NUM; // each tile length is separated to 2 part, due to double buffer
```

- 如果需要将上述代码转换为动态shape，需要在核函数定义中增加Tiling参数，在host侧计算Tiling参数并传入，然后基于Tiling参数计算得到singleCoreSize（每个核上总计算数据大小）、tileNum（每个核上总计算数据分块个数）、tileLength（每个分块大小）等变量。

```
__aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z, uint32_t totalLength, uint32_t tileNum)
{
    ASSERT(GetBlockNum() != 0 && "block dim can not be zero!");
    this->blockLength = totalLength / GetBlockNum();
    this->tileNum = tileNum;
    ASSERT(tileNum != 0 && "tile num can not be zero!");
    this->tileLength = this->blockLength / tileNum / BUFFER_NUM;
    // ...

}

extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR workspace, AddCustomTilingData tiling)
```

```
{
    KernelAdd op;
    op.Init(x, y, z, tiling.totalLength, tiling.tileNum);
    op.Process();
}
```

shape 非对齐场景

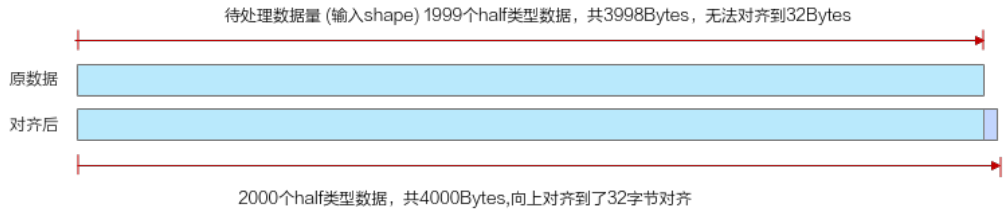
上文描述的都是shape对齐的场景：如下图中的示例，算子的输入shape为（1，2048），支持的数据类型为half类型，可以对齐到一个datablock的大小（32B），也可以平均分配到每个核上（假设使用8个核），每个核上处理256个数，16个datablock。

图 7-16 shape 对齐场景

针对一些非对齐shape，比如算子的输入shape为（1，1999），支持的数据类型为half类型，既无法对齐到一个datablock的大小（32B），也无法平均分配到每个核上，需要一些特殊的Tiling处理方法。

- 因为昇腾AI处理器在进行数据搬运和Vector计算时，对于搬运的数据长度和UB首地址都有必须32B对齐的要求，**首先待处理数据需要先保证对齐到datablock的大小**。该场景下后续搬运和计算的处理细节请参考**7.2.2 非对齐场景**。下图和代码片段展示了将数据对齐到datablock大小的示例：

  图 7-17 对齐到 datablock 大小


```
constexpr uint32_t SIZE_OF_HALF = 2;
constexpr uint32_t BLOCK_SIZE = 32;
```



```
constexpr uint32_t ALIGN_NUM = BLOCK_SIZE / SIZE_OF_HALF;
// shape需要对齐到的datablock,假设原totalLength为1999，向上满足32字节对齐后为2000
uint32_t totalLengthAligned = ((totalLength + ALIGN_NUM - 1) / ALIGN_NUM) * ALIGN_NUM;
```

- 满足datablock对齐后的数据，应尽可能的均分到每个核上。如果无法均分，那么先将可以均分的部分平均分配，剩余的部分分配给部分核，会有部分核多算一个datablock。下图展示了无法均分时将数据进行多核切分的示例。对齐到datablock后为2000个half类型的数据，共125个datablock。125除以8，商为15，余数为5，说明：可以均分的部分平均分配，每个核分配到15个datablock；还剩余5个datablock，分配给5个核，所以会有5个核分配到16个datablock，剩余3个核分配到15个datablock。

图 7-18 无法均分到每个核上的示例

基于上文的描述，可以设计如下的Tiling参数：

- formerNum：分配到大块的核数
- tailNum：分配到小块的核数
- formerLength：大块计算的数据量
- tailLength：小块计算的数据量
- alignNum：一个datablock包含的元素个数

这些Tiling参数的计算方法如下：

```
constexpr uint32_t BLOCK_DIM = 8;
constexpr uint32_t SIZE_OF_HALF = 2;
constexpr uint32_t BLOCK_SIZE = 32;
// shape需要对齐到的最小单位
constexpr uint32_t ALIGN_NUM = BLOCK_SIZE / SIZE_OF_HALF;
...
uint8_t *GenerateTiling()
{
    // shape需要对齐到的datablock,假设原totalLength为1999，向上满足32字节对齐后为2000
    uint32_t totalLengthAligned = ((totalLength + ALIGN_NUM - 1) / ALIGN_NUM) * ALIGN_NUM;
    // 把所有的数据尽可能均匀地分配到每个核上
    // 如果不能均分,先将可以均分的部分平均分配，剩余的部分分配给部分核，会有部分核多算一个
datablock
    // 通过模的计算，可以得到多算一个datablock的核的数量，也可以得到剩余核的数量
    // eg：1999 对齐后的总数据量为2000个数，核心数为8，一个datablock包含16个数，那么：
    // datablock的总数：2000 / 16 = 125
    // 有5个核会分到16个datablock：125 % 8 =5，可以称之为大块
    // 有3个核会分到15个datablock：8 - 5 = 3，可以称之为小块
    uint32_t formerNum = (totalLengthAligned / ALIGN_NUM) % BLOCK_DIM;
```



```
    uint32_t tailNum = BLOCK_DIM - formerNum;
    // 大块计算的数据量：totalLengthAligned / BLOCK_DIM为每个核上计算的元素个数，formerLength为上述元素个数向上32字节对齐的结果
    uint32_t formerLength = ((totalLengthAligned / BLOCK_DIM + ALIGN_NUM - 1) / ALIGN_NUM) * ALIGN_NUM;
    // 小块计算的数据量：totalLengthAligned / BLOCK_DIM为每个核上计算的元素个数，tailLength 为上述元素个数向下32字节对齐的结果
    uint32_t tailLength = (totalLengthAligned / BLOCK_DIM / ALIGN_NUM) * ALIGN_NUM;

...
}
```

相对应的，在Kernel侧，使用获取到的信息计算得到每个核上的偏移量、每个分块大小的样例如下。

```
__aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z, uint32_t formerNum, uint32_t tailNum, uint32_t formerLength, uint32_t tailLength, uint32_t alignNum)
{
    if (GetBlockIdx() < formerNum) {
        this->tileLength = formerLength;
        xGm.SetGlobalBuffer((__gm__ half *)x + formerLength * GetBlockIdx(), formerLength);
        yGm.SetGlobalBuffer((__gm__ half *)y + formerLength * GetBlockIdx(), formerLength);
        zGm.SetGlobalBuffer((__gm__ half *)z + formerLength * GetBlockIdx(), formerLength);
    } else {
        this->tileLength = tailLength;
        xGm.SetGlobalBuffer((__gm__ half *)x + formerLength * formerNum + tailLength * (GetBlockIdx() - formerNum), tailLength);
        yGm.SetGlobalBuffer((__gm__ half *)y + formerLength * formerNum + tailLength * (GetBlockIdx() - formerNum), tailLength);
        zGm.SetGlobalBuffer((__gm__ half *)z + formerLength * formerNum + tailLength * (GetBlockIdx() - formerNum), tailLength);
    }
    pipe.InitBuffer(inQueueX, BUFFER_NUM, this->tileLength * sizeof(half));
    pipe.InitBuffer(inQueueY, BUFFER_NUM, this->tileLength * sizeof(half));
    pipe.InitBuffer(outQueueZ, BUFFER_NUM, this->tileLength * sizeof(half));
}
```

7.3 矩阵编程（高阶 API）

7.3.1 基础知识

说明

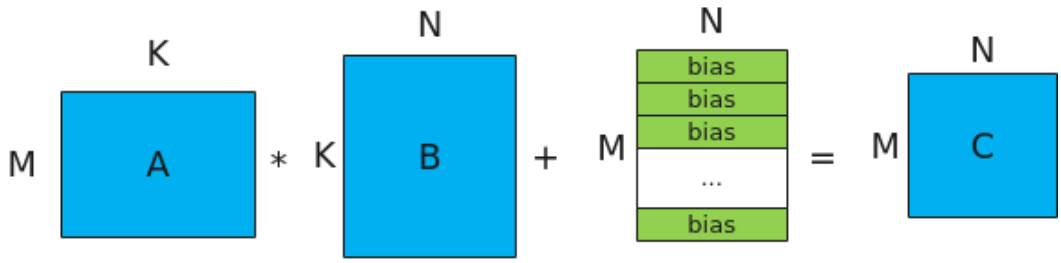
本节内容为使用高阶API进行矩阵乘法的编程指导。使用高阶API进行实际的矩阵编程时，需要通过API参考查看确认接口支持的产品型号。

矩阵乘法概述

MatMul的计算公式为：C = A * B + bias，其示意图如下。

- A、B为源操作数，A为左矩阵，形状为[M, K]；B为右矩阵，形状为[K, N]。
- C为目的操作数，存放矩阵乘结果的矩阵，形状为[M, N]。
- bias为矩阵乘偏置，形状为[1, N]。对A*B结果矩阵的每一行都采用该bias进行偏置。

图 7-19 Matmul 矩阵乘示意图



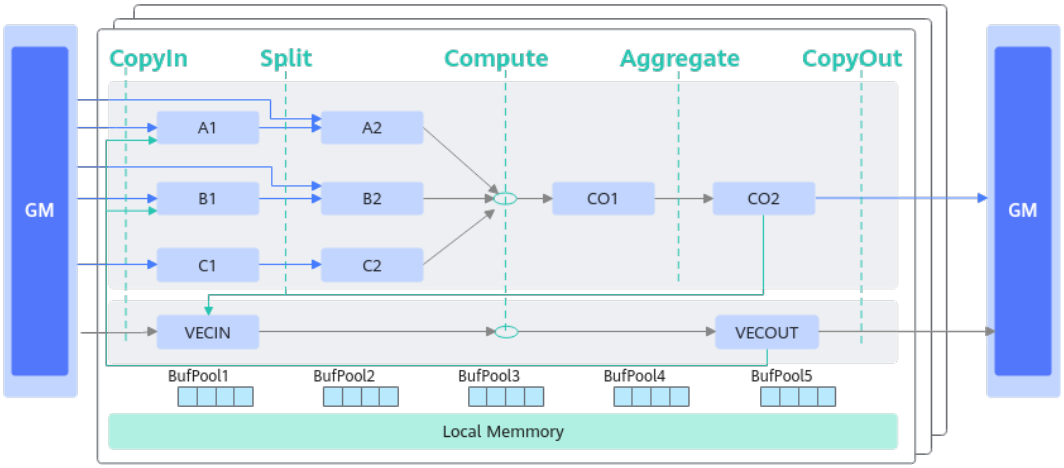
矩阵乘法数据流

在了解矩阵乘法数据流之前，需要先回顾一下几个重要的存储**逻辑位置**的概念：

- 搬入数据的存放位置：A1，用于存放整块A矩阵，可类比CPU多级缓存中的二级缓存；
- 搬入数据的存放位置：B1，用于存放整块B矩阵，可类比CPU多级缓存中的二级缓存；
- 搬入数据的存放位置：A2，用于存放切分后的小块A矩阵，可类比CPU多级缓存中的一级缓存；
- 搬入数据的存放位置：B2，用于存放切分后的小块B矩阵，可类比CPU多级缓存中的一级缓存；
- 结果数据的存放位置：CO1，用于存放小块结果C矩阵，可理解为Cube Out；
- 结果数据的存放位置：CO2，用于存放整块结果C矩阵，可理解为Cube Out；
- 搬入数据的存放位置：VECCALC，一般在计算需要临时变量时使用此位置。

矩阵乘法数据流指矩阵乘的输入输出在各存储位置间的流向。逻辑位置的数据流向如下图所示（为了简化描述，没有列出bias）：

- A矩阵从输入位置到A2的数据流如下（输入位置可以是GM或者VECOUT）：GM->A2，GM->A1->A2；VECOUT->A1->A2。
- B矩阵从输入位置到B2的数据流如下（输入位置可以是GM或者VECOUT）：GM->B2，GM->B1->B2；VECOUT->B1->B2。
- 完成A2*B2=CO1计算。
- CO1数据汇聚到CO2：CO1->CO2。
- 从CO2到输出位置（输出位置可以是GM或者VECIN）：CO2->GM/CO2->VECIN。



数据格式

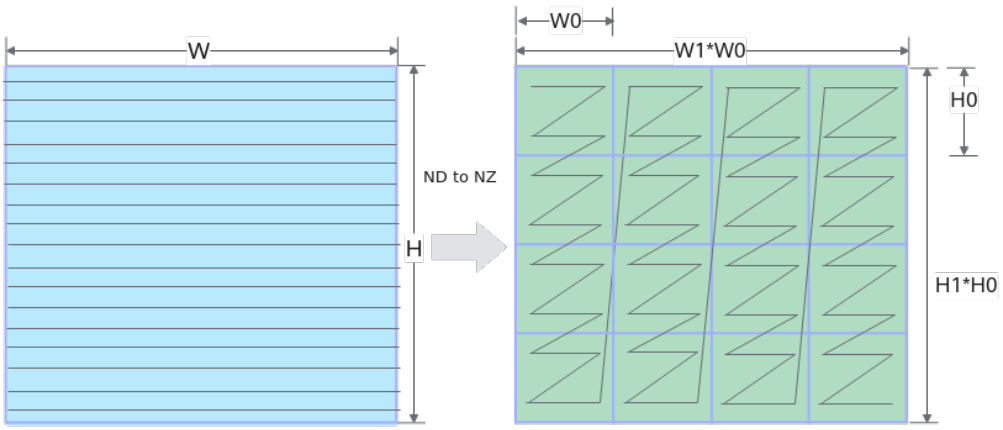
在完成Matmul矩阵乘法时，涉及到两种分形格式ND和NZ。

- ND：普通格式，N维张量。
- NZ：为满足AICore中Cube计算单元高性能计算的需要，引入该特殊格式。

  ND –> NZ的变换过程为：

  ```
  (..., N，H, W )->pad->(..., N, H1*H0, W1*W0)->reshape->(..., N, H1, H0, W1, W0)->transpose->(..., N, W1, H1, H0, W0)
  ```

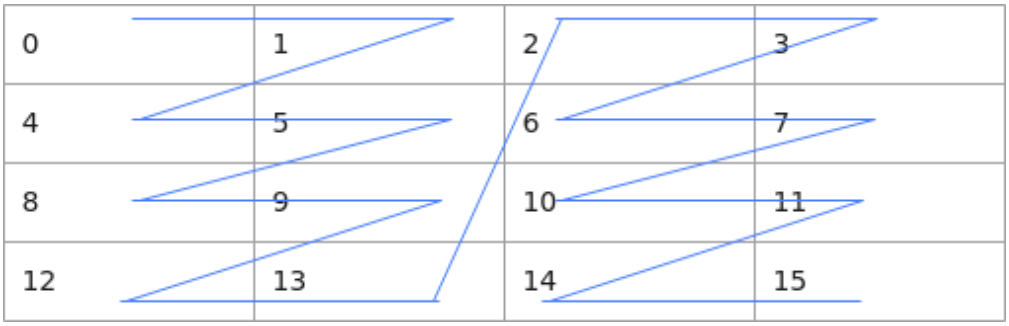
  如下图所示 （ W，H ）大小的矩阵被分为（ H1*W1 ）个分形，按照column major排布，形状如N字形；每个分形内部有（ H0*W0 ）个元素，按照row major排布，形状如z字形。所以这种数据格式称为NZ（ 大N小Z ）格式。



下面我们再通过一个具体的例子来深入理解ND和NZ格式的数据排布区别。假设分形格式为2*2，如下图所示4*4的矩阵，ND和NZ格式存储两种情况下，数据在内存中的排布格式分别为：

ND: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

NZ: 0, 1, 4, 5, 8, 9, 12, 13, 2, 3, 6, 7, 10, 11, 14, 15



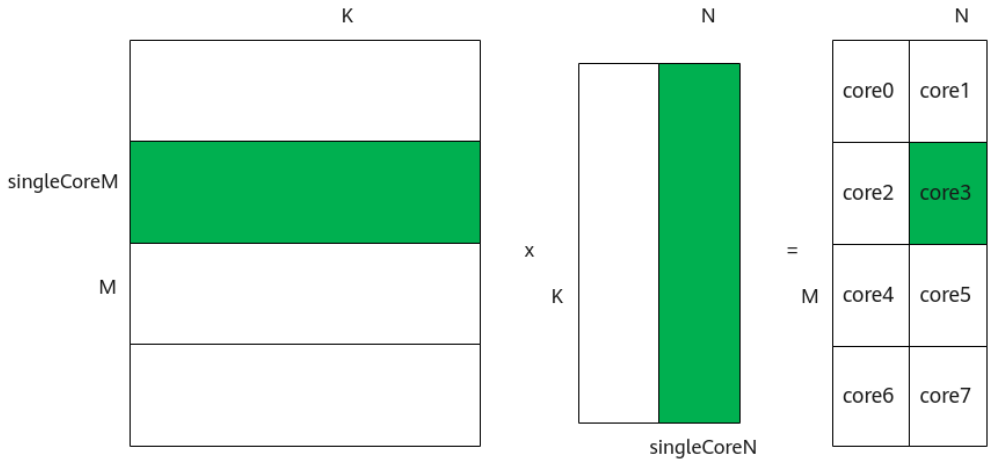
数据分块（Tiling）

- **多核切分**

  为了实现多核并行，需要将矩阵数据进行切分，分配到不同的核上进行处理。切分策略如下图所示：

  - 对于A矩阵，沿着M轴进行切分，切分成多份的singleCoreM，单核上处理SingleCoreM * K大小的数据。
  - 对于B矩阵，沿着N轴进行切分，切分成多份的singleCoreN，单核上处理K * SingleCoreN大小的数据。
  - 对于C矩阵，SingleCoreM * K大小的A矩阵和K * SingleCoreN大小的B矩阵相乘得到SingleCoreM * SingleCoreN大小的C矩阵，即为单核上输出的C矩阵大小。

  比如，下图中共有8个核参与计算，将A矩阵沿着M轴划分为4块，将B矩阵沿着N轴切分为两块，单核上仅处理某一分块（比如图中绿色部分为core3上参与计算的数据）：SingleCoreM * K大小的A矩阵分块和SingleCoreN* K大小的B矩阵分块相乘得到SingleCoreM * SingleCoreN大小的C矩阵分块。

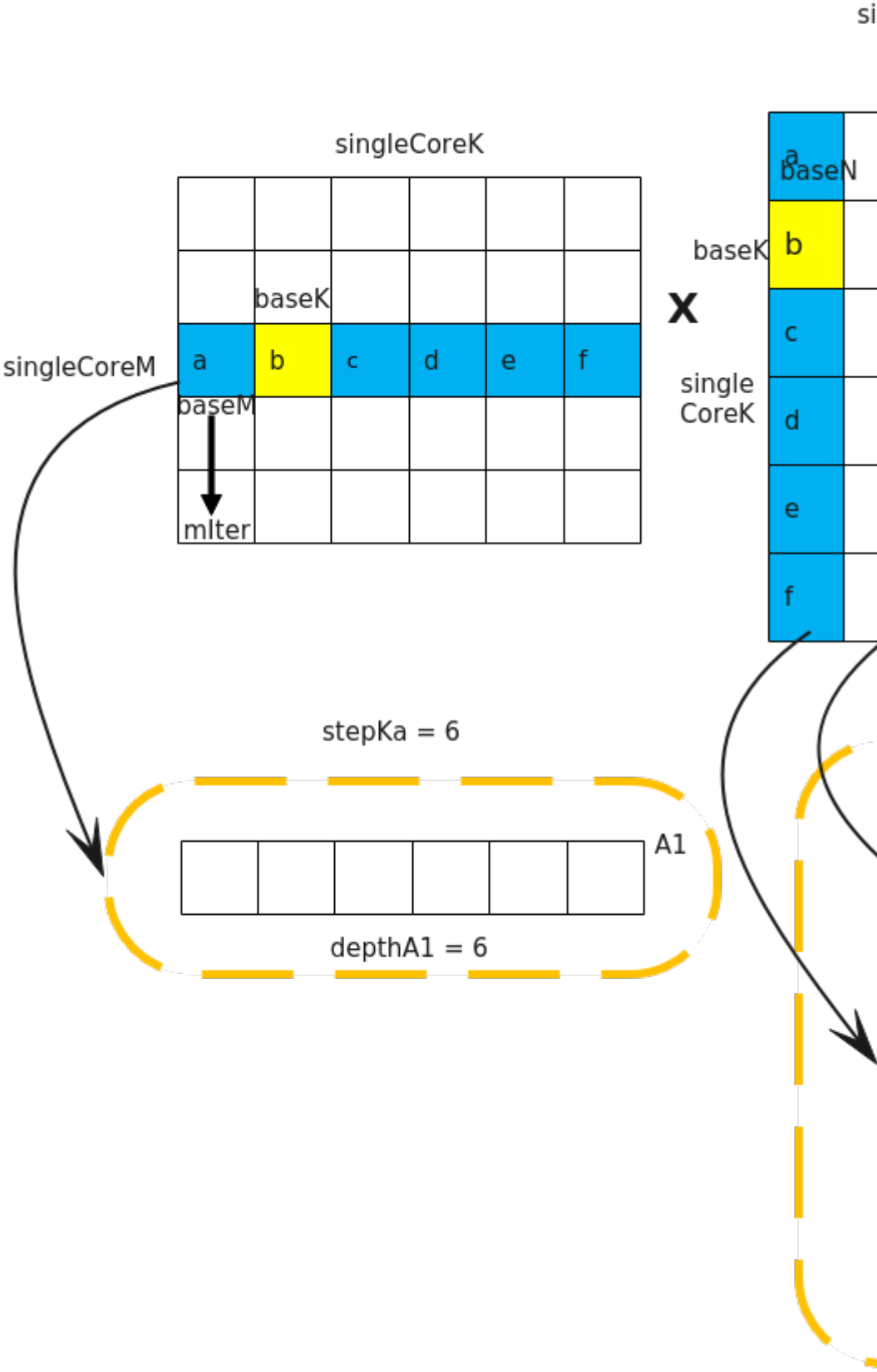


- **核内切分**

  大多数情况下，Local Memory的存储，无法完整的容纳算子的输入与输出，需要每次搬运一部分输入进行计算然后搬出，再搬运下一部分输入进行计算，直到得到完整的最终结果，也就是需要做核内的输入切分。切分的策略如下所示：

  - 对于A矩阵，沿M轴进行切分，切分成多份的baseM；沿K轴进行切分，切分成多份的baseK。
  - 对于B矩阵，沿N轴进行切分，切分成多份的baseN，沿K轴进行切分，切分成多份的baseK。

- 对于C矩阵，A矩阵中baseM*baseK大小的分块和B矩阵中baseK*baseN大小的分块相乘并累加，得到C矩阵中对应位置baseM*baseN大小的分块。比如，图中结果矩阵中的蓝色矩阵块5是通过如下的累加过程得到的：a*a+b*b+c*c+d*d+e*e+f*f。



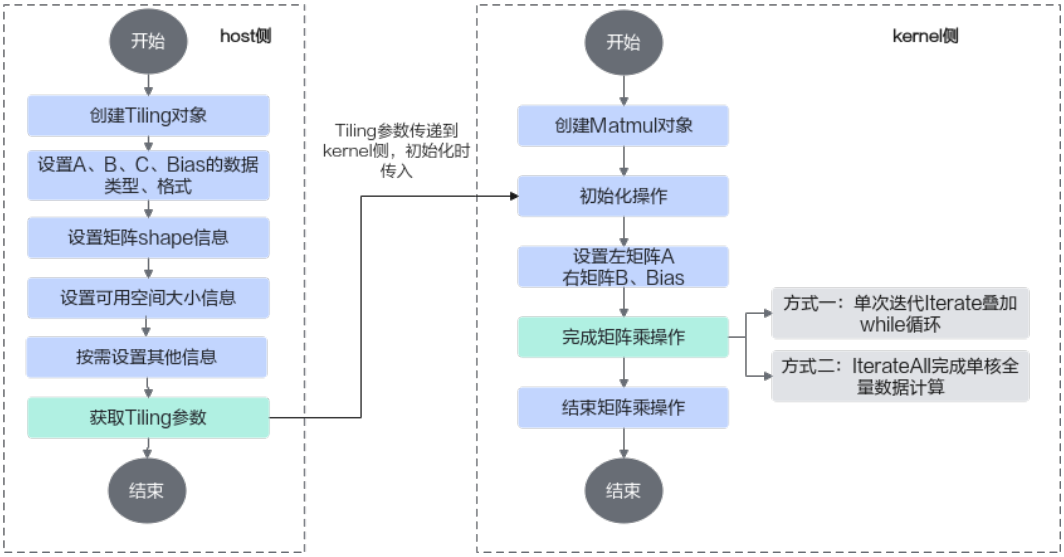
除了baseM, baseN, baseK外，还有一些常用的tiling参数，其含义如下：

- iterateOrder：一次Iterate迭代计算出[baseM, baseN]大小的C矩阵分片。Iterate完成后，Matmul会自动偏移下一次Iterate输出的C矩阵位置，iterOrder表示自动偏移的顺序。
  - 0代表先往M轴方向偏移再往N轴方向偏移
  - 1代表先往N轴方向偏移再往M轴方向偏移
- depthA1，depthB1：A1、B1上存储的矩阵片全载A2/B2的份数，A2、B2存储大小分别是baseM * baseK，baseN * baseK。
- stepM，stepN：stepM为左矩阵在A1中缓存的buffer M方向上baseM的倍数。stepN为右矩阵在B1中缓存的buffer N方向上baseN的倍数。
- stepKa，stepKb：stepKa为左矩阵在A1中缓存的buffer K方向上baseK的倍数，stepKb为右矩阵在B1中缓存的buffer K方向上baseK的倍数。

7.3.2 算子实现

实现流程

上文介绍了Matmul矩阵乘的数据切分方案和数据流。Ascend C提供一组Matmul高阶API，封装了这些常用的切分和数据搬运、计算的算法逻辑，方便用户快速实现Matmul矩阵乘法的运算操作。开发者在host侧通过调用API自动获取Tiling参数，该参数传递到kernel侧后，在初始化操作时传入，通过几个简单的API即可完成矩阵乘操作。完整样例请参考**LINK**。



host侧自动获取Tiling参数的关键步骤介绍如下：

步骤1 创建Tiling对象。

```
auto ascendcPlatform = platform_ascendc::PlatformAscendC(context->GetPlatformInfo());
matmul_tiling::MatmulApiTiling cubeTiling(ascendcPlatform);
```

创建对象时需要传入硬件平台信息，硬件平台信息可以通过GetPlatformInfo获取。

步骤2 设置A、B、Bias的数据类型和格式。

```
cubeTiling.SetAType(AscendC::TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT16);
cubeTiling.SetBType(AscendC::TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT16);
cubeTiling.SetCType(AscendC::TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT);
cubeTiling.SetBiasType(AscendC::TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT);
```


步骤3 设置矩阵shape信息。

```
cubeTiling.SetShape(M, N, K);
cubeTiling.SetOrgShape(M, N, K);
```

步骤4 设置可用空间大小信息。

```
cubeTiling.SetBufferSpace(-1, -1, -1);
```

步骤5 按需设置其他参数，比如设置bias参与计算。

```
cubeTiling.SetBias(true);
```

步骤6 获取Tiling参数。

```
MatmulCustomTilingData tiling;
if (cubeTiling.GetTiling(tiling.cubeTilingData) == -1){
    return ge::GRAPH_FAILED;
}
```

步骤7 Tiling参数的序列化保存等其他操作。

----结束

kernel侧使用Matmul API矩阵乘运算的具体步骤如下：

步骤1 创建Matmul对象

创建Matmul对象的示例如下：

- CUBE_ONLY（只有矩阵计算）场景下，需要设置ASCENDC_CUBE_ONLY代码宏，本节内容以CUBE_ONLY模式举例。
- 默认为MIX模式（包含矩阵计算和矢量计算），该场景下，不能设置ASCENDC_CUBE_ONLY代码宏，更多内容请参考**7.5 融合算子编程**。

```
// 纯cube模式（只有矩阵计算）场景下，需要设置该代码宏，并且必须在#include "lib/matmul_intf.h"之前设置
#define ASCENDC_CUBE_ONLY
#include "lib/matmul_intf.h"
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, half> aType;
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, half> bType;
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, float> cType;
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, float> biasType;
matmul::Matmul<aType, bType, cType, biasType> mm;
```

创建对象时需要传入A、B、C、Bias的参数类型信息， 类型信息通过MatmulType来定义，包括：内存逻辑位置、数据格式、数据类型。

步骤2 初始化操作。

```
REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm, &tiling); // 初始化
```

说明

Matmul高阶API内部实现时需要使用系统workspace，开发者需要：

- 在host侧Tiling实现时，设置总的workspace的数值大小（包含用户workspace和系统workspace），workspace空间由框架来申请并管理。系统workspace的空间大小通过GetLibApiWorkSpaceSize获取。

  ```
  size_t userWorkspaceSize = 0;
  size_t systemWorkspaceSize = static_cast<size_t>(ascendcPlatform.GetLibApiWorkSpaceSize());
  size_t *currentWorkspace = context->GetWorkspaceSizes(1);
  currentWorkspace[0] = userWorkspaceSize + systemWorkspaceSize;
  ```

- 若算子工程不是**自定义算子工程**，也不是带有**-DHAVE_WORKSPACE**编译宏的Kernel直调算子工程，kernel侧需要在Matmul初始化前，通过SetSysWorkSpace设置系统workspace。

  ```
  // 使用Matmul时必须设置workspace空间
  SetSysWorkspace(workspace);
  if (GetSysWorkSpacePtr() == nullptr) {
      return;
  }
  ```


步骤3 设置左矩阵A、右矩阵B、Bias。

```
mm.SetTensorA(gm_a);   // 设置左矩阵A
mm.SetTensorB(gm_b);   // 设置右矩阵B
mm.SetBias(gm_bias);   // 设置Bias
```

步骤4 完成矩阵乘操作。

- 调用Iterate完成单次迭代计算，叠加while循环完成单核全量数据的计算。Iterate方式，可以自行控制迭代次数，完成所需数据量的计算，方式比较灵活。

```
while (mm.Iterate()) {
    mm.GetTensorC(gm_c);
}
```

- 调用IterateAll完成单核上所有数据的计算。IterateAll方式，无需循环迭代，使用比较简单。

```
mm.IterateAll(gm_c);
```

步骤5 结束矩阵乘操作。

```
mm.End();
```

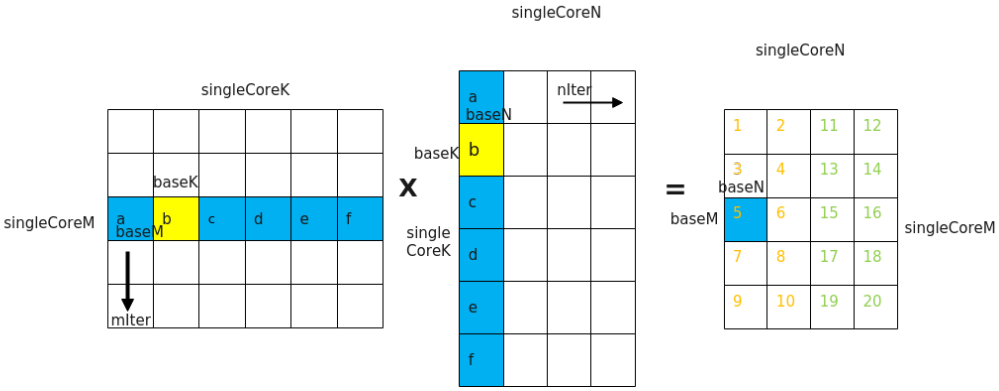
----结束

设置 Shape 信息

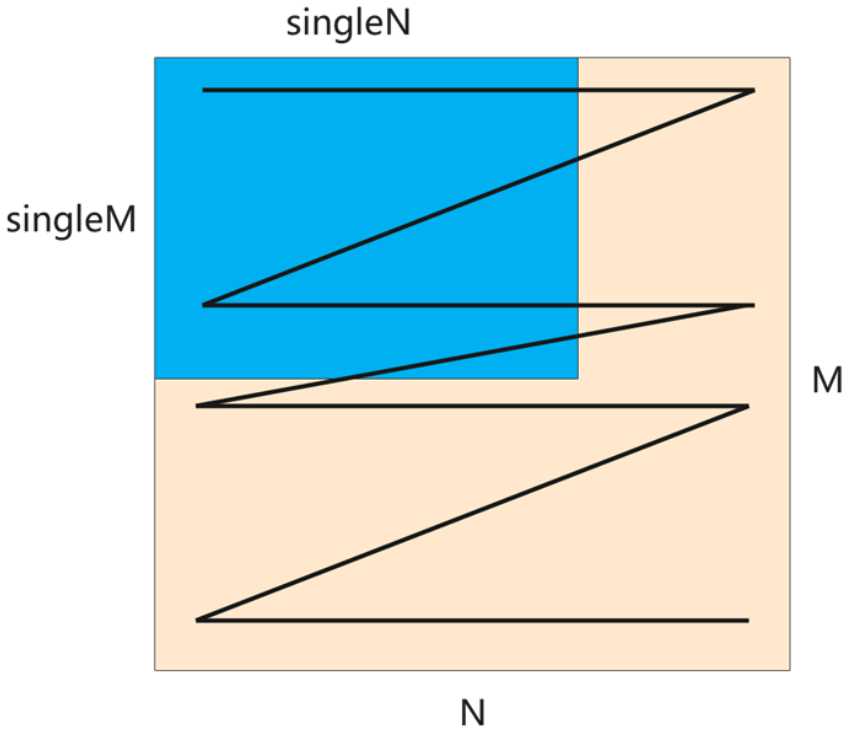
Host Tiling时可以设置Shape信息，用于Tiling计算；kernel侧运行时也可以修改部分shape信息，用于尾块设置、Matmul复用（多个Matmul计算复用一个Matmul对象）等场景。本节对涉及到的Shape概念进行介绍，并给出host侧和kernel侧设置Tiling信息的指导。

- orgShape：M、N、K
- singleCoreShape：singleCoreM、singleCoreN、singleCoreK
- singleShape：singleM、singleN、singleK
- baseShape：baseM、baseN、baseK

通过**数据分块（Tiling）**的介绍我们已经了解了orgShape(M、N、K)，singleCoreShape(singleCoreM、singleCoreN、singleCoreK)，baseShape(baseM、baseN、baseK)的概念，如下图所示：



除此之外，单核的Matmul Tiling时，实际参与Matmul计算的shape可以是原始shape中的一部分，singleM, singleN, singleK用于表达实际参与Matmul计算的shape，如下图所示。在单核的情况下，singleM, singleN, singleK会透传给singleCoreM, singleCoreN, singleCoreK。



- Kernel运行时设置
  - SetTail、SetSingleShape都是运行时修改singleCoreM、singleCoreN、singleCoreK，处理尾块时使用SetTail，Matmul复用（多个Matmul计算复用一个Matmul对象）的场景可以使用SetSingleShape重新设置。
  - SetOrgShape是运行时修改M、N、K，Matmul复用的场景可以使用SetOrgShape重新设置。
- 单核Tiling时设置
  - SetOrgShape（必选）：设置M、N、K
  - SetShape（非必选）： 设置singleM, singleN, singleK，等同于是设置singleCoreM、singleCoreN、singleCoreK
  - SetFixSplit（非必选）：设置baseM、baseN、baseK
- 多核Tiling时设置
  - SetOrgShape（必选）：设置M、N、K
  - SetShape（非必选）： 设置singleM, singleN, singleK
  - SetFixSplit（非必选）：设置baseM、baseN、baseK
  - SetSingleShape(非必选)： 设置singleCoreM、singleCoreN、singleCoreK
  - SetSingleRange(非必选) ：设置singleCoreM、singleCoreN、singleCoreK的范围

设置 format 格式

创建Matmul对象时需要传入A、B、C、Bias的参数类型信息， 类型信息通过MatmulType来定义，包括：内存逻辑位置、数据格式、数据类型。示例如下：

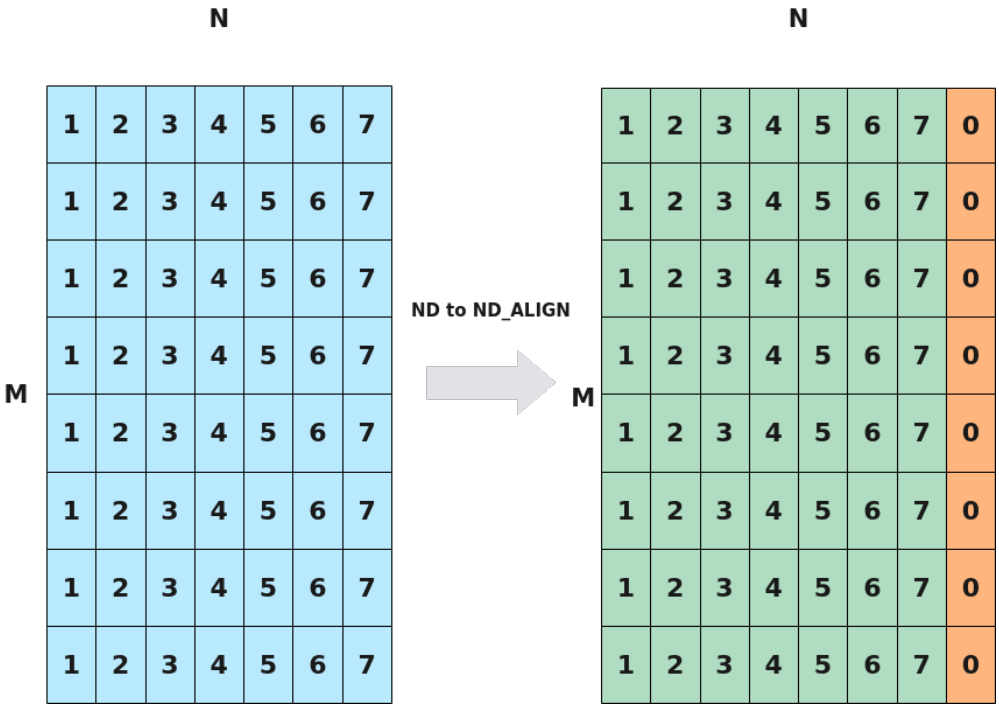
```
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, half> aType;
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, half> bType;
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, float> cType;
```



```
typedef matmul::MatmulType<AscendC::TPosition::GM, CubeFormat::ND, float> biasType;
matmul::Matmul<aType, bType, cType, biasType> mm;
```

针对数据格式，包括CubeFormat::ND, CubeFormat::NZ, CubeFormat::ND_ALIGN三种，ND和NZ格式在**数据格式**章节已经介绍。

ND_ALIGN用于配置输出矩阵时按照一定的补齐规则进行输出。ND->ND_ALIGN变换过程下图所示，矩阵数据类型为uint32_t，假设输出矩阵输出到UB，原矩阵N方向没有32字节对齐，设置ND_ALIGN则在其后补0，将其对齐到32字节。



7.3.3 多核场景

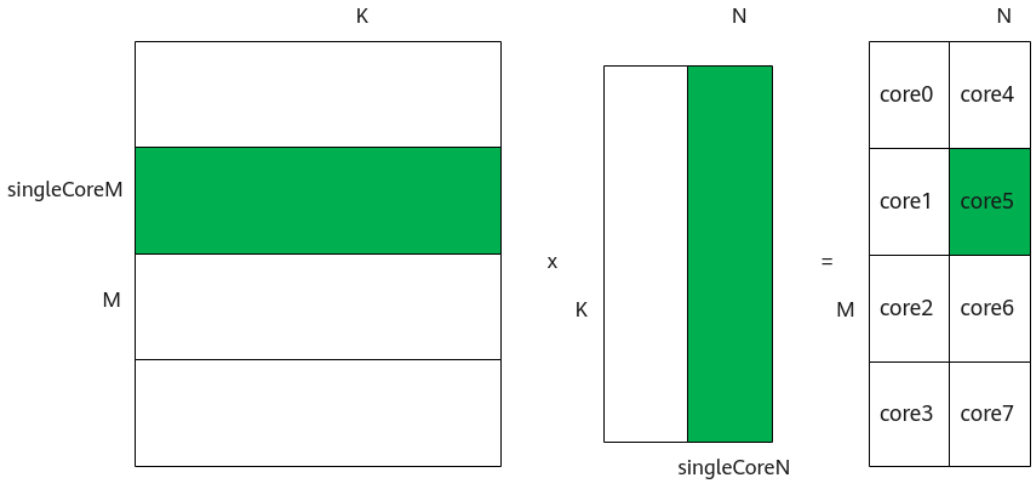
多核切分

为了实现多核并行，提升计算效率，需要将矩阵数据进行切分，分配到不同的核上进行处理。主要的切分策略有切分K轴和不切分K轴两种。

不切分K轴、仅切分M、N轴的策略如下：

- 对于A矩阵，沿着M轴进行切分，切分成多份的singleCoreM，单核上处理SingleCoreM * K大小的数据。
- 对于B矩阵，沿着N轴进行切分，切分成多份的singleCoreN，单核上处理K * SingleCoreN大小的数据。
- 对于C矩阵，SingleCoreM * K大小的A矩阵和K * SingleCoreN大小的B矩阵相乘得到SingleCoreM * SingleCoreN大小的C矩阵，即为单核上输出的C矩阵大小。

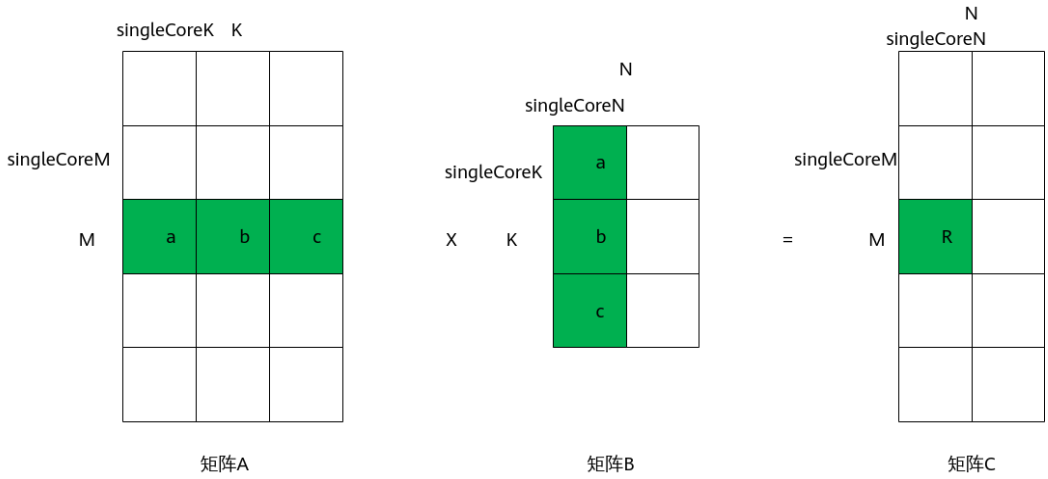
比如，下图中共有8个核参与计算，将A矩阵沿着M轴划分为4块，将B矩阵沿着N轴切分为两块，单核上仅处理某一分块（比如图中绿色部分为core5上参与计算的数据）：SingleCoreM * K大小的A矩阵分块和SingleCoreN * K大小的B矩阵分块相乘得到SingleCoreM * SingleCoreN大小的C矩阵分块。



切分M、N、K轴的策略如下图所示：

- 对于A矩阵，沿着M轴进行切分，切分成多份的singleCoreM，沿着K轴切分，切分成多份的singleCoreK，单核上处理singleCoreM * singleCoreK大小的数据。
- 对于B矩阵，沿着K轴进行切分，切分成多份的singleCoreK，沿着N轴进行切分，切分成多份的singleCoreN，单核上处理singleCoreK * singleCoreN大小的数据。
- 对于C矩阵，singleCoreM * singleCoreK大小的A矩阵与singleCoreK * singleCoreN大小的B矩阵相乘并累加得到singleCoreM * singleCoreN大小的C矩阵分块。

比如下图中，C矩阵中的R矩阵块，是通过a*a+b*b+c*c累加得到的，其中，a*a、b*b、c*c可在多个核上并行计算。



上述的切分策略会在Tiling参数中体现，比如SingleCoreM、SingleCoreN、SingleCoreK，开发者在host侧通过调用API自动获取Tiling参数，与单核场景的不同的是，多核Tiling需要使用MultiCoreMatmulTiling构造多核Tiling对象，并通过SetDim接口设置Matmul计算所用的核数。注意：这里设置的核数为Matmul计算所用的核数，仅在多核场景下设置，用于计算tiling参数；SetBlockDim为整个算子计算所用核数，是实际会加载的核数，是必须设置的。SetBlockDim的设置规则请参考**9.3.3 Host侧tiling实现**。SetDim的设置规则如下：

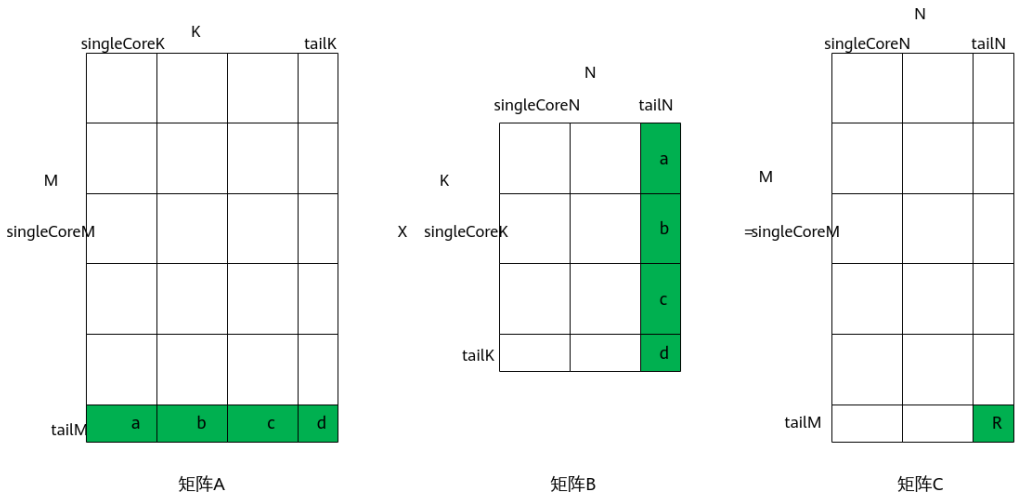
- CUBE_ONLY（只有矩阵计算）场景，本节内容以CUBE_ONLY模式举例。
  SetBlockDim加载的核数就是Matmul API实际计算会用到的核数，SetDim和SetBlockDim设置的值是一样的。
- MIX模式（包含矩阵计算和矢量计算）的设置规则请参考**MIX场景核数设置规则**。

Matmul多核场景的完整样例请参考**matmul多核单算子调用样例**、**matmul多核Kernel直调样例**。

```
// 构造多核Tiling对象
auto ascendcPlatform = platform_ascendc::PlatformAscendC(context->GetPlatformInfo());
matmul_tiling::MultiCoreMatmulTiling tiling(ascendcPlatform);
// 设置可参与矩阵乘运算的核数
tiling.SetDim(5);
tiling.SetAType(AscendC::TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT16);
...
optiling::TCubeTiling tilingData;
// 获取Tiling参数
int ret = tiling.GetTiling(tilingData);    // if ret = -1, gen tiling failed
```

非对齐场景尾块处理

多核场景，对矩阵进行切分时，若M、N、K无法整除singleCoreM 、singleCoreN、singleCoreK时，就会出现尾块，如下图矩阵A、B的最后一行和最后一列的矩阵块：



此时，C矩阵中的R矩阵块，依然是通过a*a+b*b+c*c+d*d累加得到的，处理a*a、b*b、c*c、d*d等尾块时，需在kernel侧设置尾块大小，在不改变原有tiling的情况下，重新设置本次计算的singleCoreM/singleCoreN/singleCoreK，在处理尾块的时候按照设置的值也就是tailM/tailN/tailK进行搬运和计算。如下的示例中，当tailM < singleCoreM被认为需要处理尾块，此时可以调用SetTail接口进行设置。SetTail接口需要在Iterate/IterateAll之前调用。

```
template<typename aType, typename bType, typename cType, typename biasType
__aicore__ inline void MatmulKernel<aType, bType, cType, biasType>::CalcOffset(int32_t blockIdx, const
TCubeTiling& tiling, int32_t& offsetA, int32_t& offsetB, int32_t& offsetC, int32_t& offsetBias, bool isAtrans,
bool isBtrans){
    auto mSingleBlocks = Ceil(tiling.M, tiling.singleCoreM);
    auto mCoreIndx = blockIdx % mSingleBlocks;
    auto nCoreIndx = blockIdx / mSingleBlocks;

    ...
    // 处理尾块
    int tailM = tiling.M - mCoreIndx * tiling.singleCoreM;
    tailM = tailM < tiling.singleCoreM ? tailM : tiling.singleCoreM;
```



```
    int tailN = tiling.N - nCoreIndx * tiling.singleCoreN;
    tailN = tailN < tiling.singleCoreN ? tailN : tiling.singleCoreN;
    if (tailM < tiling.singleCoreM || tailN < tiling.singleCoreN) {
        matmulObj.SetTail(tailM, tailN);
    }
}
```

7.3.4 异步场景

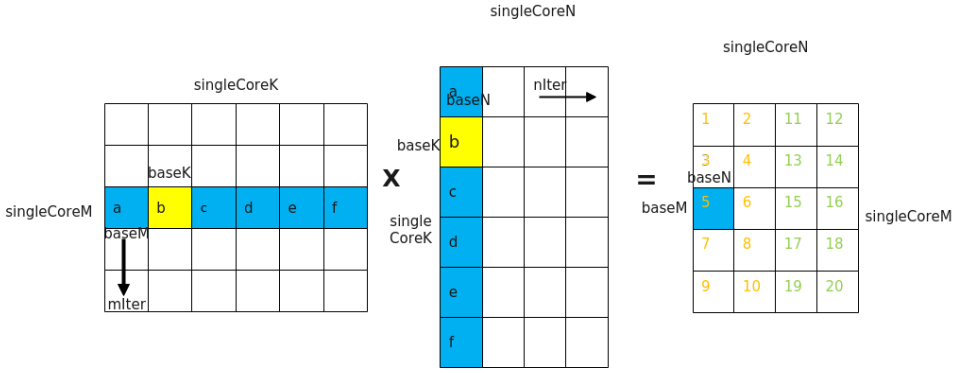
Matmul的Iterate和IterateAll接口提供了同步和异步两种模式。

同步模式指的是程序执行时，需要等待某个操作完成后才能继续执行下一步操作。 异步模式指的是程序执行时，不需要等待某个操作完成就可以继续执行下一步操作。

Iterate&GetTensorC 的同步和异步

- 同步：执行完一次Iterate迭代计算后，执行GetTensorC搬运矩阵C分片，搬运完成后，才能进行下一次计算。

  如下图所示，C矩阵中，矩阵块1搬走后，才能计算矩阵块2，阵块2搬运完成后，才能计算矩阵块3。



  同步模式的样例代码如下

```
while (mm. Iterate ()) {
    mm. GetTensorC (gm_c);
}
```

- 异步：通过设置模板参数开启异步模式。调用Iterate后，无需立即调用GetTensorC同步等待，可以先执行其他操作，待需要获取结果时再调用GetTensorC。异步方式可以减少同步等待，提高并行度，开发者对计算性能要求较高时，可以选用该方式。异步场景时，需要使用一块临时空间来缓存Iterate计算结果，否则会覆盖计算结果，调用GetTensorC时会在该临时空间中获取C的矩阵分片。临时空间通过SetWorkspace接口进行设置。SetWorkspace接口需要在Iterate接口之前调用。Iterate&GetTensorC异步场景的完整样例请参考**异步场景样例**。

```
mm.SetWorkspace(workspace, size); // 其中，workspace 为临时空间的物理地址，size为singleCoreM * singleCoreN大小的矩阵C占用的内存大小: singleCoreM * singleCoreN * sizeof(cDataType)

// 异步模式
mm.template Iterate<false>();
…… // 执行其他操作
for (int i = 0; i < singleCoreM/baseM*singleCoreN/baseN; ++i) {
    mm.GetTensorC<false> (gm_c);
}
```

IterateAll 的同步和异步

- 同步：后续操作需要同步等待IterateAll执行结束


```
mm.SetTensorA(gm_a);    // 设置左矩阵A
mm.SetTensorB(gm_b);    // 设置右矩阵B
mm.SetBias(gm_bias);    // 设置Bias
mm.IterateAll(gm_c);
// 后续操作
...
```

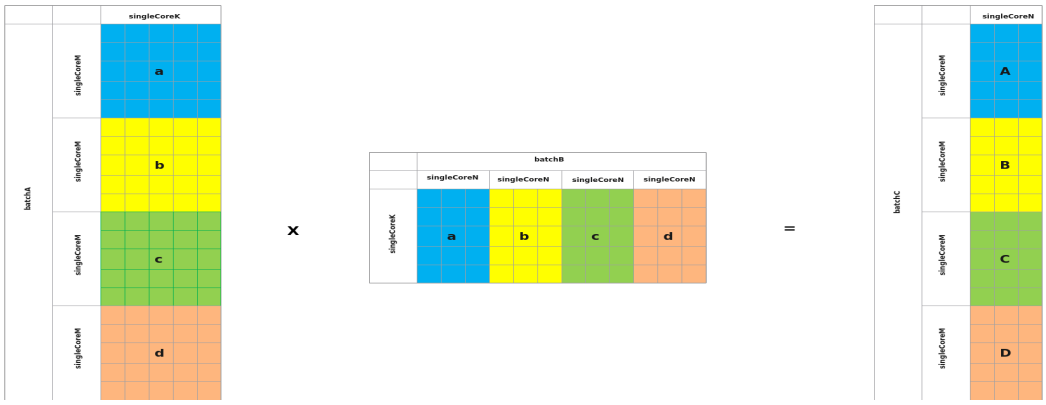
- 异步：后续操作不需要同步等待IterateAll执行结束，需要IterateAll的结果时，调用WaitIterateAll等待IterateAll异步接口返回。

```
matmul::Matmul<aType, bType, cType, biasType> mm;
mm.SetTensorA(queryGm[tensorACoreOffset]);
mm.SetTensorB(keyGm[tensorBCoreOffset + sInnerStart * singleProcessSInnerSize *
    tilingData->attentionScoreOffestStrideParams.matmulHead], true);
mm.SetTail(singleProcessSOuterSize, mmNNum);
mm.template IterateAll<false>(workspaceGm[tmp_block_idx * mmResUbSize *
sInnerLoopTimes],false,true);
// do some others compute
mm.WaitIterateAll(); // 等待IterateAll完成
DataCopy(dstUB, GM);  // 进行GM到UB的拷贝
```

7.3.5 batch 场景

单次Matmul计算处理的shape比较小时，由于每次计算均涉及到内部的通信，可能会影响性能，该接口提供批量处理Matmul的功能，调用一次IterateBatch，可以计算出多个singleCoreM * singleCoreN大小的C矩阵。

如下的示例中，包含4个矩阵乘操作a*a、b*b、c*c、d*d，需要单核上计算多个singleCoreM *singleCoreN，shape较小的情况可以使能BatchMatmul，批量处理。以BMK*BKN=BMN（相关格式参见IterateBatch）场景为例，如下图，一次IterateBatch可同时计算出A = a*a、B = b*b、C = c*c、D = d*d。



实例化Matmul时，需要通过MatmulType设置输入输出的Layout格式为NORMAL(BMNK的数据排布格式使用NORMAL表示)。Host侧Tiling时需使用SetBatchInfoForNormal设置A/B/C的M/N/K轴信息和A/B矩阵的BatchNum数。

如下示例完成aGM、bGM矩阵乘，结果保存到cGm上，其中aGM、bGM、cGM数据的layout格式均为NORMAL，左矩阵每次计算batchA个MK数据，右矩阵每次计算batchB个KN数据。更多数据排布格式的详细示例请参考**BatchMatmul样例**。

```
#include "kernel_operator.h"
#include "lib/matmul_intf.h"

extern "C" __global__  __aicore__ void kernel_matmul_rpc_batch(GM_ADDR aGM, GM_ADDR bGM,
GM_ADDR cGM, GM_ADDR biasGM, GM_ADDR tilingGM, GM_ADDR workspaceGM, uint32_t
isTransposeAIn, uint32_t isTransposeBIn, int32_t batchA,  int32_t batchB)
{
    // 定义matmul type
    typedef matmul::MatmulType <AscendC::TPosition::GM, CubeFormat::ND, half, false,
LayoutMode::NORMAL> aType;
```

```
    typedef matmul::MatmulType <AscendC::TPosition::GM, CubeFormat::ND, half, true,
LayoutMode::NORMAL> bType;
    typedef matmul::MatmulType <AscendC::TPosition::GM, CubeFormat::ND, float, false,
LayoutMode::NORMAL> cType;
    typedef matmul::MatmulType <AscendC::TPosition::GM, CubeFormat::ND, float> biasType;

    // 初始化tiling数据
    TCubeTiling tiling;
    auto tempTilingGM = (__gm__ uint32_t*)tilingGM;
    auto tempTiling = (uint32_t*)&tiling;
    for (int i = 0; i < sizeof(TCubeTiling) / sizeof(int32_t); ++i, ++tempTilingGM, ++tempTiling) {
        *tempTiling = *tempTilingGM;
    }

    // 初始化gm数据
    AscendC::GlobalTensor<half> aGlobal;
    AscendC::GlobalTensor<half> bGlobal;
    AscendC::GlobalTensor<float> cGlobal;
    AscendC::GlobalTensor<float> biasGlobal;
    int32_t sizeA = tiling.ALayoutInfoB * tiling.singleCoreM * tiling.singleCoreK * sizeof(A_T);
    int32_t sizeB = tiling.BLayoutInfoB * tiling.singleCoreK * tiling.singleCoreN * sizeof(B_T);
    int32_t sizeC = tiling.CLayoutInfoB * tiling.singleCoreM * tiling.singleCoreN * sizeof(C_T);
    int32_t sizebias = tiling.CLayoutInfoB * tiling.singleCoreN * sizeof(C_T);
    aGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ half*>(aGM), sizeA);
    bGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ half*>(bGM), sizeB);
    cGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ float*>(cGM), sizeC);
    biasGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ float*>(biasGM), sizebias);
    tiling.shareMode = 0;
    tiling.shareL1Size = 512 * 1024;
    tiling.shareL0CSize = 128 * 1024;
    tiling.shareUbSize = 0;
    int offset_a=0, offset_b=0, offset_c=0, offset_bias=0;
    AscendC::GlobalTensor<A_T> gm_a;
    gm_a.SetGlobalBuffer(const_cast<__gm__ A_T*>(aGlobal[offset_a].GetPhyAddr()), tiling.singleCoreM *
tiling.singleCoreK);
    AscendC::GlobalTensor<B_T> gm_b;
    gm_b.SetGlobalBuffer(const_cast<__gm__ B_T*>(bGlobal[offset_b].GetPhyAddr()), tiling.singleCoreK *
tiling.singleCoreN);
    AscendC::GlobalTensor<C_T> gm_c;
    gm_c.SetGlobalBuffer(const_cast<__gm__ C_T*>(cGlobal[offset_c].GetPhyAddr()), tiling.singleCoreM *
tiling.singleCoreN) ;
    AscendC::GlobalTensor<BiasT> gm_bias;
    gm_bias.SetGlobalBuffer(const_cast<__gm__ BiasT*>(biasGlobal[offset_bias].GetPhyAddr()),
tiling.singleCoreN);
    // 创建Matmul实例
    constexpr MatmulConfig MM_CFG = GetNormalConfig(false, false, false,
BatchMode::BATCH_LESS_THAN_L1);
    matmul::Matmul<aType, bType, cType, biasType, MM_CFG> mm1;
    AscendC::TPipe pipe;
    g_cubeTPipePtr = &pipe;
    SetSysWorkspace(workspaceGM);
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm1);
    mm1.Init(&tiling);
    mm1.SetTensorA(gm_a, isTransposeAIn);
    mm1.SetTensorB(gm_b, isTransposeBIn);
    if(tiling.isBias) {
        mm1.SetBias(gm_bias);
    }
    // 多batch Matmul计算
    mm1.IterateBatch(gm_c, batchA, batchB, false);
}
```

7.4 矩阵编程（基础 API）

7.4.1 简介

说明

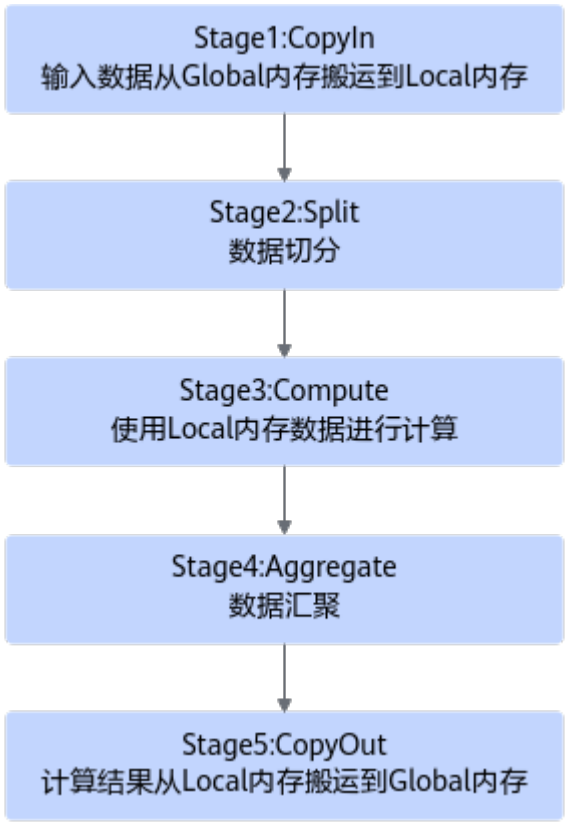
本节内容为使用基础API进行矩阵乘法的编程指导。使用基础API进行矩阵编程的功能支持的产品型号为：

- Atlas 推理系列产品
- Atlas 训练系列产品

编程范式

Cube编程范式把算子的实现流程分为5个基本任务：CopyIn，Split，Compute，Aggregate，CopyOut。CopyIn负责搬入操作，Split负责数据切分操作，Compute负责矩阵指令计算操作，Aggregate负责数据汇聚操作，CopyOut负责搬出操作。

图 7-20 矩阵编程基本任务设计



具体任务之间的交互流程和流程图如下。

步骤1 Stage1：CopyIn任务。

1. 使用DataCopy接口将GlobalTensor数据拷贝到LocalTensor。
2. 使用EnQue将LocalTensor放入A1/B1的Queue中。

步骤2 Stage2：Split任务。

1. 使用DeQue从A1/B1中取出LocalTensor。
2. 使用Ascend C接口将LocalTensor从A1/B1中搬运到A2/B2。

3. 使用EnQue将计算结果LocalTensor放入到A2/B2的Queue中。

步骤3 Stage3：Compute任务。

1. 使用DeQue从A2/B2中取出LocalTensor。
2. 使用Ascend C接口完成矩阵计算。
3. 使用EnQue将计算结果LocalTensor放入到CO1的Queue中。

步骤4 Stage4：Aggregate任务。

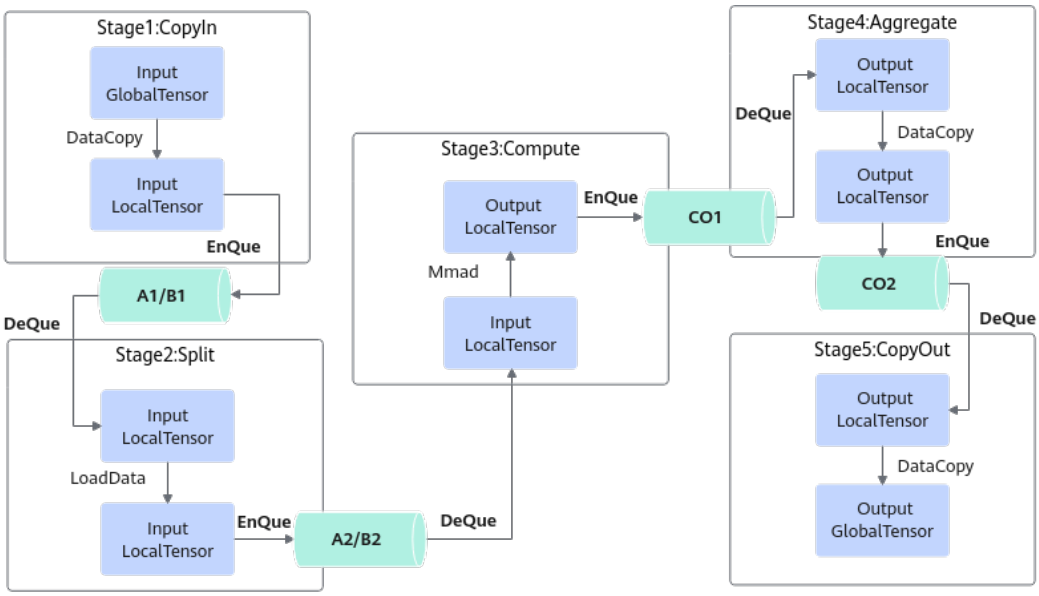
1. 使用DeQue从CO1中取出LocalTensor。
2. 使用Ascend C接口拷贝结果矩阵到CO2。
3. 使用EnQue将计算结果LocalTensor放入到CO2的Queue中。

步骤5 Stage5：CopyOut任务。

1. 使用DeQue接口从CO2的Queue中取出LocalTensor。
2. 使用DataCopy接口将LocalTensor拷贝到GlobalTensor上。

----结束

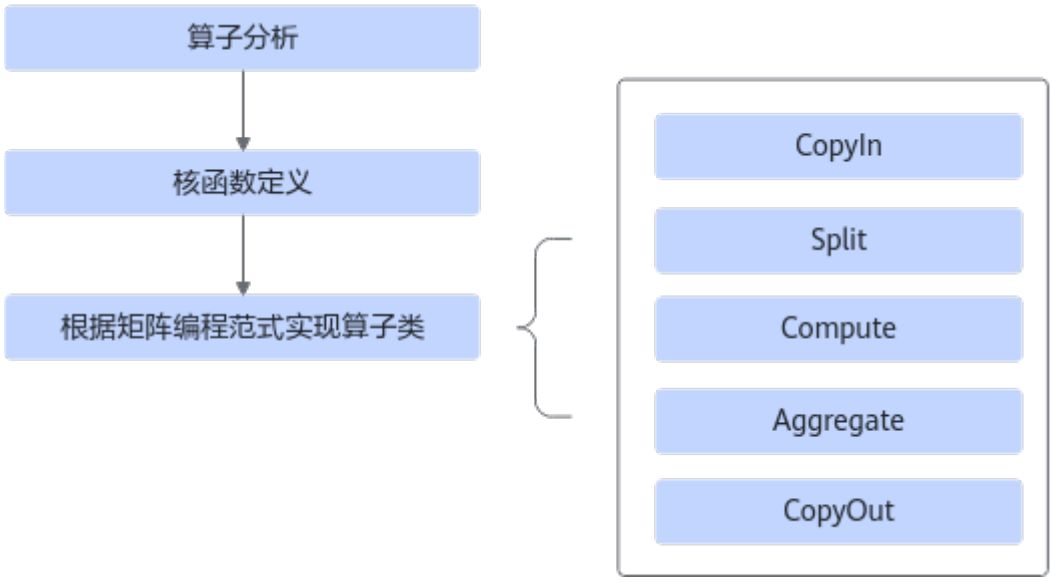
图 7-21 矩阵编程 Queue 队列



开发流程

基于Ascend C方式实现矩阵算子的流程如下图所示。

图 7-22 矩阵算子实现流程



- 算子分析：分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。
- 核函数定义：定义Ascend C算子入口函数。
- 根据矩阵编程范式实现算子类：完成核函数的内部实现，调用私有成员函数CopyIn、SplitA、SplitB、Compute、Aggregate、CopyOut完成矩阵算子的五级流水操作。

下文将以Matmul算子为例对上述步骤进行详细介绍，Matmul算子的代码框架如下，完整代码请参见**7.4.5 实现样例**。

```
#include "kernel_operator.h"

// 根据编程范式实现算子类
class KernelMatmul {
public:
    __aicore__ inline void Init(GM_ADDR a, GM_ADDR b, GM_ADDR c)
    {
        // ...
    }
    __aicore__ inline void Process()
    {
        CopyIn();
        SplitA();
        AscendC::LocalTensor<half> b1Local = inQueueB1.DeQue<half>();
        AscendC::LocalTensor<half> a2Local = inQueueA2.DeQue<half>();
        AscendC::LocalTensor<float> c2Local = outQueueCO2.AllocTensor<float>();
        // split matrix b into 2 parts, [32, 16] and [32, 16]
        for (int i = 0; i < 2; ++i) {
            SplitB(b1Local, i);
            Compute(a2Local);
            Aggregate(c2Local, i);
        }
        inQueueB1.FreeTensor(b1Local);
        inQueueA2.FreeTensor(a2Local);
        outQueueCO2.EnQue<float>(c2Local);
        CopyOut();
    }
private:
    __aicore__ inline void CopyIn()
    {
        // ...
```



```
    }
    __aicore__ inline void SplitA()
    {
        // ...
    }
    __aicore__ inline void SplitB(const LocalTensor<half>& b1Local, const int bSplitIdx)
    {
        // ...
    }
    __aicore__ inline void Compute(const LocalTensor<half>& a2Local)
    {
        // ...
    }
    __aicore__ inline void Aggregate(const LocalTensor<float>& c2Local, const int bSplitIdx)
    {
        // ...
    }
    __aicore__ inline void CopyOut()
    {
        // ...
    }
private:
    // ...

};

//核函数定义
extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c)
{
    KernelMatmul op;
    op.Init(a, b, c);
    op.Process();
}
```

7.4.2 算子分析

在开发算子代码之前需要分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。

步骤1 明确算子的数学表达式及计算逻辑。

Matmul算子完成矩阵乘操作，其数学表达式如下，形状为[m, k]的矩阵a和形状为[k, n]的矩阵b相乘，得到形状为[m, n]的矩阵c。为了方便，令m=k=n=32。

```
c = a * b
```

注意需要处理的数据过大时，需要对数据进行切分并分块搬运到A2、B2，分别计算后再进行汇聚。下文的计算逻辑为了展示Split和Aggregate阶段的样例，请您根据实际需要处理的数据大小决定是否需要切分和汇聚。

计算逻辑如下：

1. 分别搬运输入数据矩阵a、b至Local Memory A1、B1。
2. 将a矩阵从A1搬运至A2。将b矩阵切分为part1和part2，形状均为[k, n / 2]，切分后再分块搬运至B2。
3. a矩阵和b矩阵part1、part2分别做矩阵乘运算，获得矩阵c的part1和part2，形状均为[m, n / 2]。计算结果在CO1存储。
4. 将矩阵c的part1和part2分别拷贝到CO2进行合并。
5. 将合并后的输出数据从CO2搬出。

步骤2 明确输入和输出。

- Matmul算子有两个输入：a与b，输出为c。

- 本样例中算子输入支持的数据类型为half（float16），算子输出的数据类型为float32。
- 矩阵a、b、c的形状均为[32, 32]。
- 算子输入输出支持的数据格式为：ND。

步骤3 确定核函数名称和参数。

- 您可以自定义核函数名称，本样例中核函数命名为matmul_custom。
- 根据对算子输入输出的分析，确定核函数有3个参数a，b，c；a，b为输入在Global Memory上的内存地址，c为输出在Global Memory上的内存地址。

步骤4 约束分析。

由于硬件架构对矩阵乘计算的输入输出有格式约束，需要在算子实现中增加格式转换的流程。

- 搬运矩阵a、b至A1、B1时，将ND格式的矩阵a、b转换为NZ格式。
- 从A1搬运矩阵a至A2时，将NZ格式的a矩阵转换为ZZ格式；从B1搬运矩阵b到B2时将NZ格式的b矩阵转换为ZN格式。
- 将计算结果从CO2搬出时，将NZ格式的c矩阵转换为ND格式。
- 数据排布格式的相关介绍详见**数据排布格式**。

步骤5 确定算子实现所需接口。

- 实现外部存储和内部存储间的数据搬运，查看Ascend C API参考中的数据搬移接口，具体参考DataCopy。
- 实现矩阵数据格式转换，查看Ascend C API参考中的数据转换接口，具体参考LoadData。
- 矩阵计算过程涉及矩阵乘法，查看Ascend C API参考中的矩阵计算接口，具体参考Mmad。
- 计算中使用到的Tensor数据结构，使用Queue队列进行管理，会使用到EnQue、DeQue等接口。

----结束

通过以上分析，得到Ascend C Matmul算子的计算流程图和设计规格如下：

图 7-23 Matmul 算子的计算流程图

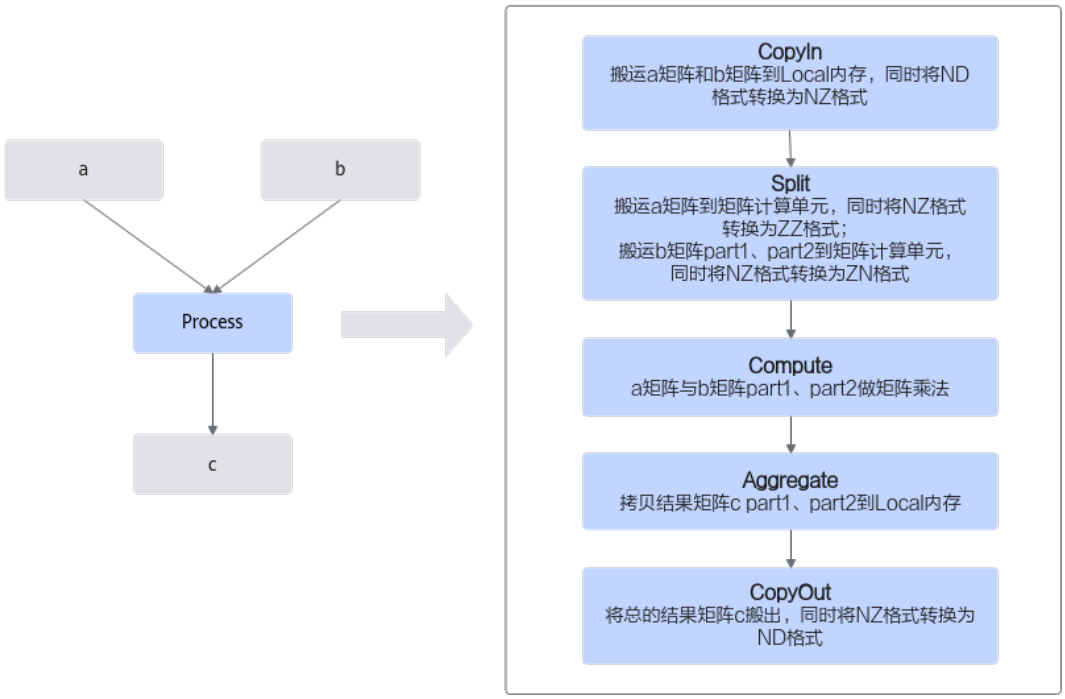


表 7-2 Ascend C Matmul 算子设计规格

算子类型（OpType）	Matmul			
算子输入	**name**	**shape**	**data type**	**format**
	a	(m, k) = (32, 32)	half	ND
	b	(k, n) = (32, 32)	half	ND
算子输出	c	(m, n) = (32, 32)	float32	ND
核函数名称	matmul_custom			
使用的主要接口	DataCopy：数据搬移接口			
	LoadData：矩阵数据格式转换接口			
	Mmad：矩阵乘计算接口			
	EnQue、DeQue等接口：Queue队列管理接口			
算子实现文件名称	matmul_custom.cpp			

7.4.3 核函数定义

根据**核函数定义**中介绍的规则进行核函数的定义。

步骤1 函数原型定义。

本样例中，函数名为matmul_custom（核函数名称可自定义）；根据**7.4.2 算子分析**中对算子输入输出的分析，确定有3个参数a，b，c，其中a，b都为输入内存，c为输出内存。根据**核函数定义**核函数的规则介绍，函数原型定义如下所示：使用__global__函数类型限定符来标识它是一个核函数，可以被<<<>>>调用；使用__aicore__函数类型限定符来标识该核函数在设备端aicore上执行；为方便起见，统一使用GM_ADDR宏修饰入参，GM_ADDR宏定义请参考**核函数定义**。

```
extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c)
{
}
```

步骤2 调用算子类的Init和Process函数。

算子类的Init函数，完成内存初始化相关工作，Process函数完成算子实现的核心逻辑，具体介绍参见**7.4.4 算子类实现**。

```
extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c)
{
    KernelMatmul op;
    op.Init(a, b, c);
    op.Process();
}
```

步骤3 对核函数进行封装，得到matmul_custom_do函数，便于主程序调用。#ifndef ASCENDC_CPU_DEBUG表示该封装函数仅在编译运行NPU侧的算子时会用到，编译运行CPU侧的算子时，可以直接调用matmul_custom函数。根据**核函数调用**章节，调用核函数时，除了需要传入参数a，b，c，还需要传入blockDim（核函数执行的核数），l2ctrl（保留参数，设置为nullptr），stream（应用程序中维护异步操作执行顺序的stream）来规定核函数的执行配置。

```
#ifndef ASCENDC_CPU_DEBUG
// call of kernel function
void matmul_custom_do(uint32_t blockDim, void* l2ctrl, void* stream, uint8_t* a, uint8_t* b, uint8_t* c)
{
    matmul_custom<<<blockDim, l2ctrl, stream>>>(a, b, c);
}
#endif
```

----结束

7.4.4 算子类实现

根据上一章节介绍，核函数中会调用算子类的Init和Process函数，本章具体讲解基于编程范式实现算子类。矩阵编程范式请参考**编程范式**。

算子类中主要包含对外开放的初始化Init函数和核心处理函数Process以及一些实现中会用到的私有成员。KernelMatmul算子类的定义如下：

```
class KernelMatmul {
public:
    __aicore__ inline KernelMatmul(){}
    // 初始化函数，完成内存初始化相关操作
    __aicore__ inline void Init(GM_ADDR a, GM_ADDR b, GM_ADDR c){}
    // 核心处理函数，实现算子逻辑
    // 调用私有成员函数CopyIn、SplitA、SplitB、Compute、Aggregate、CopyOut完成矩阵算子的五级流水操作
    __aicore__ inline void Process(){}

private:
    __aicore__ inline void CopyND2NZ(const LocalTensor<half>& dst, const GlobalTensor<half>& src, const
```



```
uint16_t height, const uint16_t width){}
    // 搬进函数，完成编程范式中的CopyIn阶段的处理，由Process函数调用
    __aicore__ inline void CopyIn(){}
    // 搬进函数，完成编程范式中的Split阶段的处理，由Process函数调用
    __aicore__ inline void SplitA(){}
    // 搬进函数，完成编程范式中的Split阶段的处理，由Process函数循环调用两次，分别搬运b矩阵的两个part
    __aicore__ inline void SplitB(const LocalTensor<half>& b1Local, const int bSplitIdx){}
    // 计算函数，完成编程范式中的Compute阶段的处理，由Process函数循环调用两次，分别计算出矩阵c的两个
part
    __aicore__ inline void Compute(const LocalTensor<half>& a2Local){}
    // 搬出函数，完成编程范式中的Aggregate阶段的处理，由Process函数循环调用两次，分别搬出矩阵c的两个
part
    __aicore__ inline void Aggregate(const LocalTensor<float>& c2Local, const int bSplitIdx){}
    // 搬出函数，完成编程范式中的CopyOut阶段的处理，由Process函数调用
    __aicore__ inline void CopyOut(){}

private:
    AscendC::TPipe pipe;  // Pipe内存管理对象，管理Queue队列的内存
    AscendC::TQue<AscendC::QuePosition::A1, 1> inQueueA1;  // 输入数据的队列，QuePosition为A1
    AscendC::TQue<AscendC::QuePosition::A2, 1> inQueueA2;  // 输入数据的队列，QuePosition为A2
    AscendC::TQue<AscendC::QuePosition::B1, 1> inQueueB1;  // 输入数据的队列，QuePosition为B1
    AscendC::TQue<AscendC::QuePosition::B2, 2> inQueueB2;  // 输入数据的队列，QuePosition为B2
    AscendC::TQue<AscendC::QuePosition::CO1, 2> outQueueCO1;  // 输出数据的队列，QuePosition为CO1
    AscendC::TQue<AscendC::QuePosition::CO2, 1> outQueueCO2;  // 输出数据的队列，QuePosition为CO2
    // 管理输入输出Global Memory内存地址的对象，其中aGM，bGM为输入，cGM为输出
    AscendC::GlobalTensor<half> aGM, bGM;
    AscendC::GlobalTensor<float> cGM;

    uint16_t m = 32;
    uint16_t n = 32;
    uint16_t k = 32;
    uint16_t aSize, bSize, cSize, mBlocks, nBlocks, kBlocks;
};
```

KernelMatmul 构造函数实现

构造函数中对私有成员变量进行初始化，具体代码如下：

```
__aicore__ inline KernelMatmul()
{
    aSize = m * k;
    bSize = k * n;
    cSize = m * n;
    mBlocks = m / 16;
    nBlocks = n / 16;
    kBlocks = k / 16;
}
```

矩阵a的形状为[m, k]，矩阵b的形状为[k, n]，矩阵c的形状为[m,n]，此样例中m、n、k均设置为32。

aSize、bSize、cSize分别为矩阵a、b、c的数值个数。

mBlocks、 nBlocks、 kBlocks为m、n、k所占分形数量，half类型一个分形长度为16，blocks计算公式为：

- mBlocks = m / 16
- nBlocks = n / 16
- kBlocks = k / 16

分形具体介绍可参考**14.1.2 数据排布格式**。

Init 函数实现

Init函数主要完成以下内容：

- 设置输入输出Global Tensor的Global Memory内存地址。

  以设置输入a在Global Memory上的内存偏移地址为例：

  ```
  aGM.SetGlobalBuffer((__gm__ half*)a);
  ```

  注意，因为本样例中Init函数的入参统一设置为uint8_t*，这里需要强转成具体的数据类型(__gm__ half*)，再进行偏移。

- 通过Pipe内存管理对象为输入输出Queue分配内存。

  比如，为输入数据队列inQueueB2分配内存，可以通过如下代码段实现：

  ```
  pipe.InitBuffer(inQueueB2, 2, bSize * sizeof(half) / 2);
  ```

  此样例中将b矩阵切分为两个part，为inQueueB2分配内存时需要申请两块内存空间，每一块的大小为b矩阵大小的一半，outQueueCO1的内存初始化同理。

具体的初始化函数代码如下：

```
__aicore__ inline void Init(GM_ADDR a, GM_ADDR b, GM_ADDR c)
{
    aGM.SetGlobalBuffer((__gm__ half*)a);
    bGM.SetGlobalBuffer((__gm__ half*)b);
    cGM.SetGlobalBuffer((__gm__ float*)c);
    pipe.InitBuffer(inQueueA1, 1, aSize * sizeof(half));
    pipe.InitBuffer(inQueueA2, 1, aSize * sizeof(half));
    pipe.InitBuffer(inQueueB1, 1, bSize * sizeof(half));
    pipe.InitBuffer(inQueueB2, 2, bSize * sizeof(half) / 2);
    pipe.InitBuffer(outQueueCO1, 2, cSize * sizeof(float) / 2);
    pipe.InitBuffer(outQueueCO2, 1, cSize * sizeof(float));
}
```

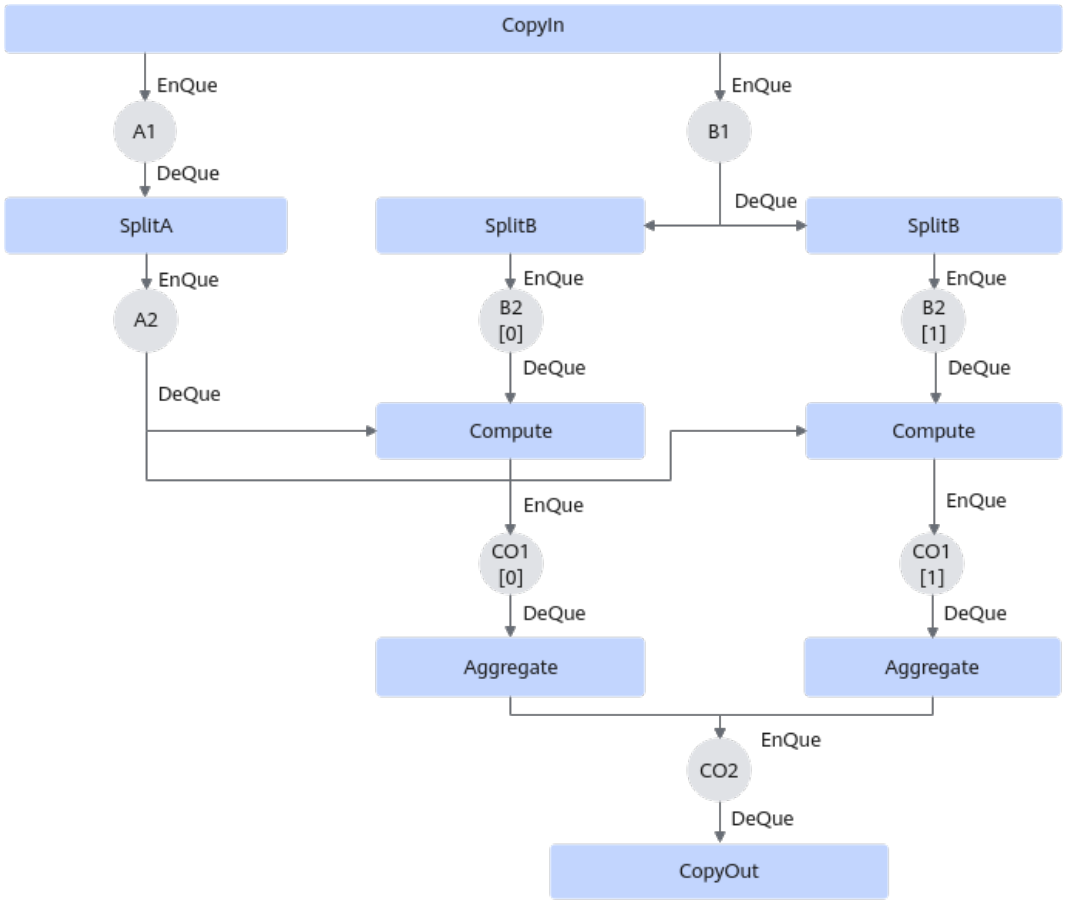
Process 函数实现

基于矩阵编程范式，将核函数的实现分为5个基本阶段：CopyIn，Split，Compute，Aggregate，CopyOut。Split，Compute，Aggregate阶段需要区分a、b矩阵。Process函数中通过如下方式调用这几个函数。

```
__aicore__ inline void Process()
{
    CopyIn();
    SplitA();
    AscendC::LocalTensor<half> b1Local = inQueueB1.DeQue<half>();
    AscendC::LocalTensor<half> a2Local = inQueueA2.DeQue<half>();
    AscendC::LocalTensor<float> c2Local = outQueueCO2.AllocTensor<float>();
    // split matrix b into 2 parts, [32, 16] and [32, 16]
    for (int i = 0; i < 2; ++i) {
        SplitB(b1Local, i);
        Compute(a2Local);
        Aggregate(c2Local, i);
    }
    inQueueB1.FreeTensor(b1Local);
    inQueueA2.FreeTensor(a2Local);
    outQueueCO2.EnQue<float>(c2Local);
    CopyOut();
}
```

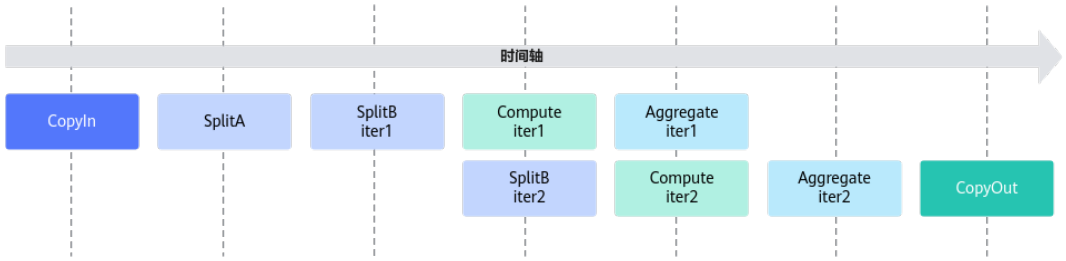
两次循环内，SplitB需要从inQueueB1中分别搬运两个part的b矩阵，Compute需要分别计算a矩阵和两个part b矩阵的乘法，Aggregate要分别搬运两个part的c矩阵，具体五个阶段数据流通示意图如下：

图 7-24 数据流通示意图



切分b矩阵，可以实现一部分的并行，本样例的流水并行示意图如下：

图 7-25 并行示意图



步骤1 Stage1：CopyIn函数实现。

1. 使用AllocTensor从A1，B1的Queue中申请a1Local和b1Local。
2. 使用DataCopy接口将矩阵a、b搬运到Local Memmory，同时将其数据格式从ND转换为NZ。

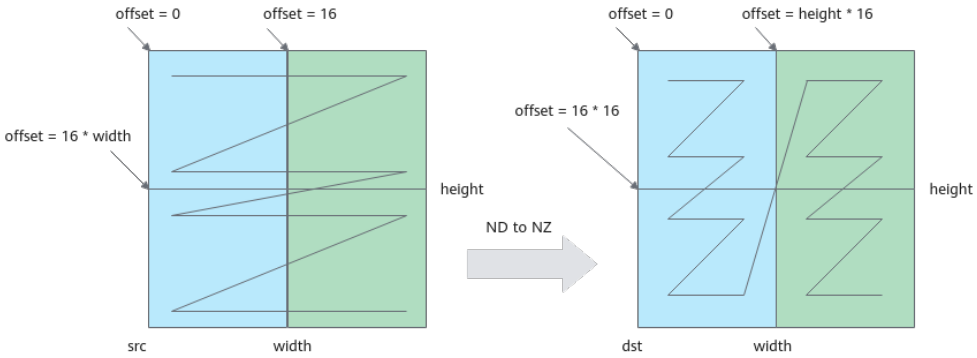
   一次DataCopy指令搬运height*16个数，循环执行width/16次。DataCopy的参数设置如下：

   - blockCount设置为height，共搬运height次。
   - blockLen设置为1，一次搬运16个类型为half的数。
   - srcStride设置为width/16 - 1，源矩阵每搬运一个block需要跳跃一行。

- dstStride设置为0，目的矩阵每个block在内存中连续存储。
- 每次循环迭代，源矩阵首地址移动16个数，目的矩阵首地址移动16*height个数。

格式转换示意图如下，第一次循环搬运蓝色部分，第二次循环搬运绿色部分；图中width为32，占两个分形，height为32，占两个分形，一共搬运4个16*16分形。

图 7-26 ND to NZ 转换示意图



注意：上述ND到NZ的格式转换仅作为举例说明，开发者可根据实际情况选择合适的转换方式。

3. 使用EnQue将a1Local、b1Local分别放入A1、B1的Queue中。

具体代码如下：

```
__aicore__ inline void CopyND2NZ(const LocalTensor<half>& dst, const GlobalTensor<half>& src, const uint16_t height, const uint16_t width)
{
    for (int i = 0; i < width / 16; ++i) {
        int srcOffset = i * 16;
        int dstOffset = i * 16 * height;
        AscendC::DataCopy(dst[dstOffset], src[srcOffset], { height, 1, uint16_t(width / 16 - 1), 0 });
    }
}
__aicore__ inline void CopyIn()
{
    AscendC::LocalTensor<half> a1Local = inQueueA1.AllocTensor<half>();
    AscendC::LocalTensor<half> b1Local = inQueueB1.AllocTensor<half>();
    CopyND2NZ(a1Local, aGM, m, k);
    CopyND2NZ(b1Local, bGM, k, n);
    inQueueA1.EnQue(a1Local);
    inQueueB1.EnQue(b1Local);
}
```

步骤2 Stage2：SplitA函数实现。

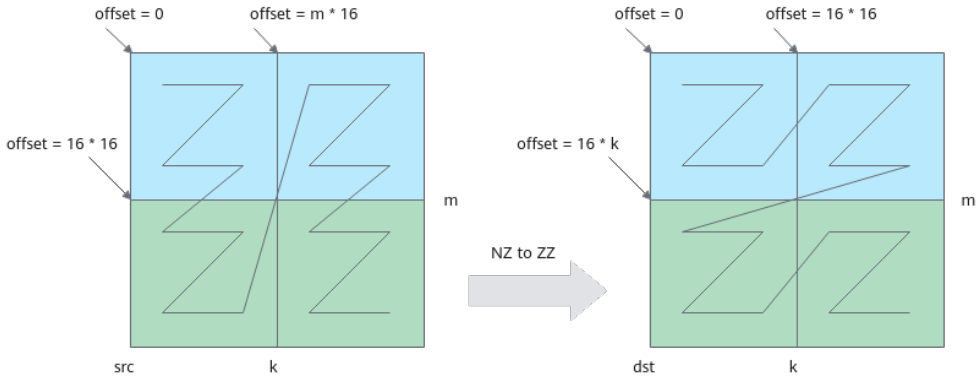
1. 使用DeQue从A1的Queue中取出a1Local。
2. 使用AllocTensor从A2的Queue中申请a2Local。
3. 使用LoadData将矩阵a搬运到A2，同时将a矩阵从NZ格式转换为ZZ格式。

   搬运及格式转换示意图如下：图中k为32，占kBlocks（k/16=2）个分形，m为32，占mBlocks（m/16=2）个分形，一共搬运4个16*16分形。本示例中，调用一次LoadData接口完成两个16*16分形的搬运，循环调用两次LoadData。第一次循环搬运蓝色部分两个分形，第二次循环搬运绿色部分两个分形。

   单次循环中LoadData（本样例中要完成2个分形的搬运，蓝色部分或者绿色部分）的参数设置如下：

- repeatTimes表示数据处理的迭代次数，因为LoadData每个迭代处理一个分形，所以也可以理解为待搬运分形的个数。本样例中即为k轴方向的分形个数，设置为kBlocks，表示搬运kBlocks个分形。
- srcStride表示，相邻迭代间源操作数分形首地址之间的间隔，以搬运蓝色部分分形为例：下图中左侧源操作数矩阵，第一个蓝色分形和第二个蓝色分形起始地址之间的间隔为mBlocks个分形，此处设置为mBlocks。
- dstStride使用默认值，目的矩阵两个分形连续存储。
- ifTranspose设置为false，每块分形搬运前搬运后都为Z格式，不使能转置。
- 每次循环迭代目的矩阵首地址偏移16*k，源矩阵首地址偏移16*16。

图 7-27 NZ to ZZ 格式转换示意图



4. 使用EnQue将计算结果a2Local放入到A2的Queue中。
   具体代码如下：

```
__aicore__ inline void SplitA()
{
    int srcOffset = 0;
    int dstOffset = 0;
    AscendC::LocalTensor<half> a1Local = inQueueA1.DeQue<half>();
    AscendC::LocalTensor<half> a2Local = inQueueA2.AllocTensor<half>();

    // transform nz to zz
    for (int i = 0; i < mBlocks; ++i) {
        AscendC::LoadData2dParams loadDataParams;
        loadDataParams.repeatTimes = kBlocks;
        loadDataParams.srcStride = mBlocks;
        loadDataParams.ifTranspose = false;

        LoadData(a2Local[dstOffset], a1Local[srcOffset], loadDataParams);

        srcOffset += 16 * 16;
        dstOffset += k * 16;
    }
    inQueueA2.EnQue<half>(a2Local);
    inQueueA1.FreeTensor(a1Local);
}
```

步骤3 Stage2：SplitB函数实现。

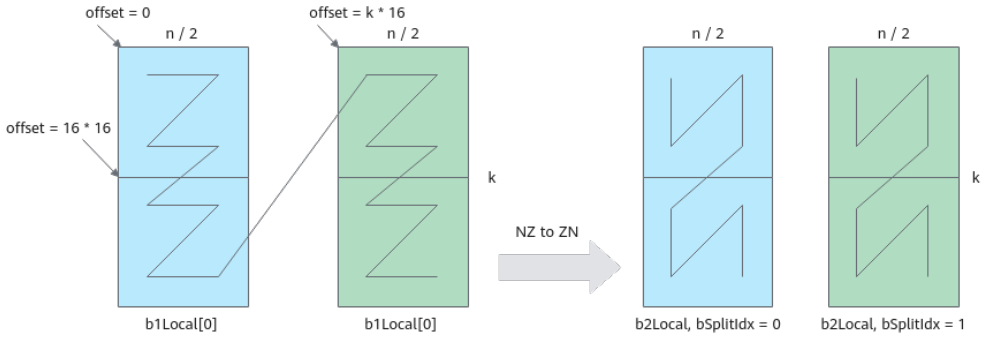
1. SplitB需要传入两个参数：使用DeQue从B1的Queue中取出的b1Local和循环迭代变量index。
2. 使用AllocTensor从B2的Queue中申请b2Local。
3. 使用LoadData将b矩阵搬运到B2，同时从NZ格式转换为ZN格式。
   搬运及格式转换示意图如下：图中k为32，占kBlocks（k/16=2）个分形，n为32，占nBlocks（n/16=2）个分形，一共搬运4个16*16分形。本示例中，调用一

次LoadData接口完成两个16*16分形的搬运，循环调用两次LoadData。第一次循环搬运蓝色部分两个分形，第二次循环搬运绿色部分两个分形。

单次循环中LoadData（本样例中要完成2个分形的搬运，蓝色部分或者绿色部分）的参数设置如下：

- repeatTimes表示数据处理的迭代次数，因为LoadData每个迭代处理一个分形，所以也可以理解为待搬运分形的个数。本样例中即为k轴方向的分形个数，设置为kBlocks，表示搬运kBlocks个分形。
- srcStride相邻迭代间源操作数分形首地址之间的间隔，以搬运蓝色部分分形为例：下图中左侧源操作数矩阵，第一个蓝色分形和第二个蓝色分形起始地址之间的间隔为1个分形，此处设置为1，源矩阵两个分形连续存储。
- dstStride使用默认值，目的矩阵两个分形连续存储。
- ifTranspose设置为true，每块分形搬运前为Z格式，搬运后需要为N格式，需要使能转置。
- 每次循环迭代，源矩阵首地址需要偏移bSize/2。

图 7-28 NZ to ZN 格式转换示意图



4. 使用EnQue将计算结果b2Local放入到B2的Queue中。

具体代码如下：

```
__aicore__ inline void SplitB(const LocalTensor<half>& b1Local, const int bSplitIdx)
{
    AscendC::LocalTensor<half> b2Local = inQueueB2.AllocTensor<half>();
    // transform nz to zn
    AscendC::LoadData2dParams loadDataParams;
    loadDataParams.repeatTimes = kBlocks;
    loadDataParams.srcStride = 1;
    loadDataParams.ifTranspose = true;

    AscendC::LoadData(b2Local, b1Local[bSplitIdx * bSize / 2], loadDataParams);

    inQueueB2.EnQue<half>(b2Local);
}
```

步骤4 Stage3：Compute函数实现，完成核心的矩阵计算功能。

1. Compute函数需要传入参数a2Local，a2Local从A2的Queue中使用DeQue取出。
2. 使用AllocTensor从CO1的Queue中申请c1Local。
3. 使用DeQue从B2中取出b2Local。
4. 使用Ascend C接口Mmad完成矩阵乘计算。
5. 使用EnQue将计算结果c1Local放入到CO1的Queue中。

具体代码如下：


```
__aicore__ inline void Compute(const LocalTensor<half>& a2Local)
{
    AscendC::LocalTensor<half> b2Local = inQueueB2.DeQue<half>();
    AscendC::LocalTensor<float> c1Local = outQueueCO1.AllocTensor<float>();
    AscendC::Mmad(c1Local, a2Local, b2Local, { m, uint16_t(n / 2), k, 0, false, true });
    outQueueCO1.EnQue<float>(c1Local);
    inQueueB2.FreeTensor(b2Local);
}
```

步骤5 Stage4：Aggregate函数实现，完成数据汇聚操作。

1. Aggregate需要传入两个参数：使用AllocTensor从CO2的Queue中申请的c2Local和循环迭代变量index。
2. 使用DeQue从CO1中取出c1Local。
3. 使用DataCopy将结果矩阵从CO1搬运到CO2。

   DataCopy参数设置如下：

   - blockCount设置为1，blockLen设置为2，连续搬运两个分形，无需格式转换。
   - blockMode设置为BlockMode::BLOCK_MODE_MATRIX，表示需要按分形搬运。
   - c2Local首地址偏移量设置为index * cSize / 2。

具体代码如下：

```
__aicore__ inline void Aggregate(const LocalTensor<float>& c2Local, const int bSplitIdx)
{
    AscendC::LocalTensor<float> c1Local = outQueueCO1.DeQue<float>();
    AscendC::DataCopyParams dataCopyParams;
    dataCopyParams.blockCount = 1;
    dataCopyParams.blockLen = 2;
    AscendC::DataCopyEnhancedParams enhancedParams;
    enhancedParams.blockMode = BlockMode::BLOCK_MODE_MATRIX;

    AscendC::DataCopy(c2Local[bSplitIdx * cSize / 2], c1Local, dataCopyParams, enhancedParams);

    outQueueCO1.FreeTensor(c1Local);
}
```

步骤6 Stage5：CopyOut函数实现。

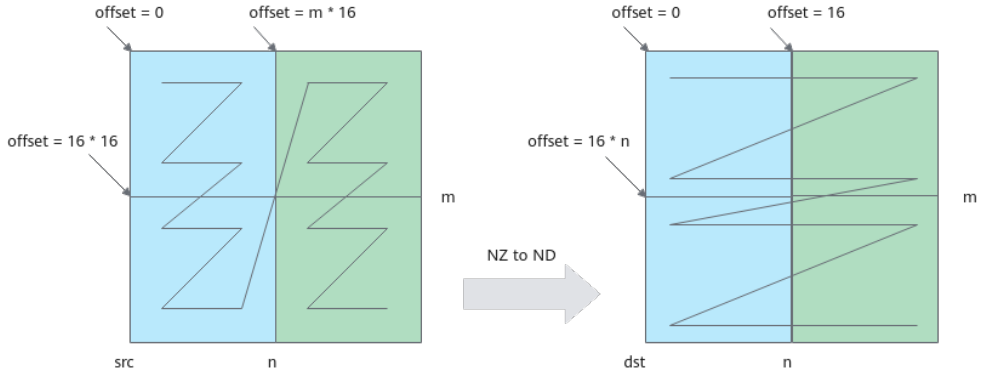
1. 使用DeQue从CO2中取出c2Local。
2. 使用DataCopy将结果矩阵从CO2搬运到Global Memory，同时需要将格式从NZ转换为ND。

   每次循环移动一个分形，搬运m*16个数。DataCopy参数说明如下：

   - blockCount设置为m，共搬运m次。
   - blockLen设置为2，DataCopy指令一次搬运16个数，共2个block。
   - srcStride设置为0，每两次搬运间没有间隙。
   - dstStride设置为(nBlocks - 1) * 2，每两次搬运间隔2个block。
   - 每次循环迭代，目的矩阵偏移16，源矩阵偏移m*16。

   格式转换示意图如下，第一次循环搬运蓝色部分数据，第二次循环搬运绿色部分数据。

图 7-29 NZ to ND 格式转换示意图



具体代码如下：

```
__aicore__ inline void CopyOut()
{
    AscendC::LocalTensor<float> c2Local = outQueueCO2.DeQue<float>();
    // transform nz to nd
    for (int i = 0; i < nBlocks; ++i) {
        AscendC::DataCopy(cGM[i * 16], c2Local[i * m * 16], { m, 2, 0, uint16_t((nBlocks - 1) * 2) });
    }
    outQueueCO2.FreeTensor(c2Local);
}
```

----结束

7.4.5 实现样例

Ascend C Matmul算子实现文件：matmul_custom.cpp

```
/*
 * Copyright (c) Huawei Technologies Co., Ltd. 2022-2023. All rights reserved.
 *
 * Function : c = a * b (matrix multiplication)
 * This sample is a very basic sample that implements Matmul on Ascend platform.
 * In this sample:
 * Shape of matrix a is [m, k]: [32, 32]
 * Shape of matrix b is [k, n]: [32, 32]
 * Shape of matrix c is [m, n]: [32, 32]
 */

#include "kernel_operator.h"

class KernelMatmul
{
public:
    __aicore__ inline KernelMatmul()
    {
        aSize = m * k;
        bSize = k * n;
        cSize = m * m;
        mBlocks = m / 16;
        nBlocks = n / 16;
        kBlocks = k / 16;
    }
    __aicore__ inline void Init(GM_ADDR a, GM_ADDR b, GM_ADDR c)
    {
        aGM.SetGlobalBuffer((__gm__ half *)a);
        bGM.SetGlobalBuffer((__gm__ half *)b);
        cGM.SetGlobalBuffer((__gm__ float *)c);
        pipe.InitBuffer(inQueueA1, 1, aSize * sizeof(half));
        pipe.InitBuffer(inQueueA2, 1, aSize * sizeof(half));
```

```
        pipe.InitBuffer(inQueueB1, 1, bSize * sizeof(half));
        pipe.InitBuffer(inQueueB2, 2, bSize * sizeof(half) / 2);
        pipe.InitBuffer(outQueueCO1, 2, cSize * sizeof(float) / 2);
        pipe.InitBuffer(outQueueCO2, 1, cSize * sizeof(float));
    }
    __aicore__ inline void Process()
    {
        CopyIn();
        SplitA();

        AscendC::LocalTensor<half> b1Local = inQueueB1.DeQue<half>();
        AscendC::LocalTensor<half> a2Local = inQueueA2.DeQue<half>();
        AscendC::LocalTensor<float> c2Local = outQueueCO2.AllocTensor<float>();
        // split matrix b into 2 parts, [32, 16] and [32, 16]
        for (int i = 0; i < 2; ++i)
        {
            SplitB(b1Local, i);
            Compute(a2Local);
            Aggregate(c2Local, i);
        }
        inQueueB1.FreeTensor(b1Local);
        inQueueA2.FreeTensor(a2Local);
        outQueueCO2.EnQue<float>(c2Local);

        CopyOut();
    }

private:
    __aicore__ inline void CopyND2NZ(const AscendC::LocalTensor<half> &dst, const
AscendC::GlobalTensor<half> &src, const uint16_t height,
                                     const uint16_t width)
    {
        for (int i = 0; i < width / 16; ++i)
        {
            int srcOffset = i * 16;
            int dstOffset = i * 16 * height;
            AscendC::DataCopy(dst[dstOffset], src[srcOffset], {height, 1, uint16_t(width / 16 - 1), 0});
        }
    }
    __aicore__ inline void CopyIn()
    {
        AscendC::LocalTensor<half> a1Local = inQueueA1.AllocTensor<half>();
        AscendC::LocalTensor<half> b1Local = inQueueB1.AllocTensor<half>();

        CopyND2NZ(a1Local, aGM, m, k);
        CopyND2NZ(b1Local, bGM, k, n);

        inQueueA1.EnQue(a1Local);
        inQueueB1.EnQue(b1Local);
    }
    __aicore__ inline void SplitA()
    {
        int srcOffset = 0;
        int dstOffset = 0;
        AscendC::LocalTensor<half> a1Local = inQueueA1.DeQue<half>();
        AscendC::LocalTensor<half> a2Local = inQueueA2.AllocTensor<half>();

        // transform nz to zz
        for (int i = 0; i < mBlocks; ++i)
        {
            AscendC::LoadData2dParams loadDataParams;
            loadDataParams.repeatTimes = kBlocks;
            loadDataParams.srcStride = mBlocks;
            loadDataParams.ifTranspose = false;

            LoadData(a2Local[dstOffset], a1Local[srcOffset], loadDataParams);

            srcOffset += 16 * 16;
            dstOffset += k * 16;
```



```
        }

        inQueueA2.EnQue<half>(a2Local);
        inQueueA1.FreeTensor(a1Local);
    }
    __aicore__ inline void SplitB(const AscendC::LocalTensor<half> &b1Local, const int bSplitIdx)
    {
        AscendC::LocalTensor<half> b2Local = inQueueB2.AllocTensor<half>();

        // transform nz to zn
        AscendC::LoadData2dParams loadDataParams;
        loadDataParams.repeatTimes = kBlocks;
        loadDataParams.srcStride = 1;
        loadDataParams.ifTranspose = true;

        AscendC::LoadData(b2Local, b1Local[bSplitIdx * bSize / 2], loadDataParams);

        inQueueB2.EnQue<half>(b2Local);
    }
    __aicore__ inline void Compute(const AscendC::LocalTensor<half> &a2Local)
    {
        AscendC::LocalTensor<half> b2Local = inQueueB2.DeQue<half>();
        AscendC::LocalTensor<float> c1Local = outQueueCO1.AllocTensor<float>();

        AscendC::Mmad(c1Local, a2Local, b2Local, {m, uint16_t(n / 2), k, false, 0, false, false, false});

        outQueueCO1.EnQue<float>(c1Local);
        inQueueB2.FreeTensor(b2Local);
    }
    __aicore__ inline void Aggregate(const AscendC::LocalTensor<float> &c2Local, const int bSplitIdx)
    {
        AscendC::LocalTensor<float> c1Local = outQueueCO1.DeQue<float>();

        AscendC::DataCopyParams dataCopyParams;
        dataCopyParams.blockCount = 1;
        dataCopyParams.blockLen = 2;
        AscendC::DataCopyEnhancedParams enhancedParams;
        enhancedParams.blockMode = AscendC::BlockMode::BLOCK_MODE_MATRIX;
        AscendC::DataCopy(c2Local[bSplitIdx * cSize / 2], c1Local, dataCopyParams, enhancedParams);

        outQueueCO1.FreeTensor(c1Local);
    }
    __aicore__ inline void CopyOut()
    {
        AscendC::LocalTensor<float> c2Local = outQueueCO2.DeQue<float>();

        // transform nz to nd
        for (int i = 0; i < nBlocks; ++i)
        {
            AscendC::DataCopy(cGM[i * 16], c2Local[i * m * 16], {m, 2, 0, uint16_t((nBlocks - 1) * 2)});
        }

        outQueueCO2.FreeTensor(c2Local);
    }

private:
    AscendC::TPipe pipe;

    AscendC::TQue<AscendC::QuePosition::A1, 1> inQueueA1;
    AscendC::TQue<AscendC::QuePosition::A2, 1> inQueueA2;
    AscendC::TQue<AscendC::QuePosition::B1, 1> inQueueB1;
    AscendC::TQue<AscendC::QuePosition::B2, 2> inQueueB2;
    // dst queue
    AscendC::TQue<AscendC::QuePosition::CO1, 2> outQueueCO1;
    AscendC::TQue<AscendC::QuePosition::CO2, 1> outQueueCO2;

    AscendC::GlobalTensor<half> aGM, bGM;
    AscendC::GlobalTensor<float> cGM;
```



```
    uint16_t m = 32;
    uint16_t n = 32;
    uint16_t k = 32;

    uint16_t aSize, bSize, cSize, mBlocks, nBlocks, kBlocks;
};

extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c)
{
    KernelMatmul op;
    op.Init(a, b, c);
    op.Process();
}

#ifndef ASCENDC_CPU_DEBUG
// call of kernel function
void matmul_custom_do(uint32_t blockDim, void *l2ctrl, void *stream, uint8_t *a, uint8_t *b, uint8_t *c)
{
    matmul_custom<<<blockDim, l2ctrl, stream>>>(a, b, c);
}
#endif
```

7.5 融合算子编程

7.5.1 基础知识

说明

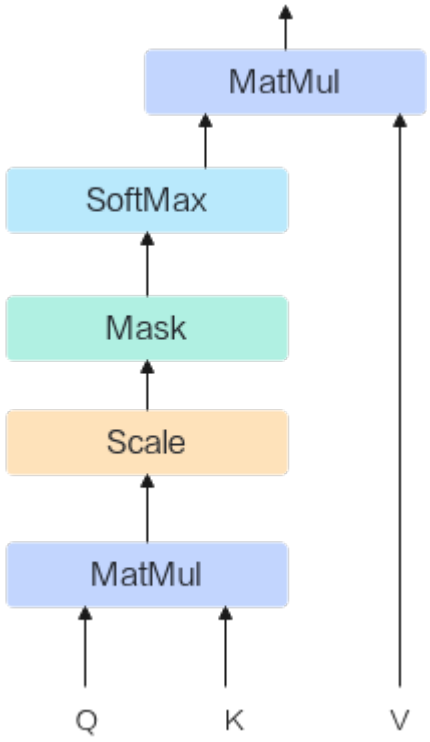
学习融合算子编程之前，请确保已经掌握**矩阵编程**相关知识。

融合算子

融合算子是指将多个独立的“小算子”融合起来成为一个“大算子”，多个小算子的功能和大算子的功能等价，大算子的性能优于独立的小算子。可以根据具体算法的实现自由融合Vector、Cube算子以达到性能上的收益。

比如对于LLM大模型中最核心的一个融合算子Flash Attention， 其核心实现如下图。图中的MatMul算子（Cube）、Scale算子（Vector）、Mask算子（Vector）、SoftMax算子（Vector）融合为一个大的算子Flash Attention。

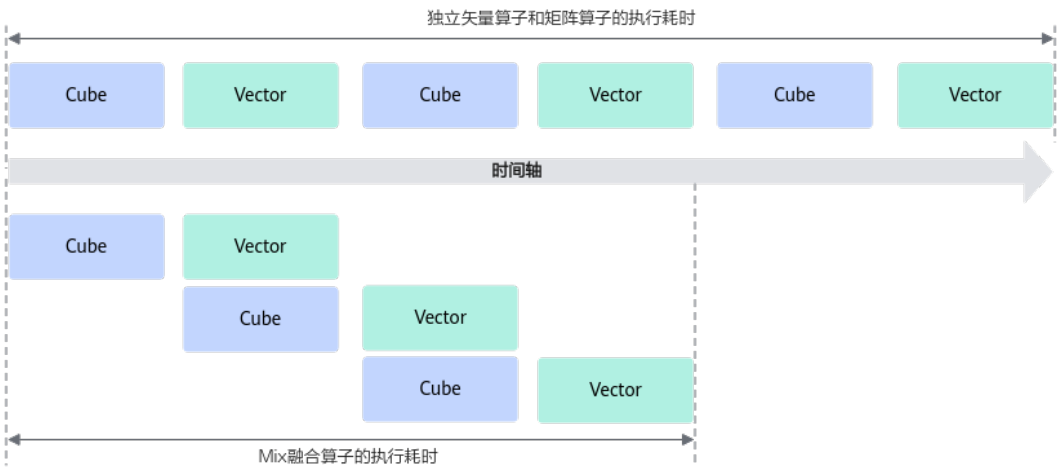
图 7-30 Flash Attention 核心实现



融合算子使用场景和优势

当对算子的性能要求较高时，可以通过融合算子编程的方式，将矢量算子和矩阵算子进行融合，通过一个算子kernel函数来承载，由此来获得性能上的收益。下图展示了独立矢量算子和矩阵算子、Mix融合算子的执行耗时对比，由此可以看出为什么开发Mix融合算子会带来性能上的收益。

图 7-31 独立矢量算子和矩阵算子、Mix 融合算子的执行耗时对比



- **独立的矢量算子和矩阵算子实现**：矩阵计算后的结果需要搬运到Global Memory上，然后由Global Memory搬运到LocalMemory，再进行矢量算子的计算，计算和搬运都是串行执行：另外多个算子的调度执行，会增加算子的调度耗时。

- **融合算子的实现方法**：可以对数据进行切片，再通过流水的设计，使得矢量计算单元和矩阵计算单元实现并行计算；另外相比于不融合的单算子，减少了算子的调度耗时。

除了有效提升算子性能，充分发挥AI处理器的算力，融合算子还有如下优势：

- 减少计算量：融合算子可以将多个算子合并为一个，简化计算过程，减少计算量，提高计算效率。
- 减少内存占用：融合算子可以将多个算子的中间结果合并为一个，从而减少内存占用，提高内存利用率。
- 优化数据流：融合算子可以优化数据流，减少数据在不同算子之间的传输，从而提高数据处理效率。
- 简化代码实现：融合算子可以简化代码实现，减少代码量，提高代码可读性和可维护性。

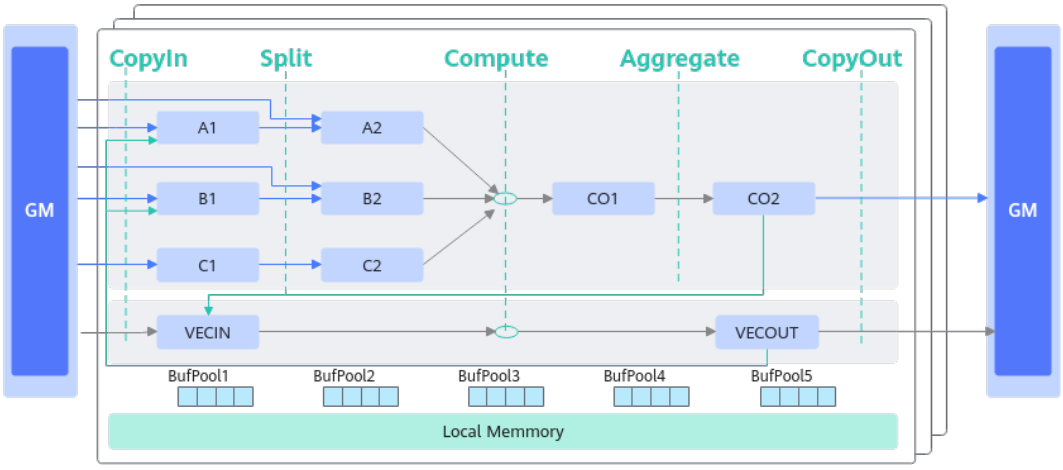
总之，融合算子是一种优化计算的有效手段，可以提高计算效率和内存利用率，优化数据流，简化代码实现。

编程范式

Ascend C提供**融合算子的编程范式**，方便开发者基于该范式表达融合算子的数据流，快速实现自己的融合算子。

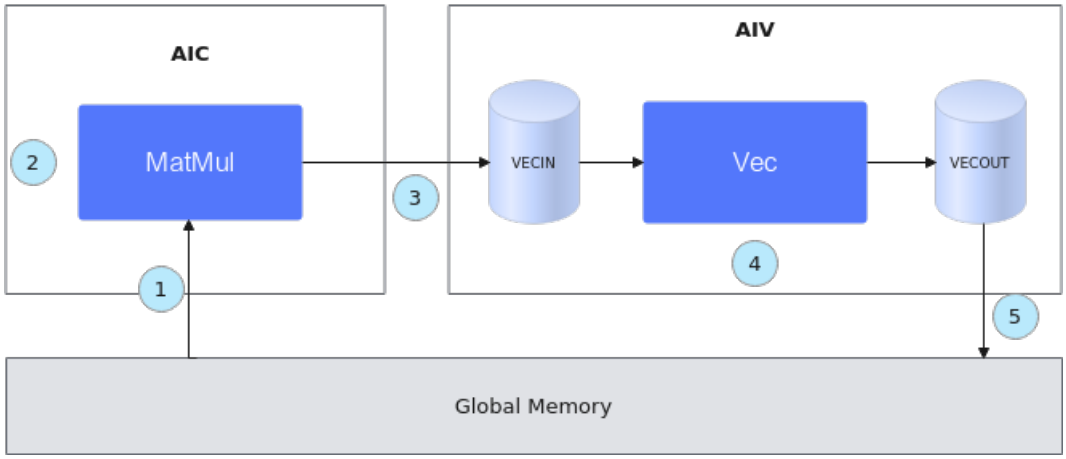
融合算子数据流指融合算子的输入输出在各存储位置间的流向。以一个典型的Cube和Vector融合算子为例，逻辑位置间的数据流向如下图所示：

- Cube的输出可以作为Vector的输入：CO2->VECIN
- Vector的输出可以作为Cube的输入：VECOUT->A1->A2、VECOUT->B1->B2



基于Matmul高阶API的融合算子编程范式，对上述数据流简化表达如下：

图 7-32 融合算子编程范式



1. 初始化一个MatMul对象，将输入数据从Global Memory搬运到Cube核上。
2. 进行MatMul内部的计算。
3. 将MatMul的计算结果搬运到Vector核上。
4. 进行Vector矢量计算。
5. 将输出结果搬运到Global Memory上。

整个过程的示例代码如下（伪代码）：

```
template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::Process()
{
    // 步骤1：初始化一个MatMul对象，将输入数据从Global Memory搬运到Cube核上。
    uint32_t computeRound = 0;
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), matmulObj);
    matmulObj.Init(&tiling);
    matmulObj.SetTensorA(aGlobal);
    matmulObj.SetTensorB(bGlobal);
    matmulObj.SetBias(biasGlobal);

    while (matmulObj.template Iterate<true>()) { // 步骤2：进行MatMul内部的计算。
        // 步骤3：将MatMul的计算结果搬运到Vector核上。
        reluOutLocal = reluOutQueue_.AllocTensor<cType>();
        matmulObj.template GetTensorC<true>(reluOutLocal, false, true);
       // 步骤4：进行Vector矢量计算。
        AscendC::LeakyRelu(reluOutLocal, reluOutLocal, (cType)alpha, tiling.baseM * tiling.baseN);
        reluOutQueue_.EnQue(reluOutLocal);
        // 步骤5：将输出结果搬运到Global Memory上
        reluOutQueue_.DeQue<cType>();
        ...
        AscendC::DataCopy(cGlobal[startOffset], reluOutLocal, copyParam);
        reluOutQueue_.FreeTensor(reluOutLocal);

        computeRound++;
    }
    matmulObj.End();
}
```

7.5.2 算子实现

下文将以Matmul+LeakyRelu融合算子的实现为例，介绍Mix融合算子的设计和实现流程。该样例仅支持在**Atlas A2训练系列产品/Atlas 800I A2推理产品**上运行。

算子的设计过程分为**算子分析、数据流分析、Tiling策略设计**三部分。

算子分析

算子分析是指明确算子的数学表达式、输入、输出，核函数的名称等信息。

步骤1 明确算子的数学表达式及计算逻辑。该算子的计算逻辑为，先进行一个矩阵乘操作，然后将矩阵乘的结果与一个alpha参数进行LeakyRelu操作。数学表达式如下：

```
c = LeakyRelu(a * b + bias, alpha);
```

步骤2 明确输入和输出。

- MatMul+Leakyrelu算子输入为a、b、bias、alpha，输出为c。
- 本样例中算子输入a、b支持的数据类型为half（float16），算子输入bias、alpha支持的数据类型为float32，算子输出c的数据类型为float32。
- 输入矩阵a的形状为[M，K]，输入矩阵b的形状为[K, N]，输出矩阵c的形状为[M，N]，输入bias的形状为[1, N]。
- 算子输入输出支持的数据格式为：ND。

步骤3 确定核函数名称和参数。

- 您可以自定义核函数名称，本样例中核函数命名为matmul_leakyrelu_custom。
- 根据对算子输入输出的分析，确定核函数的参数a，b，bias，alpha，c；a，b，bias,alpha为输入在Global Memory上的内存地址，c为输出在Global Memory上的内存地址。

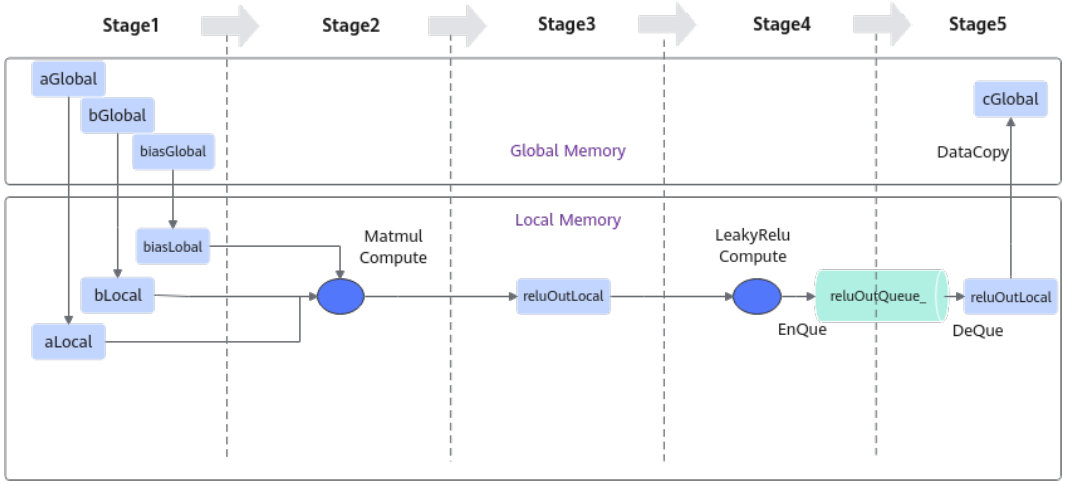
----结束

表 7-3 Ascend C MatMul+Leakyrelu 算子设计规格

算子类型（OpType）	MATMUL_LEAKYRELU			
算子输入	**name**	**shape**	**data type**	**format**
	a	[M, K]	half	ND
	b	[K, N]	half	ND
	bias	[1, N]	float32	-
	alpha	-	float32	-
算子输出	c	[M, N]	float32	ND
核函数名称	matmul_leakyrelu_custom			

数据流分析

进行算子的数据流分析：数据流向为在Cube核上完成Matmul计算后将数据搬运至Vector核进行LeakyRelu计算。根据上述数据流并结合融合算子的编程范式，规划并行的流水任务。如下图所示：

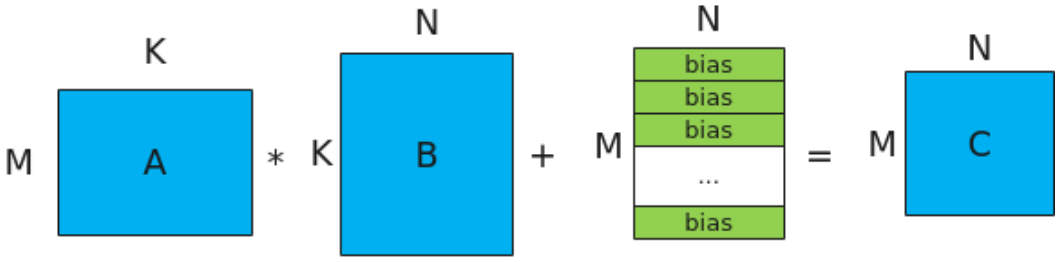


步骤1 将输入数据从Global Memory搬运到Cube核。

步骤2 进行MatMul内部的计算，计算公式和计算示意图如下：

注：bias的shape为[1, N], 对A*B结果矩阵的每一行都采用该bias进行偏置。

图 7-33 Matmul 矩阵乘示意图

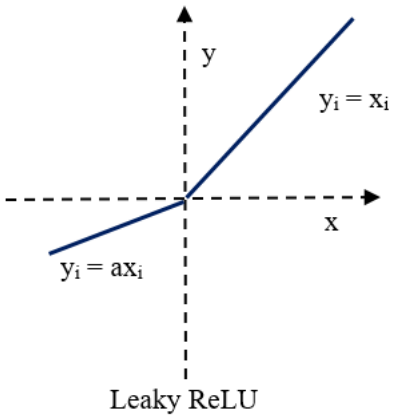


步骤3 将MatMul的计算结果搬运到Vector核。

步骤4 进行Vector矢量计算，该样例中进行LeakyReLU计算。

Leaky ReLU（带泄露线性整流函数）激活函数，是人工神经网络中一种常用的激活函数，其数学表达式和函数图像如下所示：

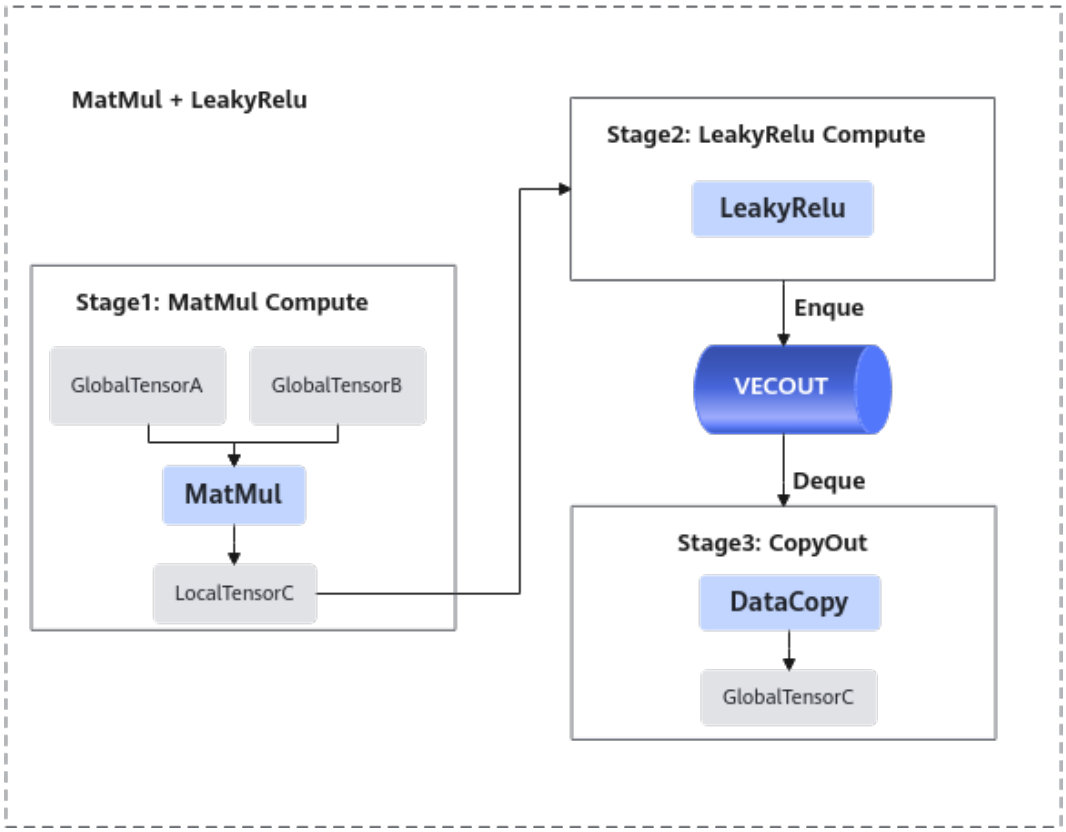
$$f(x_i)=\begin{cases} x_i & \text{if } x_i \geq 0 \\ ax_i & \text{if } x_i < 0 \end{cases}$$



步骤5 将输出结果搬运到Global Memory。

----结束

前三步的内容都封装在Matmul高阶API内，本样例中可以简化为3个stage。如下图所示：



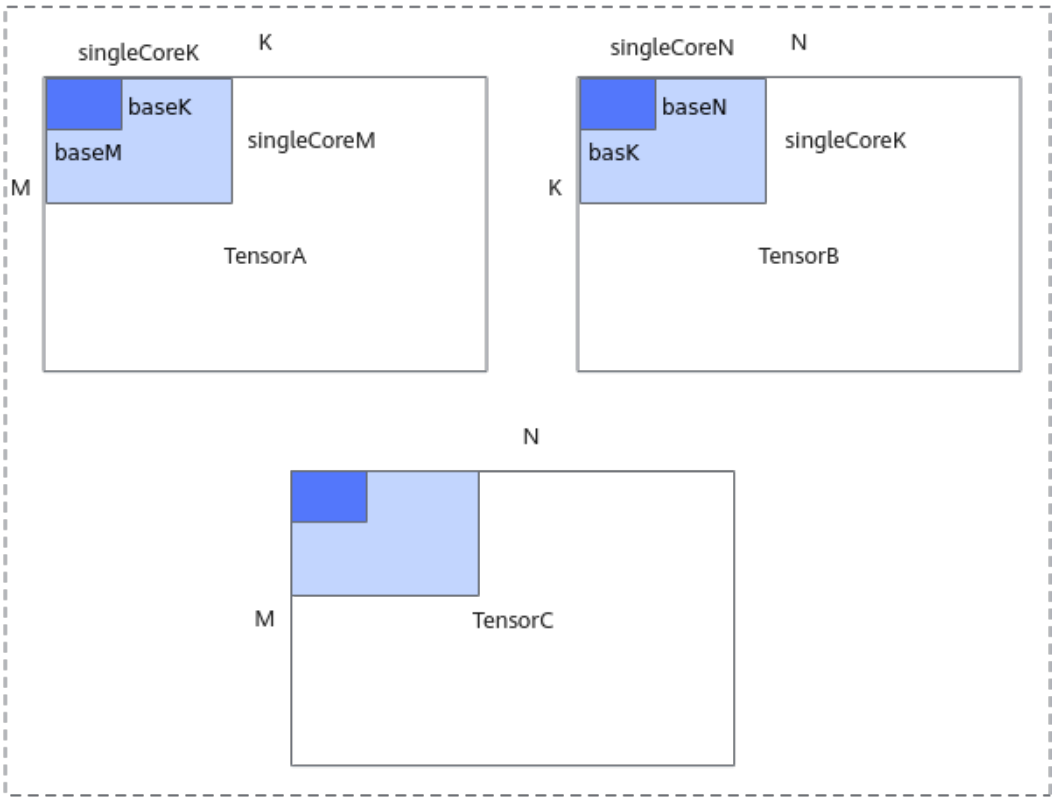
根据上述分析，明确实现过程中会使用到Matmul高阶API接口，LeakyRelu Vector计算接口、DataCopy、EnQue、DeQue接口。

Tiling 策略设计

Tiling策略的设计主要包括多核切分和核内切分策略。

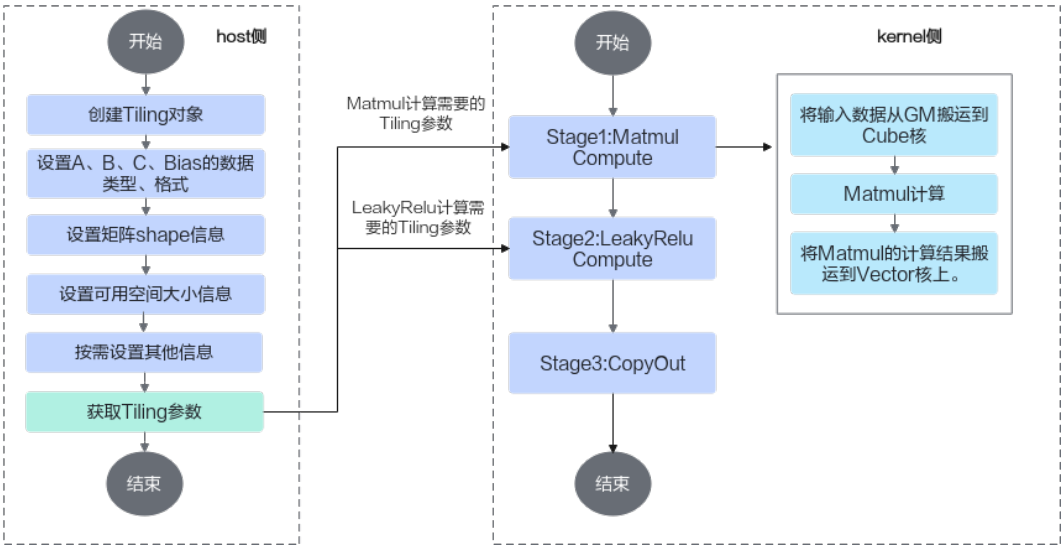
- 多核切分: 根据当前核数，对输入shapeM, K, N进行多核切分，得到单核内shape大小singleCoreM, singleCoreK, singleCoreN。
- 核内切分: 根据Local Memory的大小约束，对单核内的shape大小进一步切分，得到A、B、C矩阵参与一次矩阵乘指令的shape大小baseM, baseN, baseK 。切分时需要注意：GetTensorC的结果如果放在LocalMemory（UB）上，需要注意，baseM * baseN的大小不能超出UB的限制。

切分策略示意图如下，更多切分策略相关原理请参考**数据分块（Tiling）**。



算子实现

在**矩阵编程章节**，我们得知Ascend C提供一组Matmul高阶API，封装了常用的切分和数据搬运、计算的算法逻辑，方便用户快速实现Matmul矩阵乘法的运算操作。融合算子中的矩阵编程的部分实现与之类似，开发者在host侧通过调用API自动获取Tiling参数，该参数传递到kernel侧后，在初始化操作时传入，通过几个简单的API即可完成矩阵乘操作。再结合上文的融合算子的编程范式，融合算子实现的步骤如下。完整样例请参考**MatmulLeakyRelu**。



kernel侧实现的代码框架如下，在完成Matmul对象的初始化、左矩阵A、右矩阵B、Bias的设置后，通过单次Iterate叠加while循环的方式完成后续的Matmul计算、LeakyRelu计算、CopyOut流程。

```
template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::Process(){
    uint32_t computeRound = 0;
    // Matmul对象初始化
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), matmulObj);
    // 设置Matmul的输入（包括左矩阵、右矩阵、bias）
    matmulObj.Init(&tiling);
    matmulObj.SetTensorA(aGlobal);
    matmulObj.SetTensorB(bGlobal);
    matmulObj.SetBias(biasGlobal);
    // 调用matmul iterate获取一块[baseM, baseN]的计算结果
    while (matmulObj.template Iterate<true>())
    {
        MatmulCompute();
        LeakyReluCompute();
        CopyOut(computeRound);
        computeRound++;

    }
    matmulObj.End();
}
```

Matmul计算、LeakyRelu计算、CopyOut的具体实现代码如下：

步骤1 Matmul计算：

1. 将输入数据从Global Memory搬运到Cube核。
2. 进行MatMul内部的计算。
3. 将MatMul的计算结果搬运到Vector核。

```
template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::Process(){
    uint32_t computeRound = 0;
    // ...
    // 调用matmul iterate获取一块[baseM, baseN]的计算结果
    while (matmulObj.template Iterate<true>())
    {
        MatmulCompute();
        // ...
        computeRound++;

    }
    matmulObj.End();
}

template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::MatmulCompute(){
    reluOutLocal = reluOutQueue_.AllocTensor<cType>();
    // 调用GetTensorC将Matmul的计算结果搬运到Vector核。
    matmulObj.template GetTensorC<true>(reluOutLocal, false, true);
}
```

步骤2 LeakyRelu计算。

```
// 调用LeakyRule接口进行计算
template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::LeakyReluCompute(){
    AscendC::LeakyRelu(reluOutLocal, reluOutLocal, (cType)alpha, tiling.baseM * tiling.baseN);
    reluOutQueue_.EnQue(reluOutLocal);
}
```

步骤3 CopyOut，将输出结果搬运到Global Memory。

```
// 将结果搬出到GM
template<typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::CopyOut(uint32_t count){
```



```
    reluOutQueue_.DeQue<cType>();
    const uint32_t roundM = tiling.singleCoreM / tiling.baseM;
    const uint32_t roundN = tiling.singleCoreN / tiling.baseN;
    uint32_t startOffset = (count % roundM * tiling.baseM * tiling.N + count / roundM * tiling.baseN);
    AscendC::DataCopyParams copyParam = {(uint16_t)tiling.baseM,
        (uint16_t)(tiling.baseN * sizeof(cType) / DEFAULT_C0_SIZE), 0,
        (uint16_t)((tiling.N - tiling.baseN) * sizeof(cType) / DEFAULT_C0_SIZE)};
    AscendC::DataCopy(cGlobal[startOffset], reluOutLocal, copyParam);
    reluOutQueue_.FreeTensor(reluOutLocal);
}
```

----结束

host侧自动获取Tiling参数的关键步骤介绍如下：

步骤1 创建Tiling对象。

```
auto ascendcPlatform = platform_ascendc::PlatformAscendC(context->GetPlatformInfo());
matmul_tiling::MultiCoreMatmulTiling cubeTiling(ascendcPlatform);
```

创建对象时需要传入硬件平台信息，硬件平台信息可以通过GetPlatformInfo获取。

步骤2 设置A、B、Bias的数据类型和格式。

```
cubeTiling.SetAType(matmul_tiling::TPosition::GM, matmul_tiling::CubeFormat::ND,
matmul_tiling::DataType::DT_FLOAT16);
cubeTiling.SetBType(matmul_tiling::TPosition::GM, matmul_tiling::CubeFormat::ND,
matmul_tiling::DataType::DT_FLOAT16);
cubeTiling.SetCType(matmul_tiling::TPosition::LCM, matmul_tiling::CubeFormat::ND,
matmul_tiling::DataType::DT_FLOAT);
cubeTiling.SetBiasType(matmul_tiling::TPosition::GM, matmul_tiling::CubeFormat::ND,
matmul_tiling::DataType::DT_FLOAT);
```

步骤3 设置矩阵shape信息。

```
cubeTiling.SetShape(M, N, K);
cubeTiling.SetOrgShape(M, N, K);
```

步骤4 设置可用空间大小信息。

```
cubeTiling.SetBufferSpace(-1, -1, -1);
```

步骤5 按需设置其他参数，比如设置bias参与计算。

```
cubeTiling.SetBias(true);
```

步骤6 获取Tiling参数。

```
MatmulLeakyreluCustomTilingData tiling;
if (cubeTiling.GetTiling(tiling.cubeTilingData) == -1){
    return ge::GRAPH_FAILED;
}
```

步骤7 Tiling参数的序列化保存等其他操作。

----结束

说明

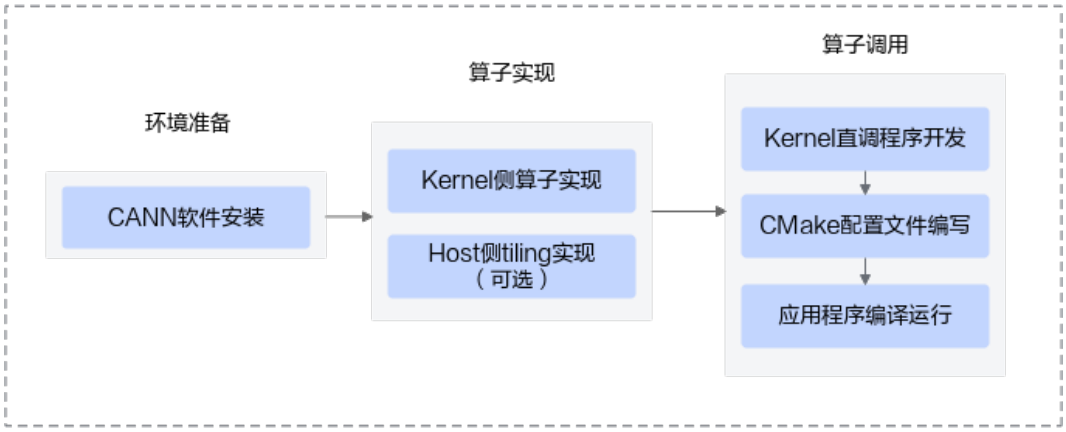
- 特别的对于多核场景，需要通过SetDim接口设置Matmul计算所用的核数，MIX模式（包含矩阵计算和矢量计算）的设置规则如下：
  - 分离架构：Matmul API都是从AIV侧发起的，调用Iterate计算时在AIV侧只会起到通知的作用，通知AIC去做矩阵计算，计算完成后AIC告知AIV计算完成，在开发者层面感知的是AIV的核数，比如：SetBlockDim时可以设置为20，启动20个AI Core（AIC AIV的组合），SetDim设置成40，表示按照40个AIV进行切分。
  - 耦合架构：SetBlockDim加载的核数就是Matmul API实际计算会用到的核数，SetDim和SetBlockDim设置的值是一样的。
- Matmul高阶API内部实现时需要使用系统workspace，开发者需要：
  - 在host侧Tiling实现时，设置总的workspace的数值大小（包含用户workspace和系统workspace），workspace空间由框架来申请并管理。系统workspace的空间大小通过GetLibApiWorkSpaceSize获取。
    ```
    size_t userWorkspaceSize = 0;
    size_t systemWorkspaceSize = ascendcPlatform.GetLibApiWorkSpaceSize();
    size_t *currentWorkspace = context->GetWorkspaceSizes(1);
    currentWorkspace[0] = userWorkspaceSize + systemWorkspaceSize;
    ```
  - 若算子工程不是**自定义算子工程**，也不是带有**-DHAVE_WORKSPACE**编译宏的Kernel直调算子工程，kernel侧需要在Matmul初始化前，通过SetSysWorkSpace设置系统workspace。
    ```
    // 使用Matmul时必须设置workspace空间
    SetSysWorkspace(workspace);
    if (GetSysWorkSpacePtr() == nullptr) {
        return;
    }
    ```


8 Kernel 直调算子开发

8.1 概述

开发者完成**kernel侧算子实现**和**host侧tiling实现**后，即可通过AscendCL运行时接口，完成算子的调用。该方式下tiling开发不受CANN框架的限制，简单直接，多用于算子功能的快速验证。

Kernel直调算子开发流程如下图所示：



步骤1 环境准备

CANN软件安装请参考**2 环境准备**。

步骤2 算子实现

Kernel侧算子实现和host侧tiling实现请参考**7 算子实现**。

步骤3 算子调用

完成kernel直调程序的开发、CMake配置文件的编写后，按照如下**kernel直调工程**（如下工程结构仅为示例）组织相关代码文件，最后完成应用程序编译及运行。具体内容请参考**8.2 Kernel直调**。

```
|-- cmake                    // CMake编译文件
|-- CMakeLists.txt           // CMake编译配置文件
|-- my_add.cpp               // Kernel侧算子实现
|-- main.cpp                 // Kernel直调程序
```

----结束

8.2 Kernel 直调

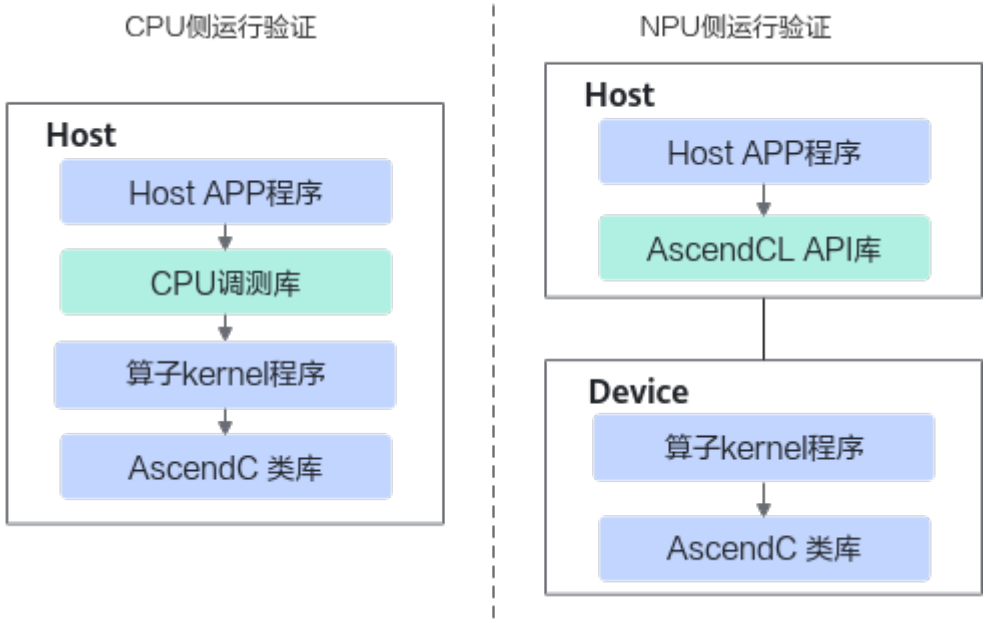
8.2.1 核函数运行验证简介

核函数即算子kernel程序开发完成后，即可编写host侧的核函数调用程序，实现从host侧的APP程序调用算子，进行运行验证。本节将会介绍CPU侧和NPU侧两种运行验证方法：

- **CPU侧运行验证**主要通过ICPU_RUN_KF CPU调测宏等CPU调测库提供的接口来完成；
- **NPU侧运行验证**主要通过使用Kernel Launch接口或者<<<>>>内核调用符，以及AscendCL API提供的运行时接口来完成。

CPU侧和NPU侧的运行验证原理图如下：

图 8-1 CPU 侧和 NPU 侧运行验证原理图



您可以根据下文**8.2.3 运行验证算子工程**的介绍来完成基本的运行验证流程，流程中使用到接口请参考：

- CPU调测库请参考调测接口。
- NPU调测的Kernel Launch接口请参考**8.2.2 NPU侧调用方式**，<<<>>>内核调用符使用方法请参考**核函数调用**。
- AscendCL API使用方法请参考AscendCL API参考。

基于NPU域算子调用编写的算子程序，通过毕昇编译器编译后运行，可以完成算子NPU域的运行验证；基于CPU域算子的调用接口（ICPU_RUN_KF CPU）编写的算子程序，通过标准的GCC编译器进行编译后运行，可以完成算子CPU域的运行验证。

CPU侧的运行程序，通过GDB通用调试工具进行单步调试，精准验证程序执行流程是否符合预期。如果您想进一步了解CPU侧调试的具体内容，可在完成本节内容的学习后参考**10.2 CPU域调试**。

8.2.2 NPU 侧调用方式

NPU侧的运行验证当前分为三种方式，使用Kernel Launch接口调用，使用<<<>>>内核调用符调用或者使用**8.2.4 Pybind调用**。开发者完成算子核函数的开发和Tiling实现后，即可通过任意一种方式以及AscendCL运行时接口，完成算子的调用并实现自己的推理应用；同时提供简易的kernel开发工程，开发者仅需提供kernel侧实现，基于工程框架可以快速实现任意一种方式的调用。

Kernel Launch 方式

须知

- 当前版本，Kernel Launch开放式编程为试用特性，后续版本会存在变更，不支持应用于商用产品中。
- 当前版本暂不支持获取用户workspace特性。

ACLRT_LAUNCH_KERNEL调用接口的使用方法如下：

```
ACLRT_LAUNCH_KERNEL(kernel_name)(blockDim, stream, argument list);
```

- kernel_name：算子核函数的名称。
- blockDim：规定了核函数将会在几个核上执行。每个执行该核函数的核会被分配一个逻辑ID，即block_idx，可以在核函数的实现中调用GetBlockIdx来获取block_idx。
- stream，类型为aclrtStream，stream用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序在Device上执行。stream创建等管理接口请参考Stream管理。
- argument list：参数列表，与核函数的参数列表保持一致。

考虑名为add_custom的核函数调用的例子，该函数实现两个矢量的相加，调用示例如下：

```
// blockDim设置为8表示在8个核上调用了add_custom核函数，每个核都会独立且并行地执行该核函数，该核函数的参数列表为x，y，z。
ACLRT_LAUNCH_KERNEL(add_custom)(8, stream, x, y, z)
```

内核调用符方式

核函数可以使用内核调用符<<<...>>>这种语法形式，来规定核函数的执行配置：

```
kernel_name<<<blockDim, l2ctrl, stream>>>(argument list);
```

- blockDim，规定了核函数将会在几个核上执行。每个执行该核函数的核会被分配一个逻辑ID，即block_idx，可以在核函数的实现中调用GetBlockIdx来获取block_idx；

说明

blockDim是逻辑核的概念，取值范围为[1,65535]。为了充分利用硬件资源，一般设置为物理核的核数或其倍数。对于耦合架构和分离架构，blockDim在运行时的意义和设置规则有一些区别，具体说明如下：

- 耦合架构：由于其Vector、Cube单元是集成在一起的，blockDim用于设置启动多个AICore核实例执行，不区分Vector、Cube。AI Core的核数可以通过GetCoreNumAiv或者GetCoreNumAic获取。
- 分离架构
  - 针对仅包含Vector计算的算子，blockDim用于设置启动多少个Vector（AIV）实例执行，比如某款AI处理器上有40个Vector核，建议设置为40。
  - 针对仅包含Cube计算的算子，blockDim用于设置启动多少个Cube（AIC）实例执行，比如某款AI处理器上有20个Cube核，建议设置为20。
  - 针对Vector/Cube融合计算的算子，启动时，按照AIV和AIC组合启动，blockDim用于设置启动多少个组合执行，比如某款AI处理器上有40个Vector核和20个Cube核，一个组合是2个Vector核和1个Cube核，建议设置为20，此时会启动20个组合，即40个Vector核和20个Cube核。**注意：该场景下，设置的blockDim逻辑核的核数不能超过物理核（2个Vector核和1个Cube核组合为1个物理核）的核数。**
  - AIC/AIV的核数分别通过GetCoreNumAic和GetCoreNumAiv接口获取。

- l2ctrl，保留参数，暂时设置为固定值nullptr，开发者无需关注；
- stream，类型为aclrtStream，stream用于维护一些异步操作的执行顺序，确保按照应用程序中的代码调用顺序在device上执行。stream创建等管理接口请参考Stream管理。

考虑名为add_custom的核函数调用的例子，该函数实现两个矢量的相加，调用示例如下：

```
// blockDim设置为8表示在8个核上调用了add_custom核函数，每个核都会独立且并行地执行该核函数，该核函数的参数列表为x，y，z。
add_custom<<<8, nullptr, stream>>>(x, y, z);
```

内核调用符进行核函数调用的详细示例可参见**核函数调用**。

核函数的调用是异步的，核函数的调用结束后，控制权立刻返回给主机端，可以调用以下aclrtSynchronizeStream函数来强制主机端程序等待所有核函数执行完毕。

```
aclError aclrtSynchronizeStream(aclrtStream stream);
```

aclrtSynchronizeStream的具体用法参考《**AscendCL应用软件开发指南（C&C++）**》“AscendCL API参考” - 同步等待 - aclrtSynchronizeStream章节。

8.2.3 运行验证算子工程

为了帮助开发者快速的完成算子的Kernel直调，方便开发者调试调优，提供简易的算子工程，您可以基于该算子工程中的样例代码和工程框架进行算子开发。算子工程提供的功能如下：

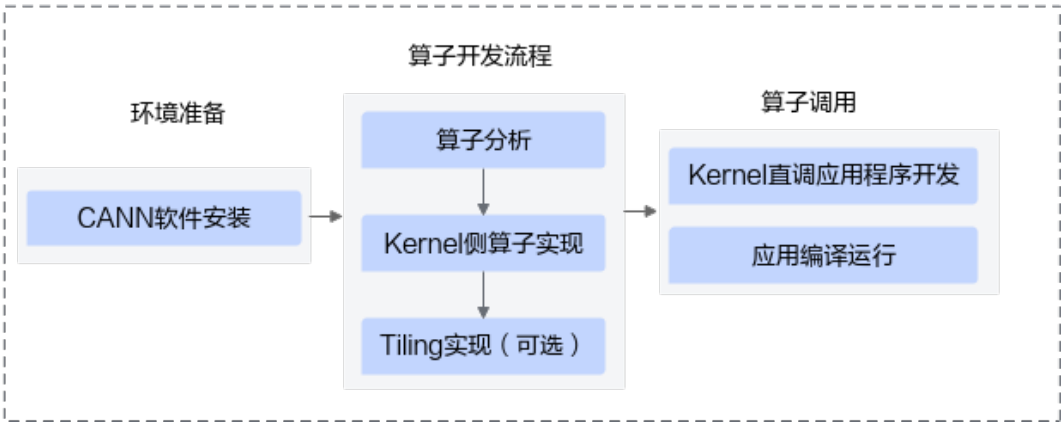
- 该工程支持prinf功能、DumpTensor功能。
- 通过该工程编译生成的应用程序。可以通过msProf命令行方式采集和解析性能数据。性能分析工具的具体使用方法请参考《**性能调优工具指南**》。

算子样例工程请通过如下链接获取：

- **矢量算子样例**
- **支持Tiling的矢量算子样例**

- **矩阵算子样例**
- **矢量+矩阵融合算子样例**

基于Kernel直调工程的算子开发流程图如下：



下文将以Add矢量算子为例对算子工程进行详细介绍。

环境准备

- 使用Kernel Launch算子工程之前，需要参考**2 环境准备**章节安装驱动固件和CANN软件包，完成开发环境和运行环境的准备。
- 使用该算子工程要求cmake版本为3.16及以上版本，如不符合要求，请参考如下的命令示例更新cmake版本，如下示例以更新到3.16.0版本为例。

```
wget https://cmake.org/files/v3.16/cmake-3.16.0.tar.gz --no-check-certificate
tar -zxvf cmake-3.16.0.tar.gz
cd cmake-3.16.0
 ./bootstrap --prefix=/usr
sudo make
sudo make install
```

工程目录

您可以单击**矢量算子样例**，获取核函数开发和运行验证的完整样例。样例目录结构如下所示：

```
AddKernelInvocationNeo
|-- cmake                                  // CMake编译文件
|-- scripts
|   ├── gen_data.py                        // 输入数据和真值数据生成脚本文件
|   ├── verify_result.py                   // 验证输出数据和真值数据是否一致的验证脚本
|-- CMakeLists.txt                         // CMake编译配置文件
|-- add_custom.cpp                         // 矢量算子kernel实现
|-- data_utils.h                           // 数据读入写出函数
|-- main.cpp                               // 主函数，调用算子的应用程序，含CPU域及NPU域调用
|-- run.sh                                 // 编译运行算子的脚本
```

基于该算子工程，开发者进行算子开发的步骤如下：

- 完成算子kernel侧实现。
- 编写算子调用应用程序main.cpp。
- 编写CMake编译配置文件CMakeLists.txt。
- 根据实际需要修改输入数据和真值数据生成脚本文件gen_data.py；验证输出数据和真值数据是否一致的验证脚本verify_result.py。

- 根据实际需要修改编译运行算子的脚本run.sh并执行该脚本，完成算子的编译运行和结果验证。

算子 kernel 侧实现

请参考7.2 矢量编程和工程目录中的矩阵算子、融合算子的kernel实现完成Ascend C算子实现文件的编写。

算子调用应用程序

下面代码以固定shape的add_custom算子为例，介绍算子核函数调用的应用程序main.cpp如何编写。您在实现自己的应用程序时，需要关注由于算子核函数不同带来的修改，包括算子核函数名，入参出参的不同等，合理安排相应的内存分配、内存拷贝和文件读写等，相关API的调用方式直接复用即可。

步骤1 按需包含头文件，通过ASCENDC_CPU_DEBUG宏区分CPU/NPU侧需要包含的头文件。需要注意的是，NPU侧需要包含对应的核函数调用接口声明所在的头文件alcrtlaunch_{kernel_name}.h（该头文件为工程框架自动生成），kernel_name为算子核函数的名称。

```
#include "data_utils.h"
#ifndef ASCENDC_CPU_DEBUG
#include "acl/acl.h"
#include "aclrtlaunch_add_custom.h"
#else
#include "tikicpulib.h"
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z);
#endif
```

步骤2 应用程序框架编写。该应用程序通过ASCENDC_CPU_DEBUG宏区分代码逻辑运行于CPU侧还是NPU侧。

```
int32_t main(int32_t argc, char* argv[])
{
    uint32_t blockDim = 8;
    size_t inputByteSize = 8 * 2048 * sizeof(uint16_t);
    size_t outputByteSize = 8 * 2048 * sizeof(uint16_t);

#ifdef ASCENDC_CPU_DEBUG
    // 用于CPU调试的调用程序
#else
    // NPU侧运行算子的调用程序
#endif
    return 0;
}
```

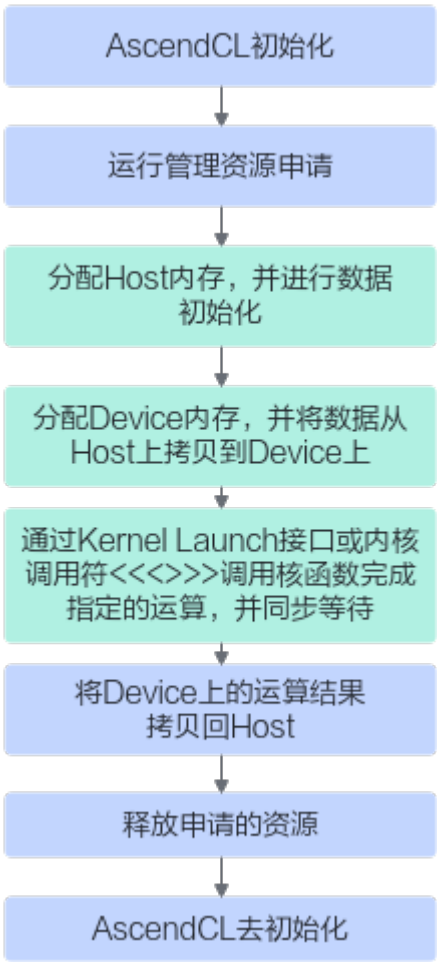
步骤3 CPU侧运行验证。完成算子核函数CPU侧运行验证的步骤如下：

图 8-2 CPU 侧运行验证步骤


```
// 使用GmAlloc分配共享内存，并进行数据初始化
uint8_t* x = (uint8_t*)AscendC::GmAlloc(inputByteSize);
uint8_t* y = (uint8_t*)AscendC::GmAlloc(inputByteSize);
uint8_t* z = (uint8_t*)AscendC::GmAlloc(outputByteSize);

ReadFile("./input/input_x.bin", inputByteSize, x, inputByteSize);
ReadFile("./input/input_y.bin", inputByteSize, y, inputByteSize);
// 矢量算子需要设置内核模式为AIV模式
AscendC::SetKernelMode(KernelMode::AIV_MODE);
// 调用ICPU_RUN_KF调测宏，完成核函数CPU侧的调用
ICPU_RUN_KF(add_custom, blockDim, x, y, z);
// 输出数据写出
WriteFile("./output/output_z.bin", z, outputByteSize);
// 调用GmFree释放申请的资源
AscendC::GmFree((void *)x);
AscendC::GmFree((void *)y);
AscendC::GmFree((void *)z);
```

步骤4 NPU侧运行验证。完成算子核函数NPU侧运行验证的步骤如下：

图 8-3 NPU 侧运行验证步骤




```
    // AscendCL初始化
    CHECK_ACL(aclInit(nullptr));
    // 运行管理资源申请
    int32_t deviceId = 0;
    CHECK_ACL(aclrtSetDevice(deviceId));
    aclrtStream stream = nullptr;
    CHECK_ACL(aclrtCreateStream(&stream));
    // 分配Host内存
    uint8_t *xHost, *yHost, *zHost;
    uint8_t *xDevice, *yDevice, *zDevice;

    CHECK_ACL(aclrtMallocHost((void**)(&xHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&yHost), inputByteSize));
    CHECK_ACL(aclrtMallocHost((void**)(&zHost), outputByteSize));
    // 分配Device内存
    CHECK_ACL(aclrtMalloc((void**)&xDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&yDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    CHECK_ACL(aclrtMalloc((void**)&zDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST));
    // Host内存初始化
    ReadFile("./input/input_x.bin", inputByteSize, xHost, inputByteSize);
    ReadFile("./input/input_y.bin", inputByteSize, yHost, inputByteSize);
    // 将数据从Host上拷贝到Device上
    CHECK_ACL(aclrtMemcpy(xDevice, inputByteSize, xHost, inputByteSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    CHECK_ACL(aclrtMemcpy(yDevice, inputByteSize, yHost, inputByteSize,
ACL_MEMCPY_HOST_TO_DEVICE));
    // 用ACLRT_LAUNCH_KERNEL接口调用核函数完成指定的运算
    ACLRT_LAUNCH_KERNEL(add_custom)(blockDim, stream, xDevice, yDevice, zDevice);
```

```
    // 用内核调用符<<<>>>调用核函数完成指定的运算,add_custom_do中封装了<<<>>>调用
    // add_custom_do(blockDim, nullptr, stream, xDevice, yDevice, zDevice);
    CHECK_ACL(aclrtSynchronizeStream(stream));
    // 将Device上的运算结果拷贝回Host
    CHECK_ACL(aclrtMemcpy(zHost, outputByteSize, zDevice, outputByteSize,
ACL_MEMCPY_DEVICE_TO_HOST));
    WriteFile("./output/output_z.bin", zHost, outputByteSize);
    // 释放申请的资源
    CHECK_ACL(aclrtFree(xDevice));
    CHECK_ACL(aclrtFree(yDevice));
    CHECK_ACL(aclrtFree(zDevice));
    CHECK_ACL(aclrtFreeHost(xHost));
    CHECK_ACL(aclrtFreeHost(yHost));
    CHECK_ACL(aclrtFreeHost(zHost));
    // AscendCL去初始化
    CHECK_ACL(aclrtDestroyStream(stream));
    CHECK_ACL(aclrtResetDevice(deviceId));
    CHECK_ACL(aclFinalize());
```

----结束

CMake 编译配置文件编写

本节会介绍CMake文件中一些关键环境变量和Cmake命令参数的说明，通常情况下不需要开发者修改，但是这些参数可以帮助开发者更好的理解编译原理，方便有能力的开发者对Cmake进行定制化处理。

表 8-1 环境变量说明

环境变量	配置说明
SOC_VERSION	AI处理器的型号。 ● 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。
ASCEND_CANN_PACKAGE_PATH	CANN软件包安装后的实际路径。
CMAKE_BUILD_TYPE	编译模式选项，可配置为： ● “Release”，Release版本，不包含调试信息，编译最终发布的版本。 ● “Debug”，Debug版本，包含调试信息，便于开发者开发和调试。
CMAKE_INSTALL_PREFIX	用于指定CMAKE执行install时，安装的路径前缀，执行install后编译产物（ascendc_library中指定的target以及对应的头文件）会安装在该路径下。默认路径为当前目录的out目录下。

CMAKE_CXX_COMPILER_LAUNCHER	用于配置C++语言编译器（如g++）、毕昇编译器的启动器程序为ccache，配置后即可开启cache缓存编译，加速重复编译并提高构建效率。用法如下，在对应的CMakeLists.txt进行设置：
	`set(CMAKE_CXX_COMPILER_LAUNCHER <launcher_program>)`
	其中<launcher_program>是ccache的安装路径，比如ccache的安装路径为/usr/bin/ccache，示例如下：
	`set(CMAKE_CXX_COMPILER_LAUNCHER /usr/bin/ccache)`

表 8-2 Cmake 命令语法说明

Cmake命令	语法说明
add_executable	使用指定的源文件将可执行文件添加到项目中。和Cmake通用的命令参数使用方法一致。

ascendc_library	使用指定的核函数源文件向项目（ project ）添加库。语法格式如下：
	`ascendc_library(<target_name> [STATIC | SHARED]`
	`[<source>...])`
	其中<target_name>表示库文件的名字，该库文件会根据命令里列出的源文件来建立。STATIC、SHARED的作用是指定生成的库文件的类型。STATIC库是目标文件的归档文件，在连接其它目标的时候使用。SHARED库会被动态连接（动态连接库），在运行时会被加载。<source>表示核函数源文件。

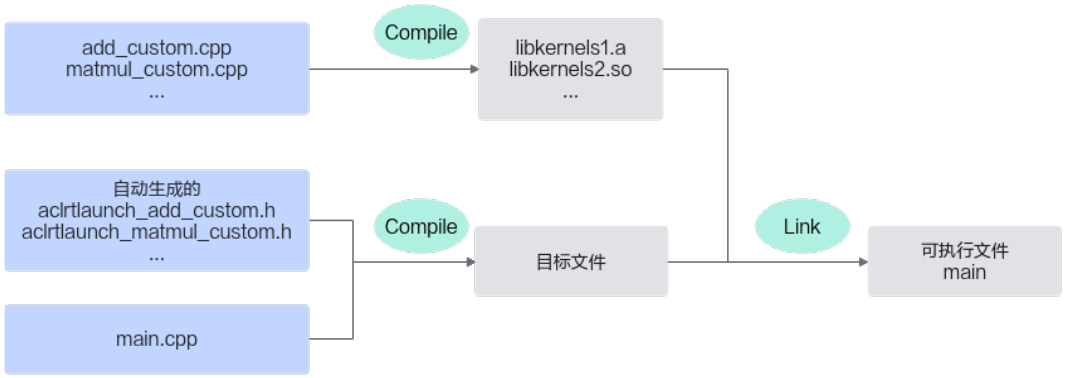
Cmake命令	语法说明

ascendc_compile_definitions	添加编译宏。可以添加Ascend C提供的编译宏和开发者自定义的编译宏。语法格式如下：
	`ascendc_compile_definitions(<target_name> [PRIVATE]`
	`  [<xxx>...])`
	Ascend C提供的编译宏介绍如下：
	● -DASCENDC_DUMP用于控制Dump开关，默认开关打开，开发者调用printf/DumpTensor/assert后会有信息打印(需要注意直调工程的kernel文件内存在host函数，如果在host函数内调用了printf接口，也会触发kernel内的printf相关初始化动作，进而影响kernel的执行性能)；设置为0后，表示开关关闭。示例如下：
	`// 关闭所有算子的printf打印功能`
	`ascendc_compile_definitions(ascendc_kernels_${RUN_MODE} PRIVATE`
	`  -DASCENDC_DUMP=0`
	`)`
	● -DASCENDC_DEBUG用于控制Ascend C API的调测开关，默认开关关闭；增加该编译宏后，表示开关打开，此时接口内部的assert校验生效，校验不通过会有assert日志打屏。开启该功能会对算子实际运行的性能带来一定影响，通常在调测阶段使用。示例如下：
	`ascendc_compile_definitions(ascendc_kernels_${RUN_MODE} PRIVATE`
	`  -DASCENDC_DEBUG`
	`)`
	当前-DASCENDC_DEBUG功能支持的产品型号为：
	Atlas 推理系列产品
	Atlas A2训练系列产品/Atlas 800I A2推理产品
	● -DHAVE_WORKSPACE用于表示kernel入口是否包含workspace入参。默认情况下为不包含；增加该编译宏后，表示包含，此时框架会获取kernel入参的倒数第一个参数（未配置**-DHAVE_TILING**），或倒数第二个参数（配置-DHAVE_TILING），自动在kernel侧设置系统workspace，开发者在kernel侧入参处获取的workspace为偏移了系统workspace后的用户workspace。当开发者使用了Matmul等需要系统workspace的高阶API时，建议开启此参数，入参排布、系统workspace的设置逻辑与**9 工程化算子开发**保持一致，可减少算子实现在不同开发方式间切换带来的修改成本。需要注意的是，host侧开发者仍需要自行申请workspace的空间，系统workspce大小可以通过PlatformAscendCManager的GetLibApiWorkSpaceSize接口获取。-DHAVE_WORKSPACE的设置样例如下：
	`ascendc_compile_definitions(ascendc_kernels_${RUN_MODE} PRIVATE`
	`  -DHAVE_WORKSPACE`
	`)`
	● -DHAVE_TILING用于表示kernel入口是否含有tiling入参。在配置了-DHAVE_WORKSPACE之后，此编译宏才会生效。默认情况下为不包含，开关关闭；增加该编译宏后，表示包含，此时框架会将kernel入参的最后一个参数当做tiling，将倒数第二个参数当做workspace。框架不会对此tiling入参做任何处理，仅通过该入参来判断workspace参数的位置，使用此编译宏可以和**9 工程化算子开发**保持入参一致，

Cmake命令	语法说明
	减少算子实现在不同开发方式间切换带来的修改成本。设置样例如下： ascendc_compile_definitions(ascendc_kernels_${RUN_MODE} PRIVATE -DHAVE_WORKSPACE -DHAVE_TILING)
ascendc_compile_options	添加编译选项。可以添加相应的编译选项用于host侧与device侧的编译过程。语法格式如下： ascendc_compile_options(ascendc_kernels_${RUN_MODE} PRIVATE [<xxx>...]) 默认情况下，指定的编译选项都将传递给device侧编译器进行编译。若想传递编译选项给host侧编译器，则需要使用“-forward-options-to-host-compiler”编译选项，该选项后的编译选项将传递给host侧编译器，示例如下： ascendc_compile_options(ascendc_kernels_${RUN_MODE} PRIVATE -g -forward-options-to-host-compiler -gdwarf-4) 如上代码所示，在编译时，“-g”编译选项传递给device侧编译器，“-gdwarf-4”编译选项传递给host侧编译器。 备注：host侧编译选项只支持g++与clang编译器共同支持的编译选项。
ascendc_include_directories	添加开发者自定义的头文件搜索路径。语法格式如下： ascendc_include_directories(<target_name> [PRIVATE] [<xxx>...])

简化的编译流程图如下图所示：将算子核函数源文件编译生成kernel侧的库文件（*.so或*.a库文件）；工程框架自动生成核函数调用接口声明头文件；编译main.cpp（算子调用应用程序）时依赖上述头文件，将编译应用程序生成的目标文件和kernel侧的库文件进行链接，生成最终的可执行文件。

图 8-4 编译简化流程图



编译安装结束后在CMAKE_INSTALL_PREFIX目录下生成的编译产物示例如下；最终的可执行文件会生成在cmake命令的执行目录下。

```
out
├── lib
│   ├── libkernels1.a
│   ├── libkernels2.so
├── include
│   ├── kernels1
│       ├── aclrtlaunch_matmul_custom.h
│       ├── aclrtlaunch_add_custom.h
│   ├── kernels2
│       ├── aclrtlaunch_xxx.h
│       ├── ...
```

对于lib目录下生成的库文件可通过msobjdump工具进一步解析得到kernel信息，具体操作参见**14.5 msobjdump工具**。

输入数据和真值数据生成以及验证脚本文件

以固定shape的add_custom算子为例，输入数据和真值数据生成的脚本样例如下：根据算子的输入输出编写脚本，生成输入数据和真值数据。

```
#!/usr/bin/python3
# -*- coding:utf-8 -*-
# Copyright 2022-2023 Huawei Technologies Co., Ltd
import numpy as np

def gen_golden_data_simple():
    input_x = np.random.uniform(1, 100, [8, 2048]).astype(np.float16)
    input_y = np.random.uniform(1, 100, [8, 2048]).astype(np.float16)
    golden = (input_x + input_y).astype(np.float16)

    input_x.tofile("./input/input_x.bin")
    input_y.tofile("./input/input_y.bin")
    golden.tofile("./output/golden.bin")

if __name__ == "__main__":
    gen_golden_data_simple()
```

验证输出数据和真值数据是否一致的验证脚本样例如下：当前使用numpy接口计算了输出数据和真值数据的绝对误差和相对误差，误差在容忍偏差范围内，视为精度符合要求，输出"test pass"字样。

```
import os
import sys
import numpy as np

loss = 1e-3 # 容忍偏差，一般fp16要求绝对误差和相对误差均不超过千分之一
minimum = 10e-10

def verify_result(real_result, golden):
    real_result = np.fromfile(real_result, dtype=np.float16) # 从bin文件读取实际运算结果
    golden = np.fromfile(golden, dtype=np.float16) # 从bin文件读取预期运算结果
    result = np.abs(real_result - golden) # 计算运算结果和预期结果偏差
    deno = np.maximum(np.abs(real_result), np.abs(golden)) # 获取最大值并组成新数组
    result_atol = np.less_equal(result, loss) # 计算绝对误差
    result_rtol = np.less_equal(result / np.add(deno, minimum), loss) # 计算相对误差
    if not result_rtol.all() and not result_atol.all():
        if np.sum(result_rtol == False) > real_result.size * loss and np.sum(result_atol == False) >
real_result.size * loss:
            print("[ERROR] result error")
            return False
    print("test pass")
    return True
```

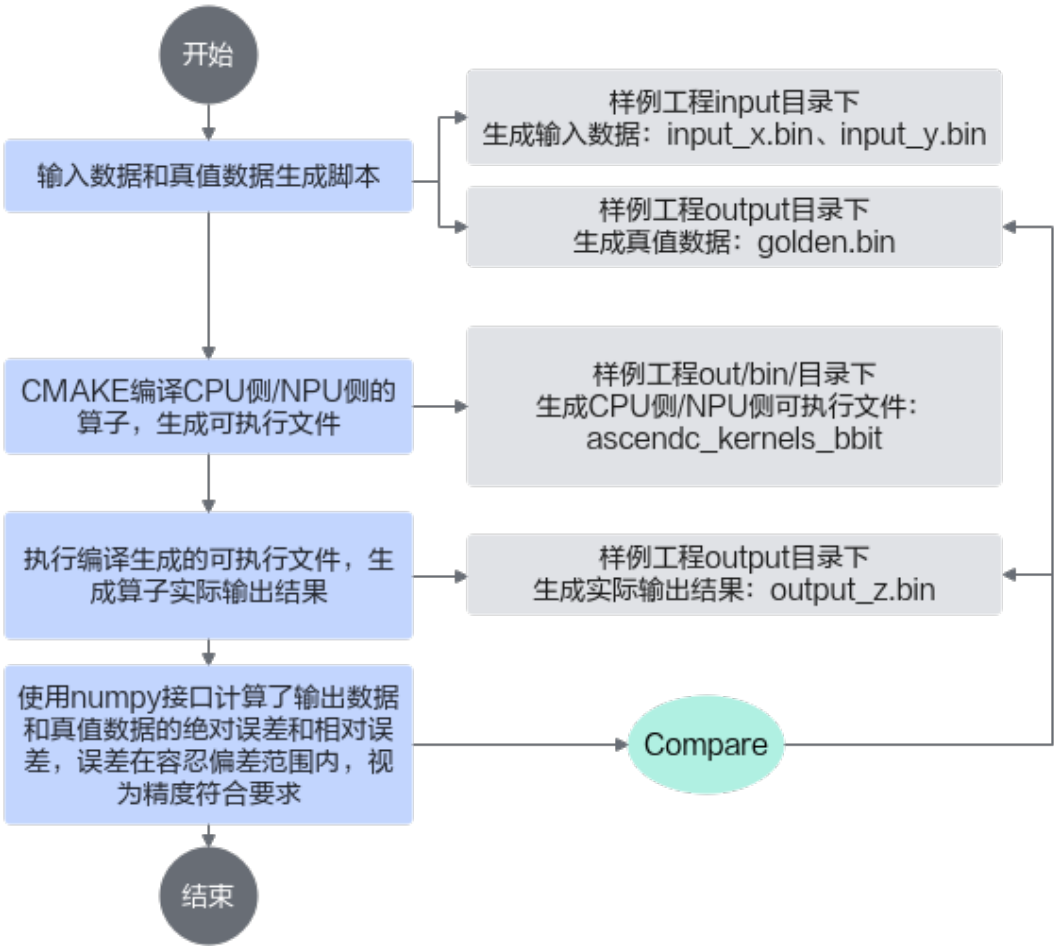


```
if __name__ == '__main__':
    verify_result(sys.argv[1],sys.argv[2])
```

修改并执行一键式编译运行脚本

您可以基于样例工程中提供的一键式编译运行脚本进行快速编译，并在CPU侧和NPU侧执行Ascend C算子。一键式编译运行脚本主要完成以下功能：

图 8-5 一键式编译运行脚本流程图



须知

样例中提供的一键式编译运行脚本并不能适用于所有的算子运行验证场景，使用时请根据实际情况进行修改。

- 根据Ascend C算子的算法原理的不同，自行实现输入和真值数据的生成脚本。

完成上述文件的编写后，可以执行一键式编译运行脚本，编译和运行应用程序。

脚本执行方式和脚本参数介绍如下：

```
bash run.sh --run-mode=npu  --soc-version=<soc_version> --install-path=<install_path>  --build-type=Debug  --install-prefix=<install-prefix>

bash run.sh -r npu  -v <soc_version> -i <install_path> -b Debug -p <install-prefix>
```


表 8-3 脚本参数介绍

参数名	参数简写	参数介绍
--run-mode	-r	表明算子以cpu模式或npu模式运行。 取值为cpu或npu。默认值为npu。
--soc-version	-v	算子运行的AI处理器型号。 **说明** AI处理器的型号<*soc_version*>请通过如下方式获取： • 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。 该样例支持以下型号： • Atlas 推理系列产品 • Atlas 训练系列产品 • Atlas A2训练系列产品/Atlas 800I A2推理产品
--install-path	-i	配置为CANN软件的安装路径，请根据实际安装路径进行修改。 默认值为$HOME/Ascend/ascend-toolkit/latest。
--build-type	-b	编译模式选项，可配置为： • Release，Release版本，不包含调试信息，编译最终发布的版本。 • Debug，Debug版本，包含调试信息，便于开发者开发和调试。 默认值为Debug。
--install-prefix	-p	用于指定CMAKE执行install时，安装的路径前缀，执行install后编译产物（ascendc_library中指定的target以及对应的头文件）会安装在该路径下。默认路径为当前目录的out目录下。

如下图所示，脚本执行完毕输出"test pass"字样表示算子精度符合要求。

8.2.4 Pybind 调用

Pybind 调用介绍

通过PyTorch框架进行模型的训练、推理时，会调用很多算子进行计算，其中的调用方式与kernel编译流程有关。对于自定义算子工程，需要使用PyTorch Ascend Adapter中的OP-Plugin算子插件对功能进行扩展，让torch可以直接调用自定义算子包中的算子，详细内容可以参考**12.2 PyTorch框架**；对于KernelLaunch开放式算子编程的方式，通过适配Pybind调用，可以实现PyTorch框架调用算子kernel程序。

Pybind是一个用于将C++代码与Python解释器集成的库，实现原理是通过将C++代码编译成动态链接库（DLL）或共享对象（SO）文件，使用Pybind提供的API将算子核函数与Python解释器进行绑定。在Python解释器中使用绑定的C++函数、类和变量，从而

实现Python与C++代码的交互。在Kernel直调中使用时，就是将Pybind模块与算子核函数进行绑定，将其封装成Python模块，从而实现两者交互。

Pybind调用方式中，使用的主要接口有：

- c10_npu::getCurrentNPUStream：获取当前npu流，返回值类型NPUStream，使用方式请参考**LINK**。
- ACLRT_LAUNCH_KERNEL：同**Kernel Launch方式**中ACLRT_LAUNCH_KERNEL接口。

算子样例工程请通过如下链接获取：

- **矢量算子样例**
- **矢量+矩阵融合算子样例**

环境准备

基于**环境准备**，还需要安装以下依赖：

- 安装pytorch (这里以2.1.0版本为例)
```
// aarch64环境上安装
pip3 install torch==2.1.0

// x86环境上安装
pip3 install torch==2.1.0+cpu  --index-url https://download.pytorch.org/whl/cpu
```
- 安装torch-npu （以Pytorch2.1.0、python3.9、CANN版本8.0.RC1.alpha002为例）
```
git clone https://gitee.com/ascend/pytorch.git -b v6.0.rc1.alpha002-pytorch2.1.0
cd pytorch/
bash ci/build.sh --python=3.9
pip3 install dist/*.whl
```
- 安装pybind11
```
pip3 install pybind11
```

工程目录

您可以单击**矢量算子样例**，获取核函数开发和运行验证的完整样例。样例目录结构如下所示：

```
├── CppExtensions
│   ├── add_custom_test.py      // Python调用脚本
│   ├── add_custom.cpp          // 算子实现
│   ├── CMakeLists.txt          // 编译工程文件
│   ├── pybind11.cpp            // pybind11函数封装
│   └── run.sh                  // 编译运行算子的脚本
```

基于该算子工程，开发者进行算子开发的步骤如下：

- 完成算子kernel侧实现。
- 编写算子调用应用程序和定义pybind模块pybind11.cpp。
- 编写Python调用脚本add_custom_test.py，包括生成输入数据和真值数据，调用封装的模块以及验证结果。
- 编写CMake编译配置文件CMakeLists.txt。
- 根据实际需要修改编译运行算子的脚本run.sh并执行该脚本，完成算子的编译运行和结果验证。

算子 kernel 侧实现

请参考7.2 矢量编程和工程目录中的算子kernel实现完成Ascend C算子实现文件的编写。

算子调用应用程序和定义 pybind 模块

下面代码以add_custom算子为例，介绍算子核函数调用的应用程序pybind11.cpp如何编写。您在实现自己的应用程序时，需要关注由于算子核函数不同带来的修改，包括算子核函数名，入参出参的不同等，相关API的调用方式直接复用即可。

步骤1 按需包含头文件。需要注意的是，需要包含对应的核函数调用接口声明所在的头文件alcrtlaunch_{kernel_name}.h（该头文件为工程框架自动生成），kernel_name为算子核函数的名称。

```
#include <pybind11/pybind11.h>
#include <torch/extension.h>

#include "aclrtlaunch_add_custom.h"
#include "torch_npu/csrc/core/npu/NPUStream.h"
```

步骤2 应用程序框架编写。需要注意的是，本样例的输入x，y的内存是在Python调用脚本add_custom_test.py中分配的。

```
namespace my_add {
at::Tensor run_add_custom(const at::Tensor &x, const at::Tensor &y) {
}
}
```

步骤3 NPU侧运行验证。使用ACLRT_LAUNCH_KERNEL接口调用算子核函数完成指定的运算。

```
// 运行资源申请，通过c10_npu::getCurrentNPUStream()的函数获取当前NPU上的流
auto acl_stream = c10_npu::getCurrentNPUStream().stream(false);
// 分配Device侧输出内存
at::Tensor z = at::empty_like(x);
uint32_t blockDim = 8;
uint32_t totalLength = 1;
for (uint32_t size : x.sizes()) {
  totalLength *= size;
}
// 用ACLRT_LAUNCH_KERNEL接口调用核函数完成指定的运算
ACLRT_LAUNCH_KERNEL(add_custom)(blockDim, acl_stream,
                                const_cast<void *>(x.storage().data()),
                                const_cast<void *>(y.storage().data()),
                                const_cast<void *>(z.storage().data()),
                                totalLength);
// 将Device上的运算结果拷贝回Host并释放申请的资源
return z;
```

步骤4 定义Pybind模块，将C++函数封装成Python函数。PYBIND11_MODULE是Pybind11库中的一个宏，用于定义一个Python模块。它接受两个参数，第一个参数是封装后的模块名，第二个参数是一个Pybind11模块对象，用于定义模块中的函数、类、常量等。通过调用m.def()方法，可以将步骤2中函数my_add::run_add_custom()转成Python函数run_add_custom，使其可以在Python代码中被调用。

```
PYBIND11_MODULE(add_custom, m) { // 模块名add_custom，模块对象m
  m.doc() = "add_custom pybind11 interfaces";  // optional module docstring
  m.def("run_add_custom", &my_add::run_add_custom, ""); // 将函数run_add_custom与Pybind模块进行绑定
}
```

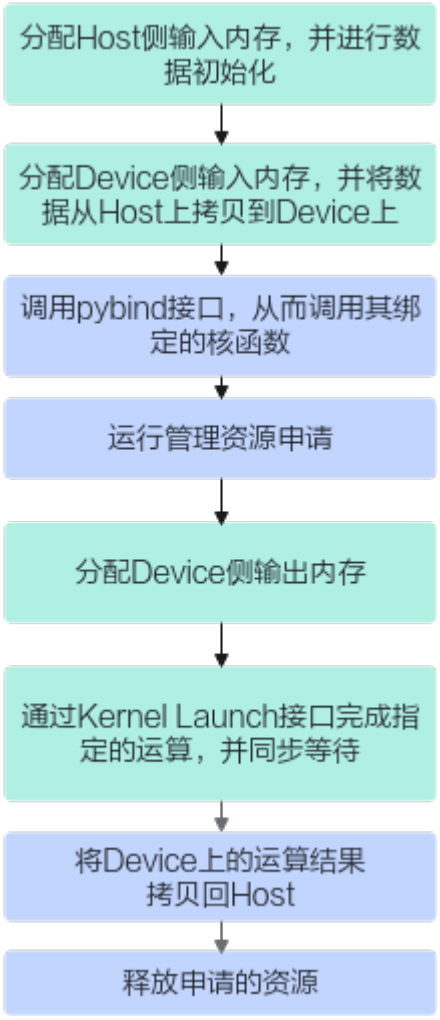
----结束

Python 调用脚本

在Python调用脚本中，使用torch接口生成随机输入数据并分配内存，通过导入封装的自定义模块add_custom，调用自定义模块add_custom中的run_add_custom函数，从

而在NPU上执行算子。算子核函数NPU侧运行验证的步骤如**图1 NPU侧运行验证原理图**。

图 8-6 NPU 侧运行验证原理图




```
import torch
import torch_npu
from torch_npu.testing.testcase import TestCase, run_tests
import sys, os
sys.path.append(os.getcwd())
import add_custom
torch.npu.config.allow_internal_format = False
class TestCustomAdd(TestCase):
    def test_add_custom_ops(self):
        // 分配Host侧输入内存，并进行数据初始化
        length = [8, 2048]
        x = torch.rand(length, device='cpu', dtype=torch.float16)
        y = torch.rand(length, device='cpu', dtype=torch.float16)
        // 分配Device侧输入内存，并将数据从Host上拷贝到Device上
        x_npu = x.npu()
        y_npu = y.npu()
        output = add_custom.run_add_custom(x_npu, y_npu)
        cpuout = torch.add(x, y)
        self.assertRtolEqual(output, cpuout)
if __name__ == "__main__":
    run_tests()
```

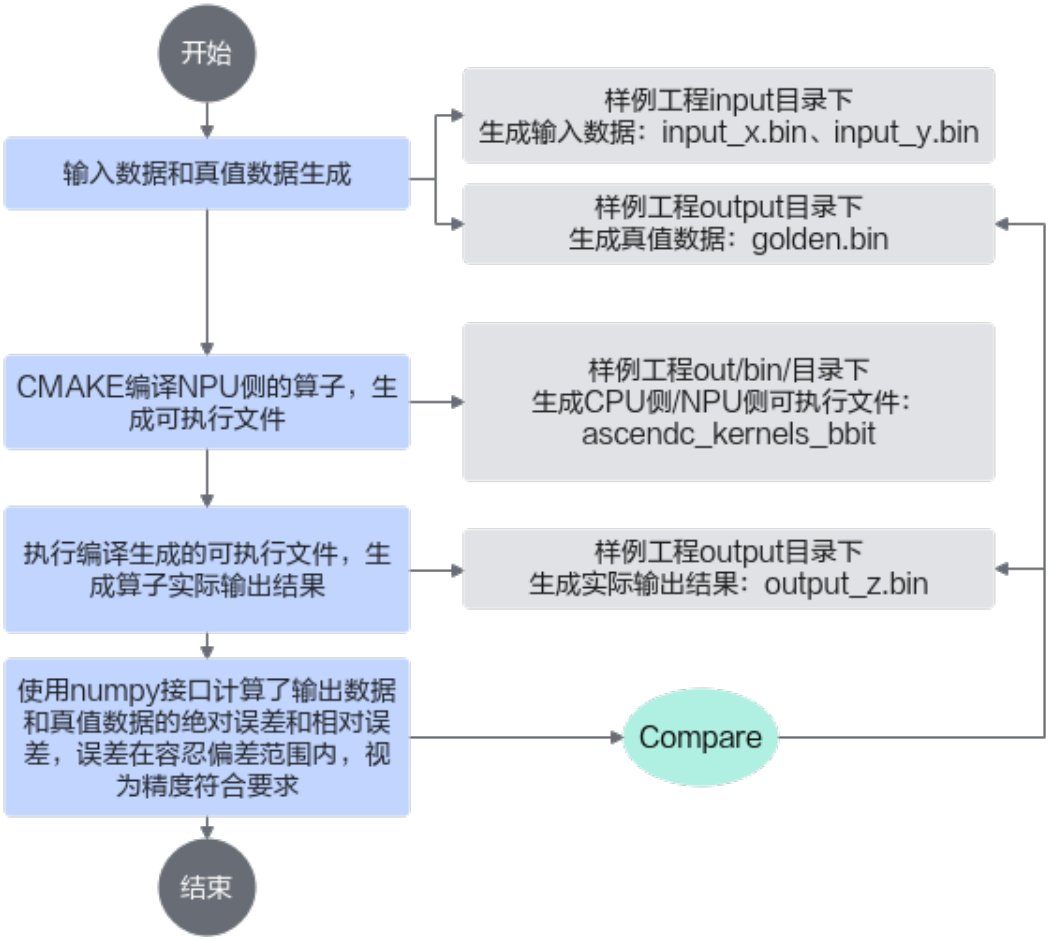

CMake 编译配置文件编写

通常情况下不需要开发者修改编译配置文件，但是了解编译配置文件可以帮助开发者更好的理解编译原理，方便有能力的开发者对Cmake进行定制化处理，具体内容请参考**CMake编译配置文件编写**。

修改并执行一键式编译运行脚本

您可以基于样例工程中提供的一键式编译运行脚本run.sh进行快速编译，在NPU侧执行Ascend C算子。一键式编译运行脚本主要完成以下功能：

图 8-7 一键式编译运行脚本流程图



须知

样例中提供的一键式编译运行脚本并不能适用于所有的算子运行验证场景，使用时请根据实际情况进行修改。

- 根据Ascend C算子的算法原理的不同，自行实现输入和真值数据的生成。

完成上述文件的编写后，可以执行一键式编译运行脚本，编译和运行应用程序。

脚本执行方式和脚本参数介绍如下：

```
bash run.sh --soc-version=<soc_version>
bash run.sh -v <soc_version>
```

表 8-4 脚本参数介绍

参数名	参数简写	参数介绍
--soc-version	-v	算子运行的AI处理器型号。 **说明** AI处理器的型号请通过如下方式获取： ● 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。 该样例支持以下型号： ● Atlas 推理系列产品 ● Atlas 训练系列产品 ● Atlas A2训练系列产品/Atlas 800I A2推理产品

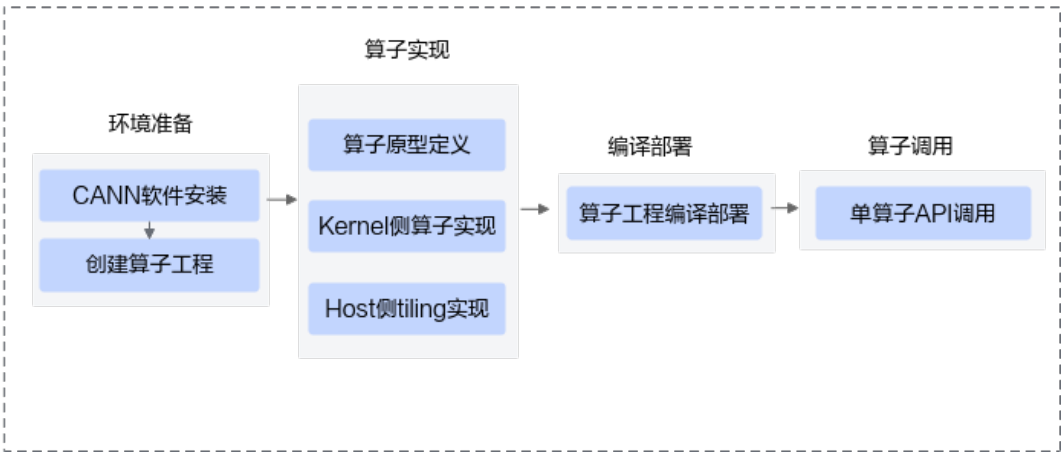
9 工程化算子开发

9.1 概述

工程化算子开发是指基于自动生成的**自定义算子工程**完成算子实现、编译部署、单算子调用代码自动生成等一系列流程。

该开发流程是标准的开发流程，建议开发者按照该流程进行算子开发。该方式下，算子开发的代码会更规范、统一、易于维护；同时该方式考虑了单算子API调用、算子入图、AI框架调用等功能的集成，使得开发者易于借助CANN框架实现上述功能。

工程化算子开发流程如下图所示：



步骤1 环境准备。

1. CANN软件安装请参考**2 环境准备**。
2. **创建算子工程**。使用msOpGen工具创建算子开发工程。

步骤2 算子实现。

- **算子原型定义**。通过原型定义来描述算子输入输出、属性等信息以及算子在AI处理器上相关实现信息，并关联tiling实现等函数。
- Kernel侧算子实现和host侧tiling实现请参考**7 算子实现**；工程化算子开发，支持开发者调用Tiling API基于CANN提供的编程框架进行tiling开发，kernel侧也提供对应的接口方便开发者获取tiling参数，具体内容请参考**9.3.2 Kernel侧算子实现**和**9.3.3 Host侧tiling实现**，由此而带来的额外约束也在上述章节说明。

步骤3 **编译部署**。通过工程编译脚本完成算子的编译部署。

步骤4 **单算子API调用**：调用单算子API接口，基于C语言的API执行算子。

----结束

9.2 创建算子工程

CANN开发套件包中提供了自定义算子工程生成工具msOpGen，可基于算子原型定义输出算子工程：包括**算子host侧代码实现文件**、**算子kernel侧实现文件**以及**工程编译配置文件等**。

说明

使用msOpGen工具创建算子工程之前，需要参考**2 环境准备**章节安装驱动固件和CANN软件包，完成开发环境和运行环境的准备。

使用msOpGen工具创建算子开发工程的步骤如下：

步骤1 编写算子的原型定义json文件，用于生成算子开发工程。

例如，AddCustom算子的json文件命名为add_custom.json，文件内容如下：

```
[
  {
    "op": "AddCustom",
    "input_desc": [
      {
        "name": "x",
        "param_type": "required",
        "format": [
          "ND",
          "ND",
          "ND"
        ],
        "type": [
          "fp16",
          "float",
          "int32"
        ]
      },
      {
        "name": "y",
        "param_type": "required",
        "format": [
          "ND",
          "ND",
          "ND"
        ],
```



```
            "type": [
               "fp16",
               "float",
               "int32"
            ]
         }
      ],
      "output_desc": [
         {
            "name": "z",
            "param_type": "required",
            "format": [
               "ND",
               "ND",
               "ND"
            ],
            "type": [
               "fp16",
               "float",
               "int32"
            ]
         }
      ]
   }
]
```

例如，ReduceMaxCustom算子（包含属性）的json文件命名为reduce_max_custom.json，文件内容如下：

```
[
   {
      "op": "ReduceMaxCustom",
       "input_desc": [
         {
            "name": "x",
            "param_type": "required",
            "format": ["ND"],
            "type": ["float16"]
         }
      ],
      "output_desc": [
         {
            "name": "y",
            "param_type": "required",
            "format": ["ND"],
            "type": ["float16"]
         },
         {
            "name": "idx",
            "param_type": "required",
            "format": ["ND"],
            "type": ["int32"]
         }
      ],
      "attr": [
         {
            "name": "reduceDim",
            "param_type": "required",
            "type": "int"
         },
         {
            "name": "isKeepDim",
            "param_type": "optional",
            "type": "int",
            "default_value": 1
         }
      ]
   }
]
```


步骤2 使用msOpGen工具生成算子的开发工程。以生成AddCustom的算子工程为例，下文仅针对关键参数进行解释，详细参数说明请参见算子工程创建（msOpGen）。

```
${INSTALL_DIR}/python/site-packages/bin/msopgen gen -i $HOME/sample/add_custom.json -c ai_core-<soc_version> -lan cpp -out $HOME/sample/AddCustom
```

- **${INSTALL_DIR}**为CANN软件安装后文件存储路径，请根据实际环境进行替换。
- -i：指定算子原型定义文件*add_custom*.json所在路径，请根据实际情况修改。
- -c：ai_core-<*soc_version*>代表算子在AI Core上执行，<*soc_version*>为昇腾AI处理器的型号。

  说明

  AI处理器的型号<*soc_version*>请通过如下方式获取：

  - 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

  基于同系列的AI处理器型号创建的算子工程，其基础功能（基于该工程进行算子开发、编译和部署）通用。
- -lan： 参数cpp代表算子基于Ascend C编程框架，使用C/C++编程语言开发。
- -out：生成文件所在路径，可配置为绝对路径或者相对路径，并且工具执行用户对路径具有可读写权限。若不配置，则默认生成在执行命令的当前路径。

步骤3 命令执行完后，会在-out指定目录或者默认路径下生成算子工程目录，工程中包含算子实现的模板文件，编译脚本等，以AddCustom算子为例，目录结构如下所示：

```
AddCustom
├── build.sh          // 编译入口脚本
├── cmake
│   ├── config.cmake
│   ├── func.cmake
│   ├── intf.cmake
│   ├── makeself.cmake
│   └── util          // 算子工程编译所需脚本及公共编译文件存放目录
├── CMakeLists.txt    // 算子工程的CMakeLists.txt
├── CMakePresets.json    // 编译配置项
├── framework         // 算子插件实现文件目录，单算子模型文件的生成不依赖算子适配插件，无需关注
├── op_host           // host侧实现文件
│   ├── add_custom_tiling.h    // 算子tiling定义文件
│   ├── add_custom.cpp         // 算子原型注册、shape推导、信息库、tiling实现等内容文件
│   ├── CMakeLists.txt
├── op_kernel         // kernel侧实现文件
│   ├── CMakeLists.txt
│   ├── add_custom.cpp         // 算子代码实现文件
└── scripts           // 自定义算子工程打包相关脚本所在目录
```

说明

上述目录结构中的粗体文件为后续算子开发过程中需要修改的文件，其他文件无需修改。

----结束

工程目录中的op_kernel和op_host包含了算子的核心实现文件。op_kernel下存放**kernel侧算子实现**。op_host下存放host侧代码实现，包括**算子原型定义**、**host侧tiling实现**。其中kernel侧算子实现和host侧tiling实现在**7 算子实现**章节已经介绍了其核心的实现方法，在该章节会侧重于介绍接入CANN框架后的编程模式和API的使用。工程目录中的CMakePresets.json，用于开发者完成工程编译相关配置，之后即可进行编译部署。

9.3 算子实现

9.3.1 算子原型定义

算子原型主要描述了算子的输入输出、属性等信息以及算子在AI处理器上相关实现信息，并关联**tiling实现**等函数。算子原型通过自定义的算子类来承载，该算子类继承自OpDef类。完成算子的原型定义等操作后，需要调用OP_ADD接口，传入算子类型（自定义算子类的类名），进行算子原型注册。下面是一个简单的Add算子原型定义和注册的例子。

```
namespace ops {
class AddCustom : public OpDef {
public:
    AddCustom(const char* name) : OpDef(name)
    {
        this->Input("x")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Input("y")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Output("z")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        // 根据用户的算子调用方式决定需不需要注册 单算子API调用方式下不需要
        this->SetInferShape(ge::InferShape);
        this->SetInferDataType(ge::InferDataType);
        this->AICore()
            .SetTiling(optiling::TilingFunc);
        // 请替换为实际的昇腾AI处理器型号
        this->AICore().AddConfig("ascendxxx");
    }
};
OP_ADD(AddCustom);
} // namespace ops
```

须知

注册算子类型后，框架会根据算子类型获取算子注册信息，同时在编译和运行时按照一定的规则匹配算子实现文件名称和kernel侧核函数名称。为了保证正确匹配，算子类型、算子实现文件名称和核函数名称需要遵循如下定义规则。通常情况下，开发者只需要保证创建算子工程时原型定义json文件中算子类型op的参数值为大驼峰命名方式即可，工程创建后自动生成的代码即满足该规则。在手动编写算子原型定义和算子实现文件时需要按照如下规则定义。

- 算子类型需要采用**大驼峰**的命名方式，即采用大写字符区分不同的语义。
- 算子实现文件名称、核函数名称需相同，均为算子类型转换为**下划线**命名方式后的值。下文描述了通过算子类型转换成算子实现文件名称和核函数名称的过程：
  - 首字符的大写字符转换为小写字符。例如：Abc -> abc。
  - 大写字符的前一个字符为小写字符或数字，则在大写字符前插一个下划线“_”，并将该字符转换为小写字符。例如：AbcDef -> abc_def。
  - 大写字符前一个字符为大写字符且后一个字符是小写字符，则在大写字符前插一个下划线“_”，并将该字符转换为小写字符。例如：AbcAAc -> abc_a_ac。
  - 其他大写字符转换为小写字符，小写字符保持不变。

算子原型定义

算子原型定义描述了算子的输入输出、属性等信息。输入输出支持的datatype、format格式的数量需要一致，并保持一一对应的关系。

如下的代码片段呈现了Add算子输入x的描述信息。

```
    this->Input("x")
        .ParamType(REQUIRED)
        .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
        .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
```

表 9-1 输入输出参数说明

原型定义	参数	具体描述
Input/Output	ParamType	参数类型，Option取值为：OPTIONAL（可选）、REQUIRED（必选）、DYNAMIC（动态输入）。 ● 类似于上文中的Add样例，其输入输出是必选的。 ● 有些算子的输入或者输出个数是动态的，例如AddN，将N个输入Tensor累加到一起，输出一个Tensor；SplitV，将一个Tensor在某个轴上，拆分为N个Tensor输出。 ● 有些算子的输入是可选的，例如BatchNorm算子，在训练的时候没有均值和方差输入，在推理的时候有均值和方差的输入。
	DataType	算子输入输出支持的datatype。datatype的取值请参考DataType。
	Format	算子输入输出支持的format。format的取值请参考Format。

从上文的原型定义中可以看出，列出了输入输出所有datatype和format的组合，保持一一对应。使用如下接口，可以达到简化这种代码逻辑的目的。

- 在指定输入/输出的datatype信息时，如果某个输入/输出的datatype支持和其他所有输入/输出的datatype/format组合使用，其datatype可以通过DataTypeList来表达；在指定输入/输出的format信息时，如果某个输入/输出的format支持和其他所有输入/输出的datatype/format组合使用，其format可以通过FormatList来表达。示例如下，以下两种代码表达含义相同。

```
// 列出所有一一对应的组合
class XxxCustom : public OpDef {
public:
    XxxCustom(const char* name) : OpDef(name)
    {
        this->Input("x")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT16, ge::DT_FLOAT16})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Input("y")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Output("z")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
```



```
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        ...
    }
};
// 通过DataTypeList和FormatList来表达，无需重复列出
class XxxCustom : public OpDef {
public:
    XxxCustom(const char* name) : OpDef(name)
    {
        this->Input("x")
            .ParamType(REQUIRED)
            .DataTypeList({ge::DT_FLOAT16})
            .FormatList({ge::FORMAT_ND});
        this->Input("y")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Output("z")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        ...
    }
};
```

- 通过Follow接口指定当前输入/输出的datatype/format/shape信息与之前定义过的某个输入一致。示例如下：输出“y1”Follow输入“x1”场景，此时“y1”的datatype、format以及shape都将会和“x1”保持一致。使用Follow接口指定shape一致时通常比**shape推导**函数逻辑更加简单，能用Follow表达的逻辑，建议使用Follow接口，则无需再编写注册InferShape函数。

```
this->Input("x1")
    .ParamType(REQUIRED)
    .DataType({ge::DT_FLOAT, ge::DT_FLOAT})
    .Format({ge::FORMAT_ND, ge::FORMAT_ND});
this->Input("x2")
    .ParamType(REQUIRED)
    .DataType({ge::DT_FLOAT, ge::DT_FLOAT})
    .Format({ge::FORMAT_ND, ge::FORMAT_ND});
this->Output("y1")
    .ParamType(REQUIRED)
    .Follow("x1")
    .OutputShapeDependOnCompute();
```

原型定义中还包括算子属性信息，如下的代码片段呈现了ReduceMax算子的属性reduceDim和isKeepDim的描述信息。

```
this->Attr("reduceDim")
    .AttrType(REQUIRED)
    .Int();
this->Attr("isKeepDim")
    .AttrType(OPTIONAL)
    .Int(1);
```

具体参数说明如下：

表 9-2 属性参数说明

原型定义	注册方式	具体描述
Attr	AttrType	设置算子属性类型，取值为：OPTIONAL（可选）、REQUIRED（必选）。

原型定义	注册方式	具体描述
	Bool/ Float/ Int...	设置算子属性数据类型为Bool/Float/Int...。具体说明请参考OpAttrDef。

AI 处理器上相关实现信息

通过AddConfig注册算子支持的AI处理器型号以及相关的配置信息。AddConfig接口原型如下：soc参数表示AI处理器型号，aicore_config表示其他配置信息。

```
void AddConfig(const char *soc);
void AddConfig(const char *soc, OpAICoreConfig &aicore_config);
```

通过该接口注册AI处理器型号的样例如下，ascendxxx请替换为实际的AI处理器型号。

```
    this->AICore().AddConfig("ascendxxx");
```

其他AI Core配置信息的配置方式请参考OpAICoreConfig。

关联 Tiling 实现、Shape 推导等函数

通过SetInferShape、SetInferDataType、SetTiling接口来关联对应的shape推导函数和Tiling函数，样例如下。

```
    this->SetInferShape(ge::InferShape);
    this->SetInferDataType(ge::InferDataType);
    this->AICore()
        .SetTiling(optiling::TilingFunc);
```

多硬件平台注册差异化的算子原型

算子类继承基类OpDef，使用Input、Output、Attr等注册算子原型信息，硬件平台支持相同的算子原型的情况下，直接通过AICore().AddConfig添加支持的AI处理器型号即可；不同的硬件形态算子原型定义不同的情况，可以通过新增OpAICoreConfig的方式，针对不同的AI处理器型号注册差异化的算子原型。

差异化的算子原型生效规则如下：

- 对于算子类的输入输出原型信息，OpAICoreConfig未配置的会继承OpDef定义的原型，比如算子类中定义了输出y，OpAICoreConfig中没有定义输出y，OpAICoreConfig会继承y的原型定义；
- 对于算子类和新增OpAICoreConfig中定义的算子原型相同的情况，新增OpAICoreConfig中定义的算子原型信息会覆盖OpDef定义的原型信息，比如算子类中定义了输入x支持DT_FLOAT16数据类型，新增OpAICoreConfig中也定义了输入x，但是支持DT_FLOAT16、DT_BF16数据类型，则以OpAICoreConfig新增定义为准。

如下样例中ascendxxx1、ascendxxx2（AI处理器型号）使用相同的算子原型，算子类通过继承基类OpDef，使用Input、Output、Attr等注册算子原型信息，再通过AICore().AddConfig添加支持的AI处理器型号；对于ascendxxx3支持的算子原型需要定制化处理，新增了DT_BF16的类型，通过新增OpAICoreConfig的方式进行注册，x，y，z的定义会覆盖算子类中对应定义的原型信息。

```
namespace ops {
class MyAdd : public OpDef {
```



```
public:
    MyAdd(const char* name) : OpDef(name)
    {
        // ascendxxx1 ascendxxx2 AI处理器型号原型定义
        this->Input("x")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16})
            .Format({ge::FORMAT_ND});
        this->Input("y")
            .ParamType(OPTIONAL)
            .DataType({ge::DT_INT64})
            .ValueDepend(REQUIRED)
            .Format({ge::FORMAT_ND});
        this->Output("z")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16})
            .Format({ge::FORMAT_ND});
        this->AICore()
            .SetTiling(optiling::TilingFunc);
        this->AICore().AddConfig("ascendxxx1");
        this->AICore().AddConfig("ascendxxx2");
        // ascendxxx3芯片定义OpAICoreConfig变量，定制化原型
        OpAICoreConfig config;
        config.Input("x")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_BF16})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND});
        config.Input("y")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_BF16})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND});
        config.Output("z")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_BF16})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND});
        this->AICore().AddConfig("ascendxxx3", config);
    }
};
OP_ADD(MyAdd);
}
```

如下的样例中，只有几个参数原型信息在不同硬件平台不一致，开发者也可以通过OpAICoreConfig定制部分算子原型信息，复用OpDef定义的其他算子原型信息，达到部分原型信息硬件平台定制化的目的。

```
class AddCustom : public OpDef {
public:
    AddCustom(const char* name) : OpDef(name)
    {
        this->Input("x").DataType({ ge::DT_FLOAT16 }).ParamType(OPTIONAL);
        this->Output("y").DataType({ ge::DT_FLOAT16 });
        OpAICoreConfig aicConfig1;
        OpAICoreConfig aicConfig2;
        aicConfig1.Input("x")
            .ParamType(OPTIONAL)
            .DataType({ ge::DT_FLOAT })
            .Format({ ge::FORMAT_ND });
        aicConfig2.Input("x")
            .ParamType(REQUIRED)
            .DataType({ ge::DT_INT32 })
            .Format({ ge::FORMAT_ND });
        this->AICore().AddConfig("ascendxxx1", aicConfig1);
        this->AICore().AddConfig("ascendxxx2", aicConfig2);
    }
};
```


9.3.2 Kernel 侧算子实现

在7 算子实现章节已经介绍了kernel侧算子核心的实现方法，本章节侧重于介绍接入CANN框架时编程模式和API的使用。

自动生成 kernel 侧算子实现模板

在算子工程目录下的“op_kernel/xxx.cpp”文件中实现算子的核函数。核函数的定义模板已通过msOpGen工具自动生成，样例如下所示。**注意这里参数的顺序按照“输入、输出、workspace、tiling”的顺序排布，开发者不要调整其顺序。**

```
#include "kernel_operator.h"
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR
workspace, GM_ADDR tiling) {
    GET_TILING_DATA(tiling_data, tiling);// 获取Tiling参数，详见下文介绍
    // TODO: user kernel impl
}
```

说明

算子原型定义中的输入和输出同名的情况下，自动生成的核函数中，输出参数增加ref后缀予以区分。示例如下：

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR x_ref,
GM_ADDR workspace, GM_ADDR tiling) {
    ...
}
```

GET_TILING_DATA 获取 Tiling 参数

提供GET_TILING_DATA，用于获取算子kernel入口函数传入的tiling信息，并填入注册的Tiling结构体中，此函数会以宏展开的方式进行编译。注意，对应的算子host实现中需要定义TilingData结构体，实现并注册计算TilingData的Tiling函数。具体请参考9.3.3 Host侧tiling实现。

核函数中调用GET_TILING_DATA获取TilingData的样例如下：

```
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR
workspace, GM_ADDR tiling)
{
    GET_TILING_DATA(tilingData, tiling);
    KernelAdd op;
    op.Init(x, y, z, tilingData.totalLength, tilingData.tileNum);
    if (TILING_KEY_IS(1)) {
        op.Process();
    }
}
```

核函数内推导输入数据类型和格式

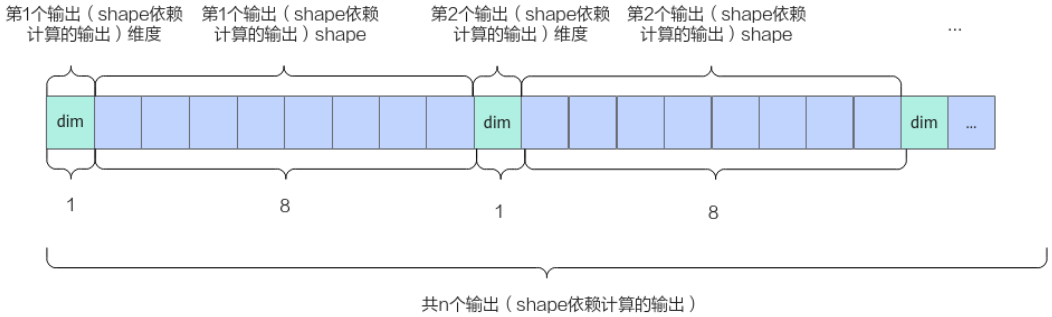
算子工程在核函数内提供了DTYPE_<Arg>、ORIG_DTYPE_<Arg>、FORMAT_<Arg>三种宏用于推导核函数入参的数据类型、原始数据类型和数据格式。其中<Arg>会自动大写。样例如下：

```
template<class T> func() {}
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR
workspace, GM_ADDR tiling)
{
    DTYPE_X temp;
    func<DTYPE_Z>();
    if (FORMAT_Y == FORMAT_ND) {
        ...
    }
}
```

输出 shape 依赖计算的算子 kernel 侧实现

某些算子，比如NonZero（统计tensor中非零值的个数），计算完成前无法得知算子输出的shape信息，算子计算完成后才能获取。该类算子在原型定义时，需要使用OutputShapeDependOnCompute接口进行标识，同时在算子核函数中将实际输出shape写入到出参中，便于框架侧基于该信息进行输出内存的管理。

在核函数所有输出的最后增加一个GM_ADDR类型的输出参数，并在核函数计算完成后，将输出shape信息写入到该出参中。shape信息的排布格式如下，大小为**n * (8 + 1)**，每个元素的数据类型为**uint64_t**。其中n表示待刷新shape信息的输出个数，每个输出的shape信息都通过第1个元素来保存实际的shape维度（dim），后续的8个元素来保存具体每个维度的shape信息。



说明

- 输出的顺序和原型定义中输出的顺序保持一致。
- 对于uint64_t的输出数据类型（对于tensor而言），需要将dim的uint32_t的高位设置为1，表示以uint64_t类型解析该tensor。

- 如下示例中，算子中有一个输出依赖计算得出，输出tensor的数据类型为uint32_t，计算完成后，得到输出的shape为（32, 64），出参shape_out用于存放该shape信息，值为（2, 32, 64）。代码示例如下：

```
extern "C" __global__ __aicore__ void xxx_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR shape_out, GM_ADDR workspace, GM_ADDR tiling) {
...
    constexpr uint32_t SHAPEOUT_SIZE = 9;
    // 输出数据为2维([32, 64])，tensor类型为uint32_t
    GlobalTensor<uint64_t> shapeoutGlobal_uint32;
    shapeoutGlobal_uint32.SetGlobalBuffer((__gm__ uint64_t*)shape_out, SHAPEOUT_SIZE);
    shapeoutGlobal_uint32.SetValue(0, 2);
    shapeoutGlobal_uint32.SetValue(1, 32);
    shapeoutGlobal_uint32.SetValue(2, 64);
...
}
```

- 如下示例中，算子中有一个输出依赖计算得出，输出tensor的数据类型为uint64_t，计算完成后，得到输出的shape为（32, 64），出参shape_out用于存放该shape信息，值为（0x0000000010000000 | 2, 32, 64）。代码示例如下：

```
extern "C" __global__ __aicore__ void xxx_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR shape_out, GM_ADDR workspace, GM_ADDR tiling) {
...
    constexpr uint32_t SHAPEOUT_SIZE = 9;
    // 输出数据为4维([1, 64, 128, 128])，tensor类型为uint64_t
    GlobalTensor<uint64_t> shapeoutGlobal_uint64;
    shapeoutGlobal_uint64.SetGlobalBuffer((__gm__ uint64_t*)shape_out, SHAPEOUT_SIZE);
    shapeoutGlobal_uint64.SetValue(0, 0x0000000010000000 | 4);
    shapeoutGlobal_uint64.SetValue(1, 1);
    shapeoutGlobal_uint64.SetValue(2, 64);
    shapeoutGlobal_uint64.SetValue(3, 128);
    shapeoutGlobal_uint64.SetValue(4, 128);
```



```
...
}
```

- 如下示例中，算子中有两个输出依赖计算得出，输出tensor的数据类型为uint64_t，计算完成后，得到输出的shape为（16, 32）和（1, 16, 16, 32），出参shape_out用于存放该shape信息。示例如下：

```
extern "C" __global__ __aicore__ void xxx_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR shape_out, GM_ADDR workspace, GM_ADDR tiling) {
    ...
    // 有两个输出需要刷新shape，一个维度为2维[16, 32]，一个维度为4维[1, 16, 16, 32]
    // tensor类型为uint64_t
    constexpr uint32_t SHAPEOUT_SIZE_2 = 18;
    GlobalTensor<uint64_t> shapeoutGlobal_uint64_2;
    shapeoutGlobal_uint64_2.SetGlobalBuffer((__gm__ uint64_t*)shape_out, SHAPEOUT_SIZE_2 );
    shapeoutGlobal_uint64_2.SetValue(0, 0x0000000010000000 | 2);
    shapeoutGlobal_uint64_2.SetValue(1, 16);
    shapeoutGlobal_uint64_2.SetValue(2, 32);
    // index[3]~index[8]数据为占位
    shapeoutGlobal_uint64_2.SetValue(9, 0x0000000010000000 | 4);
    shapeoutGlobal_uint64_2.SetValue(10, 1);
    shapeoutGlobal_uint64_2.SetValue(11, 16);
    shapeoutGlobal_uint64_2.SetValue(12, 16);
    shapeoutGlobal_uint64_2.SetValue(13, 32);
    ...
}
```

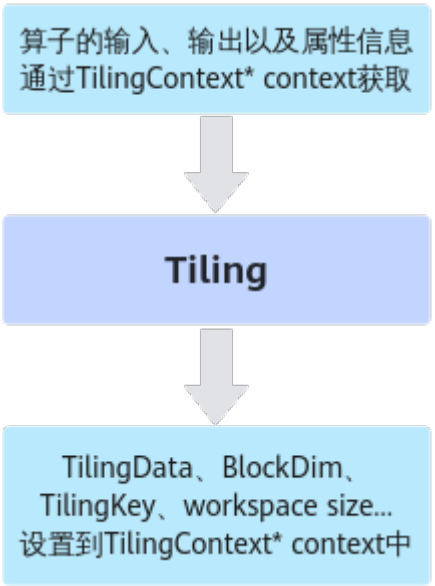
9.3.3 Host 侧 tiling 实现

在**7 算子实现**章节已经介绍了host侧tiling核心的实现方法，本章节侧重于介绍接入CANN框架时编程模式和API的使用。

大多数情况下，Local Memory的存储，无法完整的容纳算子的输入与输出，需要每次搬运一部分输入进行计算然后搬出，再搬运下一部分输入进行计算，直到得到完整的最终结果，这个数据切分、分块计算的过程称之为**Tiling**。根据算子的shape等信息来确定数据切分算法相关参数（比如每次搬运的块大小，以及总共循环多少次）的计算程序，称之为**Tiling实现**。

Tiling实现完成后，获取到的Tiling切分算法相关参数，会传递给kernel侧，用于指导并行数据的切分。由于Tiling实现中完成的均为标量计算，AI Core并不擅长，所以我们将其独立出来放在host CPU上执行。

图 9-1 Tiling 实现的输入输出



如上图所示，Tiling实现即为根据算子shape等信息来确定切分算法相关参数的过程，这里的算子shape等信息可以理解为是**Tiling实现的输入**，切分算法相关参数可以理解为是**Tiling实现的输出**。输入和输出都通过Tiling函数的参数（TilingContext* context上下文结构）来承载。也就是说，开发者可以从上下文结构中获取算子的输入、输出以及属性信息，也就是**Tiling实现的输入**，经过Tiling计算后，获取到TilingData数据结构（切分算法相关参数）、blockDim变量、用于选择不同的kernel实现分支的TilingKey、算子workspace的大小，也就是**Tiling实现的输出**，并将这些输出设置到上下文结构中。

TilingData、blockDim、TilingKey、workspace这些概念的具体解释如下：

- **TilingData：**切分算法相关参数，比如每次搬运的块大小，以及总共循环多少次，通过结构体存储，由开发者自行设计。

  TilingData结构定义支持单结构定义方法，也支持结构体嵌套：

  - 单结构定义方法，以平铺的形式定义：

    ```
    namespace optiling {
    BEGIN_TILING_DATA_DEF(MyAddTilingData)  // 声明tiling结构名字
      TILING_DATA_FIELD_DEF(uint32_t, field1);   // 结构成员的类型和名字
      TILING_DATA_FIELD_DEF(uint32_t, field2);
      TILING_DATA_FIELD_DEF(uint32_t, field3);
    END_TILING_DATA_DEF;

    REGISTER_TILING_DATA_CLASS(MyAdd, MyAddTilingData)  // tiling结构注册给算子
    }
    ```

    Tiling实现函数中对tiling结构成员赋值的方式如下：

    ```
    MyAddTilingData myTiling;
    myTiling.set_field1(1);
    myTiling.set_field2(2);
    ```

  - 支持结构体嵌套：

    ```
    namespace optiling {
    BEGIN_TILING_DATA_DEF(MyStruct1)  // 声明结构1名字
      TILING_DATA_FIELD_DEF(uint32_t, field1);   // 结构成员的类型和名字
      TILING_DATA_FIELD_DEF(uint32_t, field2);   // 结构成员的类型和名字
    END_TILING_DATA_DEF;
    REGISTER_TILING_DATA_CLASS(MyStruct1Op, MyStruct1) // 注册结构体到<op_type>Op

    BEGIN_TILING_DATA_DEF(MyStruct2)  // 声明结构2名字
      TILING_DATA_FIELD_DEF(uint32_t, field3);   // 结构成员的类型和名字
      TILING_DATA_FIELD_DEF(uint32_t, field4);   // 结构成员的类型和名字
    END_TILING_DATA_DEF;
    REGISTER_TILING_DATA_CLASS(MyStruct2Op, MyStruct2) // 注册结构体到<op_type>Op

    BEGIN_TILING_DATA_DEF(MyAddTilingData)  // 声明tiling结构名字
      TILING_DATA_FIELD_DEF_STRUCT(MyStruct1, st1);   // 结构成员的引用结构体
      TILING_DATA_FIELD_DEF_STRUCT(MyStruct2, st2);   // 结构成员的引用结构体
    END_TILING_DATA_DEF;
    REGISTER_TILING_DATA_CLASS(MyAdd, MyAddTilingData)  // tiling结构注册给算子
    }
    ```

    Tiling实现函数中对tiling结构成员赋值的方式如下：

    ```
    MyAddTilingData myTiling;
    myTiling.st1.set_field1(1);
    myTiling.st1.set_field2(2);
    myTiling.st2.set_field3(3);
    myTiling.st2.set_field4(4);
    ```

- **blockDim：**规定了核函数将会在几个核上执行。例如，需要计算8M的数据，每个核上计算1M的数据，blockDim设置为8，但是为了充分利用硬件资源，一般将blockDim设置为硬件平台的核数，根据核数进行数据切分。

说明

blockDim是逻辑核的概念，取值范围为[1,65535]。为了充分利用硬件资源，一般设置为物理核的核数或其倍数。对于耦合架构和分离架构，blockDim在运行时的意义和设置规则有一些区别，具体说明如下：

- 耦合架构：由于其Vector、Cube单元是集成在一起的，blockDim用于设置启动多个AICore核实例执行，不区分Vector、Cube。AI Core的核数可以通过GetCoreNumAiv或者GetCoreNumAic获取。
- 分离架构
  - 针对仅包含Vector计算的算子，blockDim用于设置启动多少个Vector（AIV）实例执行，比如某款AI处理器上有40个Vector核，建议设置为40。
  - 针对仅包含Cube计算的算子，blockDim用于设置启动多少个Cube（AIC）实例执行，比如某款AI处理器上有20个Cube核，建议设置为20。
  - 针对Vector/Cube融合计算的算子，启动时，按照AIV和AIC组合启动，blockDim用于设置启动多少个组合执行，比如某款AI处理器上有40个Vector核和20个Cube核，一个组合是2个Vector核和1个Cube核，建议设置为20，此时会启动20个组合，即40个Vector核和20个Cube核。**注意：该场景下，设置的blockDim逻辑核的核数不能超过物理核（2个Vector核和1个Cube核组合为1个物理核）的核数。**
  - AIC/AIV的核数分别通过GetCoreNumAic和GetCoreNumAiv接口获取。

- **TilingKey（可选）**：TilingKey是一个算子内为了区分不同的实现而将kernel代码进行区分的方法，该方法类似于C++的Template模板机制，可减少不必要的icache miss以及scalar耗时，有助于优化单次调用kernel的性能。不同的kernel实现分支可以通过TilingKey来标识，host侧设置TilingKey后，可以选择对应的分支。例如，一个算子在不同的shape下，有不同的算法逻辑，kernel侧可以通过TilingKey来选择不同的算法逻辑，在host侧Tiling算法也有差异，host/kernel侧通过相同的TilingKey进行关联。

  假如有如下kernel代码：

  ```
  if (condition) {
    ProcessA();
  } else {
    ProcessB();
  }
  ```

  如果函数ProcessA、ProcessB两个函数是个非常大的函数，那么上述代码在编译后会变得更大，而每次kernel运行只会选择1个分支，条件的判断和跳转在代码大到一定程度（16-32K，不同芯片存在差异）后会出现icache miss。通过TilingKey可以对这种情况进行优化，给2个kernel的处理函数设置不同的TilingKey 1和2：

  ```
  if (TILING_KEY_IS(1)) {
    ProcessA();
  } else if (TILING_KEY_IS(2)) {
    ProcessB();
  }
  ```

  这样device kernel编译时会自动识别到2个TilingKey并编译2个kernel入口函数，将条件判断进行常量折叠。同时需要和host tiling函数配合，判断走ProcessA的场景设置TilingKey为1，走ProcessB的场景设置TilingKey为2：

  ```
  static ge::graphStatus TilingFunc(gert::TilingContext* context)
  {
      // some code
      if (condition) {
          context->SetTilingKey(1);
      } else {
          context->SetTilingKey(2);
      }
      return ge::GRAPH_SUCCESS;
  }
  ```


> **说明**
>
> 编译时，可以通过设置**--tiling_keys**编译选项指定TilingKey，编译时只编译指定TilingKey相关的kernel代码，用于加速编译过程。

- **WorkspaceSize（可选）**：workspace是设备侧Global Memory上的一块内存。在Tiling函数中可以设置workspace的大小，框架侧会为其在申请对应大小的设备侧Global Memory，在对应的算子kernel侧实现时可以使用这块workspace内存。

  workspace内存分为两部分：Ascend C API需要的workspace内存和算子实现使用到的workspace内存（按需）。

  - AscendC API需要预留workspace内存

    API在计算过程需要一些workspace内存作为缓存，因此算子Tiling函数需要为API预留workspace内存，预留内存大小通过GetLibApiWorkSpaceSize接口获取。
  - 算子实现使用到的workspace内存（按需）

    算子内部需要通过额外的device内存进行数据交换或者缓存的时候才需要分配，根据算子计算的空间自行分配。

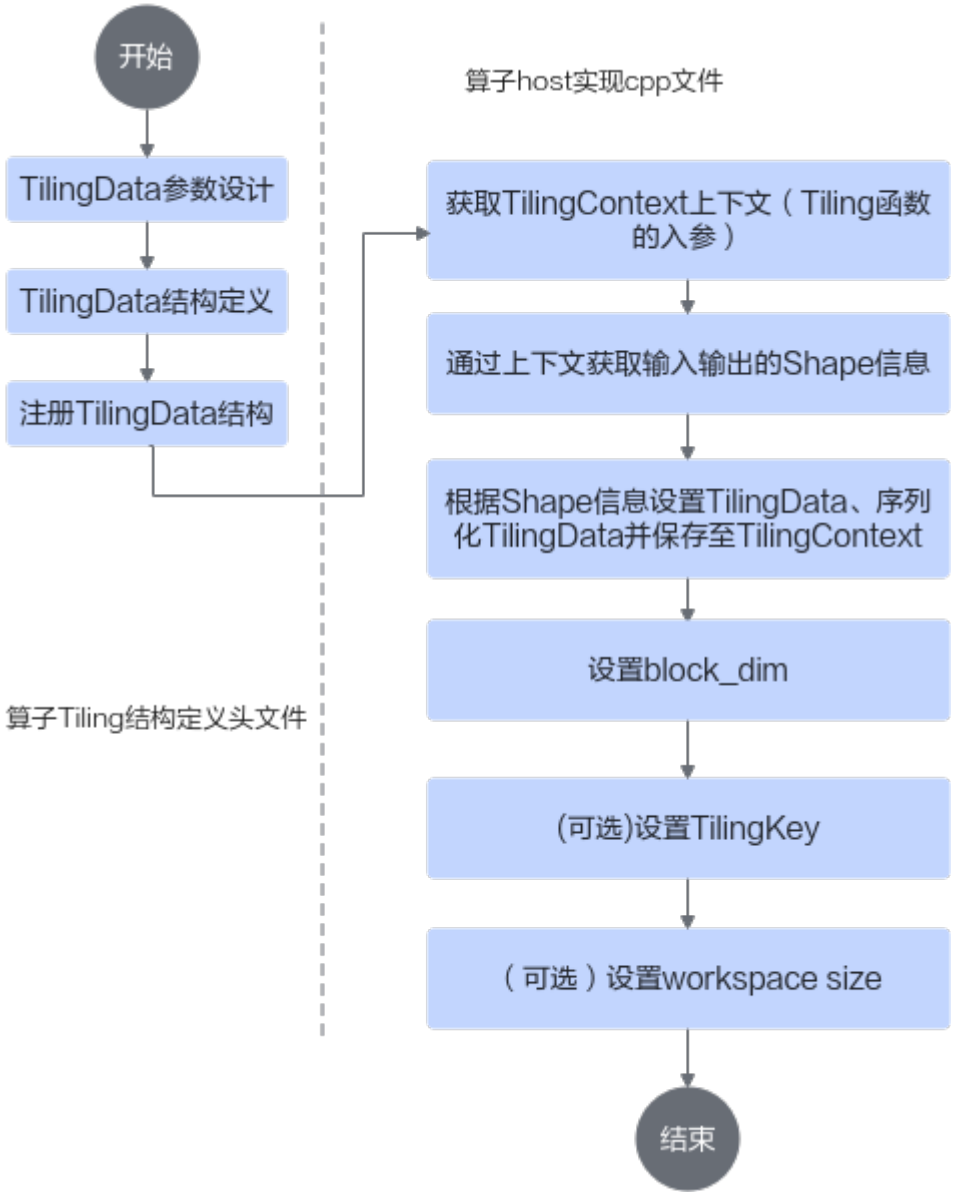
  整体的workspace内存就是上述两部分之和，在Tiling函数中设置方法如下：

  ```
  auto workspaceSizes = context->GetWorkspaceSizes(1); // 只使用1块workspace
  workspaceSizes[0] = sysWorkspaceSize + usrWorkspaceSize;
  ```

Tiling 实现基本流程

Tiling实现开发的流程图如下：

图 9-2 Tiling 开发流程图



下面将从一个简单的Add算子为例介绍Tiling的实现流程。本样例中待处理数据的Shape大小可以平均分配到每个核上，并且可以对齐到一个datablock（32B）的大小。

首先完成算子TilingData结构定义头文件的编写，该文件命名为*“算子名称_tiling.h”*，位于算子工程的op_host目录下。样例代码如下：

```
#ifndef ADD_CUSTOM_TILING_H
#define ADD_CUSTOM_TILING_H
#include "register/tilingdata_base.h"

namespace optiling {
BEGIN_TILING_DATA_DEF(TilingData)               // 注册一个tiling的类，以tiling的名字作为入参
  TILING_DATA_FIELD_DEF(uint32_t, totalLength); // 添加tiling字段，总计算数据量
  TILING_DATA_FIELD_DEF(uint32_t, tileNum);     // 添加tiling字段，每个核上总计算数据分块个数
END_TILING_DATA_DEF;
// 注册算子tilingdata类到对应的AddCustom算子
REGISTER_TILING_DATA_CLASS(AddCustom, TilingData)
```



```
}
#endif // ADD_CUSTOM_TILING_H
```

具体的编写步骤如下：

步骤1 代码框架编写，需要增加#ifndef...的判断条件，防止头文件的重复包含；需要包含register/tilingdata_base.h头文件，tilingdata_base.h中定义了多个用于tilingdata注册的宏。样例代码如下：

```
#ifndef ADD_CUSTOM_TILING_H
#define ADD_CUSTOM_TILING_H
#include "register/tilingdata_base.h"

namespace optiling {
// tiling结构定义和注册代码
// ...
}
#endif // ADD_CUSTOM_TILING_H
```

步骤2 TilingData参数设计，TilingData参数本质上是和并行数据切分相关的参数，本示例算子使用了2个tiling参数：totalLength、tileNum。totalLength是指需要计算的数据量大小，tileNum是指每个核上总计算数据分块个数。比如，totalLength这个参数传递到kernel侧后，可以通过除以参与计算的核数，得到每个核上的计算量，这样就完成了多核数据的切分。

步骤3 TilingData结构定义，通过BEGIN_TILING_DATA_DEF接口定义一个TilingData的类，通过TILING_DATA_FIELD_DEF接口增加TilingData的两个字段totalLength、tileNum，通过END_TILING_DATA_DEF接口结束TilingData定义。相关接口的详细说明请参考TilingData结构定义。

```
BEGIN_TILING_DATA_DEF(TilingData)               // 注册一个tiling的类，以tiling的名字作为入参
  TILING_DATA_FIELD_DEF(uint32_t, totalLength); // 添加tiling字段，总计算数据量
  TILING_DATA_FIELD_DEF(uint32_t, tileNum);     // 添加tiling字段，每个核上总计算数据分块个数
END_TILING_DATA_DEF;
```

步骤4 注册TilingData结构，通过REGISTER_TILING_DATA_CLASS接口，注册TilingData类，和自定义算子相关联。REGISTER_TILING_DATA_CLASS第一个参数为op_type(算子类型)，本样例中传入AddCustom，第二个参数为TilingData的类名。REGISTER_TILING_DATA_CLASS接口介绍请参考TilingData结构注册。

```
// 注册算子tilingdata类到对应的AddCustom算子
REGISTER_TILING_DATA_CLASS(AddCustom, TilingData)
```

----结束

然后完成算子host实现cpp文件中Tiling函数实现，该文件命名为*“算子名称.cpp”*，位于算子工程的op_host目录下。样例代码如下：

```
namespace optiling {
const uint32_t BLOCK_DIM = 8;
const uint32_t TILE_NUM = 8;
static ge::graphStatus TilingFunc(gert::TilingContext *context)
{
    TilingData tiling;
    uint32_t totalLength = context->GetInputShape(0)->GetOriginShape().GetShapeSize();
    context->SetBlockDim(BLOCK_DIM);
    tiling.set_totalLength(totalLength);
    tiling.set_tileNum(TILE_NUM);
    tiling.SaveToBuffer(context->GetRawTilingData()->GetData(), context->GetRawTilingData()-
>GetCapacity());
    context->GetRawTilingData()->SetDataSize(tiling.GetDataSize());
    size_t *currentWorkspace = context->GetWorkspaceSizes(1);
    currentWorkspace[0] = 0;
    return ge::GRAPH_SUCCESS;
}
} // namespace optiling
```


具体步骤如下：

步骤1 获取TilingContext的上下文，即Tiling函数的入参gert::TilingContext* context。

步骤2 设置TilingData。上文**步骤3**中，定义了TilingData的类，此时可以用TilingData定义一个具体的实例，通过调用TilingData类的set_+*field_name*接口来设置TilingData的字段值，通过调用TilingData类的SaveToBuffer接口完成TilingData的序列化和保存。

1. 通过上下文获取输入输出shape信息。本样例中通过TilingContext的GetInputShape接口获取输入的shape大小。
```
// 获取输入shape信息
uint32_t totalLength = context->GetInputShape(0)->GetOriginShape().GetShapeSize();
```
2. 设置TilingData。通过调用set_+*field_name*方法来设置TilingData的字段值。
```
// 用TilingData定义一个具体的实例
TilingData tiling;
// 设置TilingData
tiling.set_totalLength(totalLength);
tiling.set_tileNum(TILE_NUM);
```
3. 调用TilingData类的SaveToBuffer接口完成序列化并保存至TilingContext上下文。SaveToBuffer的第一个参数为存储Buffer的首地址，第二个参数为Buffer的长度。通过调用GetRawTilingData获取无类型的TilingData的地址，再通过GetData获取数据指针，作为Buffer的首地址；通过调用GetRawTilingData获取无类型的TilingData的地址，再通过GetCapacity获取TilingData的长度，作为Buffer的长度。完成SaveToBuffer操作后需要通过SetDataSize设置TilingData的长度，该长度通过TilingData类的GetDataSize接口获取。
```
// 序列化并保存
tiling.SaveToBuffer(context->GetRawTilingData()->GetData(), context->GetRawTilingData()->GetCapacity());
context->GetRawTilingData()->SetDataSize(tiling.GetDataSize());
```

步骤3 通过SetBlockDim接口设置blockDim。
```
context->SetBlockDim(BLOCK_DIM);
```

步骤4 （可选）通过SetTilingKey设置TilingKey。
```
context->SetTilingKey(1);
```

步骤5 （可选）通过GetWorkspaceSizes获取workspace size指针，并设置size大小。此处仅作为举例，设置workspace的大小为0。
```
size_t *currentWorkspace = context->GetWorkspaceSizes(1);
currentWorkspace[0] = 0;
```

----结束

Tiling 参数设计更多样例-属性信息通过 TilingData 传递

如果算子包含属性信息，该属性信息可以通过TilingData传递到kernel侧，参与kernel侧算子核函数的计算。以ReduceMaxCustom算子为例，该算子用于对输入数据按维度dim返回最大值，并且返回索引。ReduceMaxCustom算子有两个属性，reduceDim和isKeepDim，reduceDim表示按照哪一个维度进行reduce操作；isKeepDim表示是否需要保持输出的维度与输入一样。本样例仅支持对最后一维做reduce操作，输入数据类型为half。

1. ReduceMaxCustom算子TilingData的定义如下：这里我们重点关注reduceAxisLen。参数reduceAxisLen表示获取reduceDim轴的长度，这里也就是最后一维的长度。该参数后续会通过TilingData传递到kernel侧参与计算。
```
#ifndef REDUCE_MAX_CUSTOM_TILING_H
#define REDUCE_MAX_CUSTOM_TILING_H
#include "register/tilingdata_base.h"
namespace optiling {
```



```
BEGIN_TILING_DATA_DEF(ReduceMaxTilingData)
  TILING_DATA_FIELD_DEF(uint32_t, reduceAxisLen); // 添加tiling字段，reduceDim轴的长度
  //其他TilingData参数的定义
  ...
END_TILING_DATA_DEF;
// 注册算子tilingdata类到对应的ReduceMaxCustom算子
REGISTER_TILING_DATA_CLASS(ReduceMaxCustom, ReduceMaxTilingData)
}
#endif // REDUCE_MAX_CUSTOM_TILING_H
```

2. ReduceMaxCustom算子的Tiling实现如下。这里我们重点关注属性信息通过TilingData传递的过程：首先通过TilingContext上下文从attr获取reduceDim属性值；然后根据reduceDim属性值获取reduceDim轴的长度并设置到TilingData中。

```
namespace optiling {
static ge::graphStatus TilingFunc(gert::TilingContext* context)
{
    ReduceMaxTilingData tiling;
    // 从attr获取reduceDim属性值，因为reduceDim是第一个属性，所以GetAttrPointer传入的索引值为0
    const gert::RuntimeAttrs* attrs = context->GetAttrs();
    const uint32_t* reduceDim = attrs->GetAttrPointer<uint32_t>(0);
    // 获取reduceDim轴的长度
    const gert::StorageShape* xShapePtr = context->GetInputShape(0);
    const gert::Shape& xShape = xShapePtr->GetStorageShape();
    const uint32_t reduceAxisLen = xShape.GetDim(*reduceDim);
    // 计算TilingData中除了reduceAxisLen之外其他成员变量的值
    ...
    // 将reduceAxisLen设置到tiling结构体中，传递到kernel函数使用
    tiling.set_reduceAxisLen(reduceAxisLen);
    // 设置TilingData中除了reduceAxisLen之外其他成员变量的值
    ...
    // TilingData序列化保存
    tiling.SaveToBuffer(context->GetRawTilingData()->GetData(), context->GetRawTilingData()-
>GetCapacity());
    context->GetRawTilingData()->SetDataSize(tiling.GetDataSize());
    ...
    return ge::GRAPH_SUCCESS;
}} // namespace optiling
```

Tiling 参数设计更多样例-使用高阶 API 时配套的 Tiling

1. 首先进行tiling结构定义：

```
namespace optiling {
BEGIN_TILING_DATA_DEF(MyAddTilingData)  // 声明tiling结构名字
  TILING_DATA_FIELD_DEF_STRUCT(TCubeTiling, cubeTilingData);  // 引用高阶API的tiling结构体
  TILING_DATA_FIELD_DEF(uint32_t, field);  // 结构成员的引用结构体
END_TILING_DATA_DEF;
REGISTER_TILING_DATA_CLASS(MyAdd, MyAddTilingData)  // tiling结构注册给算子
}
```

2. 通过高阶API配套的tiling函数对tiling结构初始化：

```
static ge::graphStatus TilingFunc(gert::TilingContext* context) {
    int32_t M = 1024;
    int32_t N = 640;
    int32_t K = 256;
    int32_t baseM = 128;
    int32_t baseN = 128;
    auto ascendcPlatform = platform_ascendc::PlatformAscendC(context->GetPlatformInfo());
    MultiCoreMatmulTiling cubeTiling(ascendcPlatform);
    cubeTiling.SetDim(2);
    cubeTiling.SetAType(TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT16);
    cubeTiling.SetBType(TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT16);
    cubeTiling.SetCType(TPosition::LCM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT);
    cubeTiling.SetBiasType(TPosition::GM, CubeFormat::ND, matmul_tiling::DataType::DT_FLOAT);
    cubeTiling.SetShape(M, N, K);
    cubeTiling.SetOrgShape(M, N, K);
    cubeTiling.SetFixSplit(baseM, baseN, -1);
    cubeTiling.SetBias(true);
    cubeTiling.SetBufferSpace(-1, -1, -1);
    MyAddTilingData tiling;
```



```
    if (cubeTiling.GetTiling(tiling.cubeTilingData) == -1){
        return ge::GRAPH_FAILED;
    }
    // some code
}
```

9.4 编译部署

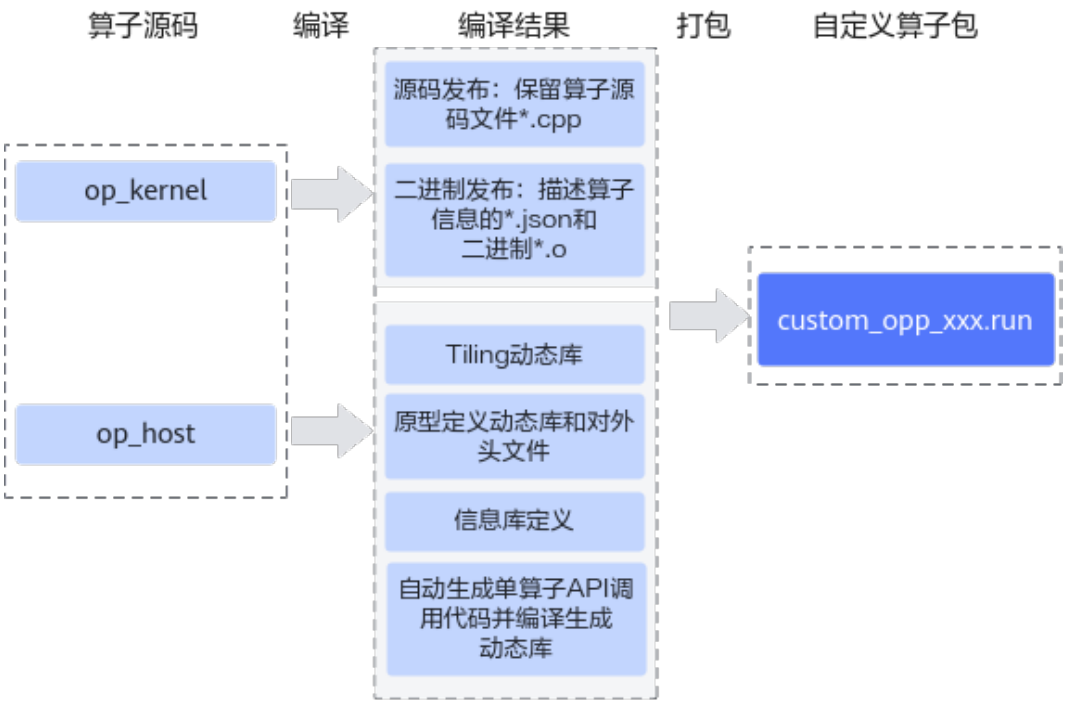
9.4.1 算子工程编译

算子kernel侧和host侧实现开发完成后，需要对算子工程进行编译，生成自定义算子安装包*.run，详细的编译操作包括：

- 编译Ascend C算子kernel侧代码实现文件*.cpp，分为源码发布和二进制发布两种方式。
  - **源码发布：**不对算子kernel侧实现进行编译，保留算子kernel源码文件*.cpp。该方式可以支持算子的在线编译、通过ATC模型转换的方式编译算子的场景。
  - **二进制发布：**对算子kernel侧实现进行编译，生成描述算子相关信息的json文件*.json和算子二进制文件*.o。算子调用时，如果需要直接调用算子二进制，则使用该编译方式，比如通过**9.5 单算子API调用**的方式完成单算子的调用，**12.2 PyTorch框架**中单算子调用的场景，动态网络中调用算子的场景。
- 编译Ascend C算子host侧代码实现文件*.cpp、*.h。
  - 将原型定义和shape推导实现编译成算子原型定义动态库libcust_opsproto_*.so，并生成算子原型对外接口op_proto.h。
  - 将算子信息库定义编译成信息库定义文件*.json。
  - 将Tiling实现编译成Tiling动态库liboptiling.so等。
  - 自动生成单算子API调用代码和头文件aclnn_*.h，并编译生成单算子API调用的动态库libcust_opapi.so。

上述编译过程示意图如下：

图 9-3 算子工程编译示意图



编译步骤

步骤1 完成工程编译相关配置。

- 修改工程目录下的**CMakePresets.json** cacheVariables的配置项。
  CMakePresets.json文件内容如下：

```
{
    "version": 1,
    "cmakeMinimumRequired": {
        "major": 3,
        "minor": 19,
        "patch": 0
    },
    "configurePresets": [
        {
            "name": "default",
            "displayName": "Default Config",
            "description": "Default build using Unix Makefiles generator",
            "generator": "Unix Makefiles",
            "binaryDir": "${sourceDir}/build_out",
            "cacheVariables": {
                "CMAKE_BUILD_TYPE": {
                    "type": "STRING",
                    "value": "Release"
                },
                "ENABLE_SOURCE_PACKAGE": {
                    "type": "BOOL",
                    "value": "True"
                },
                "ENABLE_BINARY_PACKAGE": {
                    "type": "BOOL",
                    "value": "True"
                },
                "ASCEND_COMPUTE_UNIT": {
                    "type": "STRING",
                    "value": "ascendxxx"
                },
```

```
            "ENABLE_TEST": {
                "type": "BOOL",
                "value": "True"
            },
            "vendor_name": {
                "type": "STRING",
                "value": "customize"
            },
            "ASCEND_CANN_PACKAGE_PATH": {
                "type": "PATH",
                "value": "/usr/local/Ascend/latest"
            },
            "ASCEND_PYTHON_EXECUTABLE": {
                "type": "STRING",
                "value": "python3"
            },
            "CMAKE_INSTALL_PREFIX": {
                "type": "PATH",
                "value": "${sourceDir}/build_out"
            },
            "ENABLE_CROSS_COMPILE": {
                "type": "BOOL",
                "value": "False"
            },
            "CMAKE_CROSS_PLATFORM_COMPILER": {
                "type": "PATH",
                "value": "/usr/bin/aarch64-linux-gnu-g++"
            }
        }
    }
  ]
}
```

表 9-3 需要开发者配置的常用参数列表

参数名称	参数描述	默认值
ASCEND_CANN_PACKAGE_PATH	CANN软件的安装目录，请根据实际情况进行修改，例如：/usr/local/Ascend/ascend-toolkit/latest。	“/usr/local/Ascend/latest”
CMAKE_BUILD_TYPE	编译模式选项，可配置为： – “Release”，Release版本，不包含调试信息，编译最终发布的版本。 – “Debug”，“Debug”版本，包含调试信息，便于开发者开发和调试。	“Release”
ENABLE_SOURCE_PACKAGE	是否开启源码编译	“True”
ENABLE_BINARY_PACKAGE	是否开启二进制编译	“True”
vendor_name	标识自定义算子所属厂商的名称。建议开发者自行指定所属厂商名称，避免和其他厂商提供的算子包冲突。	“customize”

- 配置编译相关环境变量（可选）

  表 9-4 环境变量说明

环境变量	配置说明

CMAKE_CXX_COMPILER_LAUNCHER	用于配置C++语言编译器（如g++）、毕昇编译器的启动器程序为ccache，配置后即可开启cache缓存编译，加速重复编译并提高构建效率。用法如下，在对应的CMakeLists.txt进行设置：
	`set(CMAKE_CXX_COMPILER_LAUNCHER <launcher_program>)`
	其中<launcher_program>是ccache的安装路径，比如ccache的安装路径为/usr/bin/ccache，示例如下：
	`set(CMAKE_CXX_COMPILER_LAUNCHER /usr/bin/ccache)`

步骤2 在算子工程目录下执行如下命令，进行算子工程编译。

```
./build.sh
```

编译成功后，会在当前目录下创建build_out目录，并在build_out目录下生成自定义算子安装包**custom_opp**_*<target os>*_*<target architecture>*.**run**。

用户如果需要编译过程日志存盘，可以使用环境变量ASCENDC_BUILD_LOG_DIR来控制存储路径。用户设置该选项之后，如果编译过程中无错误产生，则对应的log文件后缀会添加"_success"，若编译过程有错误产生，则会在屏幕打印对应的报错信息，以及指示用户log文件的具体路径与文件名，同时，对应log文件后缀会添加“_error”。

```
# 如希望编译日志存储在/home/build_log/，则可以按照如下设置，默认不打开日志存储
export ASCENDC_BUILD_LOG_DIR=/home/build_log/
```

----结束

算子包交叉编译

完成算子代码实现后，如果当前平台架构和运行环境一致则参考上一节的内容进行编译即可，如果需要实现算子包的交叉编译，您可以参考如下流程。

步骤1 交叉编译工具下载，下表以Ubuntu系列操作系统为例，展示了编译工具下载命令的样例。其他操作系统，请替换为实际的下载命令。

表 9-5 Ubuntu 系列操作系统交叉编译工具下载命令样例

当前平台架构	运行环境平台架构	编译工具下载命令
x86_64	aarch64	sudo apt-get install -y g++-aarch64-linux-gnu
aarch64	x86_64	sudo apt-get install g++-x86-64-linux-gnu

步骤2 自定义算子工程交叉编译，构建生成自定义算子包。

须知

1. 完成交叉编译前，需要修改CMakePresets.json中**ASCEND_CANN_PACKAGE_PATH**为CANN软件包安装路径。
2. 二进制包编译不支持交叉编译，需要修改CMakePresets.json中**ENABLE_BINARY_PACKAGE**为False。

1. 修改CMakePresets.json中**ENABLE_CROSS_COMPILE**为True，使能交叉编译。
   ```
   "ENABLE_CROSS_COMPILE": {
       "type": "BOOL",
       "value": "True"
    }
   ```
2. 修改CMakePresets.json中**CMAKE_CROSS_PLATFORM_COMPILER**为安装后的交叉编译工具路径。
   ```
   "CMAKE_CROSS_PLATFORM_COMPILER": {
       "type": "PATH",
       "value": "/usr/bin/aarch64-linux-gnu-g++"
   }
   ```
3. 在算子工程目录下执行如下命令，进行算子工程交叉编译。

   ./build.sh

   编译成功后，会在当前目录下创建build_out目录，并在build_out目录下生成自定义算子安装包**custom_opp_***<target os>*_*<target architecture>*.**run**

----结束

支持自定义编译选项

在算子工程中，如果开发者想对算子kernel侧代码增加一些自定义的编译选项，可以参考如下内容进行编译选项的定制。

修改算子工程op_kernel目录下的CMakeLists.txt，使用add_ops_compile_options来增加编译选项，方法如下：

```
add_ops_compile_options(OpType COMPUTE_UNIT soc_version1 soc_version2 ... OPTIONS option1 option2 ...)
```

具体参数的介绍如下：

表 9-6 具体参数介绍

参数名称	可选/必选	参数描述
OpType（算子类型）	必选	第一个参数应传入算子类型，如果需要对算子工程中的所有算子生效，需要配置为ALL。

参数名称	可选/必选	参数描述
COMPUTE_UNIT	可选	标识编译选项在哪些AI处理器型号上生效，多个型号之间通过空格间隔。不配置时表示对所有AI处理器型号生效。 **说明** COMPUTE_UNIT具体配置如下： • 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

OPTIONS	必选	自定义的编译选项。多个编译选项之间通过空格间隔。
		• 增加-D编译选项，用于在编译时定义宏。
		`OPTIONS -Dname=definition`
		• 增加-g -O0等调试用编译选项。
		• 增加Ascend C框架提供的调试用编译选项-DASCENDC_DUMP、-DASCENDC_DEBUG
		– -DASCENDC_DUMP用于控制Dump开关，默认开关打开，开发者调用printf/DumpTensor/assert后会有信息打印；设置为0后，表示开关关闭。
		`OPTIONS -DASCENDC_DUMP=0`
		– -DASCENDC_DEBUG用于控制Ascend C API的调测开关，默认开关关闭；增加该编译选项后，表示开关打开，此时接口内部的assert校验生效，校验不通过会有assert日志打屏。开启该功能会对算子实际运行的性能带来一定影响，通常在调测阶段使用。
		`OPTIONS -DASCENDC_DEBUG`
		当前-DASCENDC_DEBUG功能支持的产品型号为：
		Atlas 推理系列产品
		Atlas A2训练系列产品/Atlas 800I A2推理产品
		– --tiling_key，设置该选项后，只编译指定的**TilingKey**相关的kernel代码，用于加速编译过程。若不指定TilingKey编译，则默认编译所有的TilingKey。配置多个TilingKey时，TilingKey之间不能有空格。示例如下，其中1、2为tiling_key。
		`--tiling_key=1,2`

须知

- 编译选项是基于“算子类型+AI处理器型号系列”进行配置的，也就是说不同的“算子类型+AI处理器型号系列”可以配置不同的编译选项。
  ```
  add_ops_compile_options(AddCustom COMPUTE_UNIT Ascendxxxyy ... OPTIONS -DNEW_MACRO1=xx)
  add_ops_compile_options(AddCustom COMPUTE_UNIT Ascendxxxyy ... OPTIONS -DNEW_MACRO2=xx)
  add_ops_compile_options(AddCustom COMPUTE_UNIT Ascendxxxyy ... OPTIONS -DNEW_MACRO3=xx)
  ```
- 对相同算子类型+AI处理器型号系列，做多次编译选项配置，以后配置的为准。
- 对ALL生效的编译选项和对单一算子生效的编译选项如果没有冲突，同时生效，如果有冲突，以单一算子的编译选项为准。

9.4.2 算子包部署

算子包部署指执行自定义算子包的安装，算子工程的编译结果会自动部署到算子包安装目录下。

算子包部署

步骤1 自定义算子安装包部署。

在自定义算子包所在路径下，执行如下命令，安装自定义算子包。

./custom_opp_*<target os>*_*<target architecture>***.run --install-path=***<path>*

--install-path为可选参数，用于指定自定义算子包的安装目录。支持指定绝对路径，运行用户需要对指定的安装路径有可读写权限。

下文描述中的*<vendor_name>*为算子工程编译时CMakePresets.json配置文件中字段“vendor_name”的取值，默认为“customize”。

- 默认安装场景，不配置--install-path参数，安装成功后会将编译生成的自定义算子相关文件部署到${INSTALL_DIR}/opp/vendors/*<vendor_name>*目录。${INSTALL_DIR}请替换为CANN软件安装后文件存储路径。例如，若安装的Ascend-cann-toolkit软件包，安装后文件存储路径示例为：$HOME/Ascend/ascend-toolkit/latest。

  说明

  自定义算子包默认安装路径${INSTALL_DIR}/opp/vendors的默认权限为700（仅本用户访问）。如果权限不足导致自定义算子包安装失败，可使用*--install-path*参数并配置环境变量ASCEND_CUSTOM_OPP_PATH来指定安装目录（参考**指定目录安装**）或者联系CANN软件包的安装用户修改vendors目录权限来解决。
- 指定目录安装场景，配置--install-path参数，安装成功后会将编译生成的自定义算子相关文件部署到*<path>*/vendors/*<vendor_name>*目录，并在*<path>*/vendors/*<vendor_name>*/*bin*目录下新增set_env.bash，写入当前自定义算子包相关的环境变量。

须知

如果部署算子包时通过配置--install-path参数指定了算子包的安装目录，则在使用自定义算子前，需要执行**source *<path>*/vendors/*<vendor_name>*/bin/*set_env.bash***命令，**set_env.bash**脚本中将自定义算子包的安装路径追加到环境变量ASCEND_CUSTOM_OPP_PATH中，使自定义算子在当前环境中生效。

命令执行成功后，自定义算子包中的相关文件将部署至当前环境中。

步骤2 以默认安装场景为例，可查看部署后的目录结构，如下所示：

```
├── opp    //算子库目录
│   ├── vendors    //自定义算子所在目录
│   │   ├── config.ini
│   │   └── vendor_name1   // 存储对应厂商部署的自定义算子，此名字为编译自定义算子安装包时配置的vendor_name，若未配置，默认值为customize
│   │       ├── framework    //自定义算子插件库
│   │       ├── op_api
│   │       │   ├── include
│   │       │   │   └── aclnn_xx.h    //算子调用API声明文件
│   │       │   └── lib
│   │       │       └── libcust_opapi.so
│   │       ├── op_impl
│   │       │   └── ai_core
│   │       │       └── tbe
│   │       │           ├── config
│   │       │           │   └── ${soc_version}    //昇腾AI处理器类型
│   │       │           │       └── aic-${soc_version}-ops-info.json    //自定义算子信息库文件
│   │       │           ├── vendor_name1_impl    //自定义算子实现代码文件
│   │       │           │   └── dynamic
│   │       │           │       ├── xx.cpp
│   │       │           │       └── xx.py
│   │       │           ├── kernel    //自定义算子二进制文件
│   │       │           │   └── ${soc_version}    //昇腾AI处理器类型
│   │       │           │       └── config
│   │       │           └── op_tiling
│   │       │               ├── lib
│   │       │               └── liboptiling.so
│   │       └── op_proto    //自定义算子原型库所在目录
│   │           ├── inc
│   │           │   └── op_proto.h
│   │           └── lib
│   ├── vendor_name2   // 存储厂商vendor_name2部署的自定义算子
```

步骤3 配置自定义算子优先级。

多算子包共存的情况下，若不同的算子包目录下存在相同OpType的自定义算子，则以优先级高的算子包目录下的算子为准。下面介绍如何配置算子包优先级：

- 默认安装场景

  当“opp/vendors”目录下存在多个厂商的自定义算子时，您可通过配置“opp/vendors”目录下的“config.ini”文件，配置自定义算子包的优先级。

  “config.ini”文件的配置示例如下：

  ```
  load_priority=vendor_name1,vendor_name2,vendor_name3
  ```

  - “load_priority”：优先级配置序列的关键字，不允许修改。
  - “*vendor_name*1,*vendor_name*2,*vendor_name*3”：自定义算子厂商的优先级序列，按照优先级从高到低的顺序进行排列。

- 指定目录安装场景

  指定目录安装场景下，如果需要多个自定义算子包同时生效，分别执行各算子包安装路径下的**set_env.bash**脚本即可。每次脚本执行都会将当前算子包的安装路

径追加到ASCEND_CUSTOM_OPP_PATH环境变量的最前面。因此可以按照脚本执行顺序确定优先级：脚本执行顺序越靠后，算子包优先级越高。

比如先执行**source *<path>/vendor_name1/bin/set_env.bash***，后执行**source *<path>/vendor_name2/bin/set_env.bash***，vendor_name2算子包的优先级高于vendor_name1。ASCEND_CUSTOM_OPP_PATH示例如下：

```
ASCEND_CUSTOM_OPP_PATH=<path>/vendor_name2:<path>/vendor_name1:
```

- 指定目录安装场景下安装的算子包优先级高于默认方式安装的算子包。

----结束

多平台算子包部署

支持安装多平台的自定义算子包，安装时同样支持默认路径和自定义路径安装。

以默认路径安装为例，在aarch64平台上分别安装aarch64平台和x86_64平台算子包，安装成功后可查看目录结构兼容两种平台类型，如下所示：

```
├── opp    //算子库目录
│   ├── vendors    //自定义算子所在目录
│   │   ├── config.ini
│   │   └── vendor_name1    // 存储对应厂商部署的自定义算子，此名字为编译自定义算子安装包时配置的vendor_name，若未配置，默认值为customize
│   │       ├── framework    //自定义算子插件库
│   │       ├── op_impl
│   │       │   └── ai_core
│   │       │       └── tbe
│   │       │           ├── config
│   │       │           │   └── ${soc_version}
│   │       │           │       └── aic-${soc_version}-ops-info.json    //自定义算子信息库文件
│   │       │           ├── vendor_name1_impl    //自定义算子实现代码文件
│   │       │           │   └── dynamic
│   │       │           │       ├── xx.cpp
│   │       │           │       └── xx.py
│   │       │           ├── kernel    //自定义算子二进制文件
│   │       │           │   └── ${soc_version}
│   │       │           │   └── config
│   │       │           └── op_tiling
│   │       │               ├── lib
│   │       │               │   └──linux
│   │       │               │      ├──aarch64
│   │       │               │      │  └──libcust_opmaster_rt2.0.so
│   │       │               │      └──x86_64
│   │       │               │         └──libcust_opmaster_rt2.0.so
│   │       │               └── liboptiling.so  -> lib/linux/aarch64/libcust_opmaster_rt2.0.so
│   │       └── op_proto    //自定义算子原型库所在目录
│   │           ├── inc
│   │           │   └── op_proto.h
│   │           └── lib
│   │               └──linux
│   │                  ├──aarch64
│   │                  │  └──libcust_opsproto_rt2.0.so
│   │                  └──x86_64
│   │                     └──libcust_opsproto_rt2.0.so
│   ├── vendor_name2    // 存储厂商vendor_name2部署的自定义算子
```

9.5 单算子 API 调用

单算子API调用方式，是指直接调用单算子API接口，基于C语言的API执行算子。算子工程创建完成后，基于工程代码框架完成算子原型定义、kernel侧算子实现、host侧tiling实现，通过工程编译脚本完成算子的编译部署，之后再进行单算子API的调用。

基本原理

完成自定义算子编译后，会自动生成单算子API，可以直接在应用程序中调用。

单算子API的形式一般定义为“两段式接口”，形如：

```
aclnnStatus aclnnXxxGetWorkspaceSize(const aclTensor *src, ..., aclTensor *out, uint64_t *workspaceSize, aclOpExecutor **executor);
aclnnStatus aclnnXxx(void *workspace, uint64_t workspaceSize, aclOpExecutor *executor, aclrtStream stream);
```

其中aclnn*Xxx*GetWorkspaceSize为第一段接口，主要用于计算本次API调用计算过程中需要多少的workspace内存。获取到本次API计算需要的workspace大小后，按照workspaceSize大小申请Device侧内存，然后调用第二段接口aclnn*Xxx*执行计算。*Xxx*代表**算子原型注册**时传入的算子类型。

aclnn*Xxx*GetWorkspaceSize接口的输入输出参数生成规则如下：

- 可选输入的命名增加Optional后缀。如下样例中，x是可选输入。
  ```
  aclnnStatus aclnnXxxGetWorkspaceSize(const aclTensor *xOptional, ..., aclTensor *out, uint64_t *workspaceSize, aclOpExecutor **executor);
  ```
- 输入输出同名、使用同一个Tensor承载的情况下，生成的aclnn接口中只保留input参数同时去掉input的const修饰，并以Ref作为后缀。如下样例中，原型定义input、output都定义为x，xRef既作为输入，又作为输出。
  ```
  aclnnStatus aclnnXxxGetWorkspaceSize(aclTensor *xRef, ..., uint64_t *workspaceSize, aclOpExecutor **executor);
  ```
- 如果仅有一个输出，输出参数命名为out；如果存在多个输出，每个输出后面都以Out作为后缀。
  ```
  // 仅有一个输出
  aclnnStatus aclnnXxxGetWorkspaceSize(const aclTensor *src, ..., aclTensor *out, uint64_t *workspaceSize, aclOpExecutor **executor);
  // 存在多个输出
  aclnnStatus aclnnXxxGetWorkspaceSize(const aclTensor *src, ..., aclTensor *yOut, aclTensor *y1Out, ..., uint64_t *workspaceSize, aclOpExecutor **executor);
  ```

前置步骤

- 参考**9.2 创建算子工程**完成自定义算子工程的创建或者参考**14.3 简易自定义算子工程**完成简易自定义算子工程的创建。
- 参考**9.3.2 Kernel侧算子实现**完成kernel侧实现的相关准备，参考**9.3.3 Host侧tiling实现**、**9.3.1 算子原型定义**完成host侧实现相关准备。
- 对于自定义算子工程，参考**9.4.1 算子工程编译**、**9.4.2 算子包部署**完成算子的编译部署，编译部署时需要开启算子的二进制编译功能：修改算子工程中的编译配置项文件CMakePresets.json，将ENABLE_BINARY_PACKAGE设置为True。编译部署时可将算子的二进制部署到当前环境，便于后续算子的调用。
  ```
  "ENABLE_BINARY_PACKAGE": {
                  "type": "BOOL",
                  "value": "True"
              },
  ```
  算子编译部署后，会在算子包安装目录下的op_api目录生成单算子调用的头文件aclnn_*xx*.h和动态库libcust_opapi.so。

  以默认安装场景为例，单算子调用的头文件.h和动态库libcust_opapi.so所在的目录结构，如下所示：
  ```
  ├── opp    //算子库目录
  │   ├── vendors    //自定义算子所在目录
  │   │   ├── config.ini
  │   │   └── vendor_name1   // 存储对应厂商部署的自定义算子，此名字为编译自定义算子安装包时配置的vendor_name，若未配置，默认值为customize
  │   │       ├── op_api
  ```



```
│   │   ├── include
│   │   │   └── aclnn_xx.h
│   │   └── lib
│   │       └── libcust_opapi.so
...
```

- 对于简易自定义算子开发工程，参考**14.3 简易自定义算子工程**完成算子的编译。编译完成后会在如下路径生成单算子调用的头文件aclnn_*xx*.h和动态库libcust_opapi.so。其中CMAKE_INSTALL_PREFIX为开发者在cmake文件中配置的编译产物存放路径。
  - 动态库路径：${CMAKE_INSTALL_PREFIX}/op_api/lib/libcust_opapi.so
  - 头文件路径：${CMAKE_INSTALL_PREFIX}/op_api/include

准备验证代码工程

代码工程目录结构如下，您可以单击**LINK**，获取样例工程的完整样例：

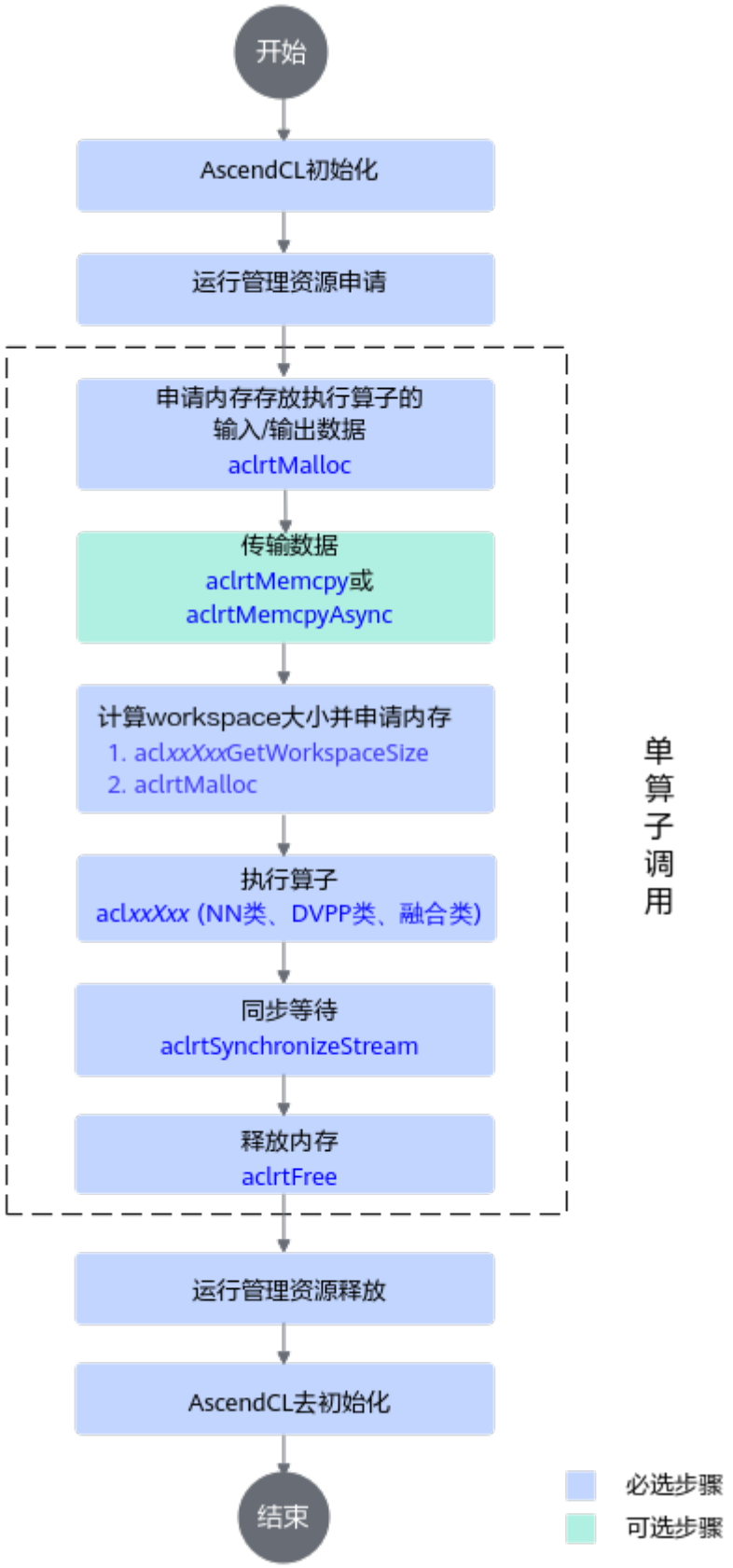
```
├──input                                    // 存放脚本生成的输入数据目录
├──output                                   // 存放算子运行输出数据和真值数据的目录
├── inc                         // 头文件目录
│   ├── common.h                    // 声明公共方法类，用于读取二进制文件
│   ├── operator_desc.h             // 算子描述声明文件，包含算子输入/输出，算子类型以及输入描述与输出描述
│   ├── op_runner.h                 // 算子运行相关信息声明文件，包含算子输入/输出个数，输入/输出大小等
├── src
│   ├── CMakeLists.txt    // 编译规则文件
│   ├── common.cpp          // 公共函数，读取二进制文件函数的实现文件
│   ├── main.cpp    // 单算子调用应用的入口
│   ├── operator_desc.cpp     // 构造算子的输入与输出描述
│   ├── op_runner.cpp   // 单算子调用主体流程实现文件
├── scripts
│   ├── verify_result.py    // 真值对比文件
│   ├── gen_data.py    // 输入数据和真值数据生成脚本文件
│   ├── acl.json    // acl配置文件
```

下文将重点介绍和单算子调用流程相关的main.cpp、op_runner.cpp文件、CMakeLists.txt文件如何编写，其他文件请自行参考。

单算子调用流程

单算子API执行流程如下：

图 9-4 单算子 API 执行接口调用流程



本节以**AddCustom自定义算子**调用为例，介绍如何编写算子调用的代码逻辑。其他算子的调用逻辑与Add算子大致一样，请根据实际情况自行修改代码。

以下是关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考，调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。

说明

因为单算子API执行方式，会自动在编译工程的build_out/autogen目录下生成.cpp和.h，编写单算子的调用代码时，要包含自动生成的单算子API执行接口头文件。示例如下：

```
#include "aclnn_add_custom.h"
```

```
// 1.AscendCL初始化
aclRet = aclInit("../scripts/acl.json");

// 2.运行管理资源申请
int deviceId = 0;
aclRet = aclrtSetDevice(deviceid);
// 获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）
aclrtRunMode runMode;
bool g_isDevice = false;
aclError aclRet = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);

// 3.申请内存存放算子的输入输出
// ......

// 4.传输数据
if (aclrtMemcpy(devInputs_[i], size, hostInputs_[i], size, kind) != ACL_SUCCESS) {
    return false;
}

// 5.计算workspace大小并申请内存
size_t workspaceSize = 0;
aclOpExecutor *handle = nullptr;
auto ret = aclnnAddCustomGetWorkspaceSize(inputTensor_[0], inputTensor_[1], outputTensor_[0],
                                          &workspaceSize, &handle);
// ...
void *workspace = nullptr;
if (workspaceSize != 0) {
    if (aclrtMalloc(&workspace, workspaceSize, ACL_MEM_MALLOC_HUGE_FIRST) != ACL_SUCCESS) {
        ERROR_LOG("Malloc device memory failed");
    }
}

// 6.执行算子
if (aclnnAddCustom(workspace, workspaceSize, handle, stream) != ACL_SUCCESS) {
    (void)aclrtDestroyStream(stream);
    ERROR_LOG("Execute Operator failed. error code is %d", static_cast<int32_t>(ret));
    return false;
}

// 7.同步等待
aclrtSynchronizeStream(stream);

// 8.处理执行算子后的输出数据，例如在屏幕上显示、写入文件等，由用户根据实际情况自行实现
// ......

// 9.释放运行管理资源
aclRet = aclrtResetDevice(deviceid);
// ....

// 10.AscendCL去初始化
aclRet = aclFinalize();
```


CMakeLists 文件

算子编译后，会生成单算子调用的头文件aclnn_*xx*.h和动态库libcust_opapi.so。具体路径请参考**前置步骤**。

编译算子调用程序时，需要在头文件的搜索路径include_directories中增加单算子调用的头文件目录，便于找到该头文件；同时需要链接cust_opapi动态库并在库文件的搜索路径link_directories中增加libcust_opapi.so所在目录。

- 在头文件的搜索路径include_directories中增加单算子调用的头文件目录。以下样例仅为参考，请根据头文件的实际目录位置进行设置。
```
include_directories(
    ${INC_PATH}/runtime/include
    ${INC_PATH}/atc/include
    ../inc
    ${OP_API_PATH}/include
)
```
- 链接cust_opapi链接库。
```
target_link_libraries(execute_add_op
    ascendcl
    cust_opapi
    acl_op_compiler
    nnopbase
    stdc++
)
```
- 在库文件的搜索路径link_directories中增加libcust_opapi.so所在目录。以下样例仅为参考，请根据库文件的实际目录位置进行设置。
```
link_directories(
    ${LIB_PATH}
    ${LIB_PATH1}
    ${OP_API_PATH}/lib
)
```

生成测试数据

在样例工程目录下，执行如下命令：

```
python3 scripts/gen_data.py
```

会在工程目录下input目录中生成两个shape为(8,2048)，数据类型为float16的数据文件input_0.bin与input_1.bin，用于进行AddCustom算子的验证。

代码样例如下：

```
import numpy as np
a = np.random.randint(100, size=(8, 2048,)).astype(np.float16)
b = np.random.randint(100, size=(8, 2048,)).astype(np.float16)
a.tofile('input_0.bin')
b.tofile('input_1.bin')
```

编译与运行

1. 开发环境上，设置环境变量，配置AscendCL单算子验证程序编译依赖的头文件与库文件路径，如下为设置环境变量的示例。${INSTALL_DIR}表示CANN软件安装目录，例如，$HOME/Ascend/ascend-toolkit/latest。*{arch-os}*为运行环境的架构和操作系统，*arch*表示操作系统架构，*os*表示操作系统，例如x86_64-linux或aarch64-linux。
```
export DDK_PATH=${INSTALL_DIR}
export NPU_HOST_LIB=${INSTALL_DIR}/{arch-os}/lib64
```
2. 编译样例工程，生成单算子验证可执行文件。

a. 切换到样例工程根目录，然后在样例工程根目录下执行如下命令创建目录用于存放编译文件，例如，创建的目录为“build”。
```
mkdir -p build
```
b. 进入build目录，执行cmake编译命令，生成编译文件

命令示例如下所示：
```
cd build
cmake ../src
```
c. 执行如下命令，生成可执行文件。
```
make
```
会在工程目录的output目录下生成可执行文件**execute_add_op**。

3. 执行单算子

a. 以运行用户（例如HwHiAiUser）拷贝开发环境中样例工程output目录下的**execute_add_op**到运行环境任一目录。

说明： 若您的开发环境即为运行环境，此拷贝操作可跳过。

b. 在运行环境中，执行**execute_add_op**文件：
```
chmod +x execute_add_op
./execute_add_op
```
会有如下屏显信息：
```
[INFO]  Set device[0] success
[INFO]  Get RunMode[1] success
[INFO]  Init resource success
[INFO]  Set input success
[INFO]  Copy input[0] success
[INFO]  Copy input[1] success
[INFO]  Create stream success
[INFO]  Execute aclnnAddCustomGetWorkspaceSize success, workspace size 0
[INFO]  Execute aclnnAddCustom success
[INFO]  Synchronize stream success
[INFO]  Copy output[0] success
[INFO]  Write output success
[INFO]  Run op success
[INFO]  Reset Device success
[INFO]  Destroy resource success
```
如果有Run op success，表明执行成功，会在output目录下生成输出文件output_z.bin。

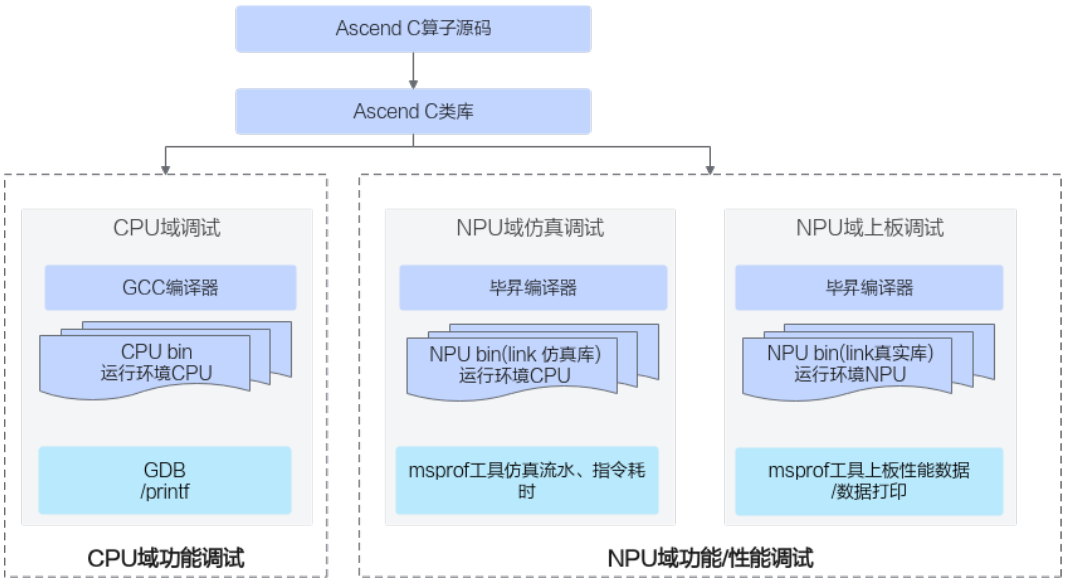
4. 比较真值文件

切换到样例工程根目录，然后执行如下命令：
```
python3 scripts/verify_result.py output/output_z.bin output/golden.bin
```
会有如下屏显信息：
```
test pass
```
可见，AddCustom算子验证结果正确。

10 算子调试调优

10.1 孪生调试简介

Ascend C提供**孪生调试**方法，相同的算子代码可以在CPU域调试精度，NPU域调试性能。孪生调试的整体方案如下：开发者通过调用Ascend C类库编写Ascend C算子kernel侧源码，kernel侧源码通过通用的GCC编译器进行编译，编译生成通用的CPU域的二进制，可以通过gdb通用调试工具等调试手段进行调试；kernel侧源码通过毕昇编译器进行编译，编译生成NPU域的二进制文件，可以通过仿真打点图或者Profiling工具进行上板数据采集等方式进行调试。



10.2 CPU 域调试

本节介绍CPU域调试的方法：gdb调试、使用printf打印命令打印。

本节提供的调试方法基于**8.2.3 运行验证算子工程**章节中的算子程序进行调试，请先完成核函数运行验证章节的学习。

说明

CPU调测过程中，配置日志相关环境变量，可以记录程序的运行过程及异常信息，有助于开发者进行功能调测。

关于环境变量的使用约束以及详细说明，可参见日志。

gdb 调试

可使用**gdb**单步调试算子计算精度。由于cpu调测已转为多进程调试，每个核都会拉起独立的子进程，故gdb需要转换成子进程调试的方式。针对Atlas 推理系列产品、Atlas 训练系列产品，每个核会拉起1个子进程。针对Atlas A2训练系列产品/Atlas 800I A2 推理产品，每个核会拉起3个子进程，1个Cube，2个Vector。

- 调试单独一个子进程

  在gdb启动后，首先设置跟踪子进程，之后再打断点，就会停留在子进程中，设置的命令为：

  ```
  set follow-fork-mode child
  ```

  但是这种方式只会停留在遇到断点的第一个子进程中，其余子进程和主进程会继续执行直到退出。涉及到核间同步的算子无法使用这种方法进行调试。

- 调试多个子进程

  如果涉及到核间同步，那么需要能同时调试多个子进程。

  在gdb启动后，首先设置调试模式为只调试一个进程，挂起其他进程。设置的命令如下：

  ```
  (gdb) set detach-on-fork off
  ```

  查看当前调试模式的命令为：

  ```
  (gdb) show detach-on-fork
  ```

  中断gdb程序要使用捕捉事件的方式，即gdb程序捕捉fork这一事件并中断。这样在每一次起子进程时就可以中断gdb程序。设置的命令为：

  ```
  (gdb) catch fork
  ```

  当执行r后，可以查看当前的进程信息：

  ```
  (gdb) info inferiors
    Num  Description
  * 1    process 19613
  ```

  可以看到，当第一次执行fork的时候，程序断在了主进程fork的位置，子进程还未生成。

  执行c后，再次查看info inferiors，可以看到此时第一个子进程已经启动。

  ```
  (gdb) info inferiors
    Num  Description
  * 1    process 19613
    2    process 19626
  ```

  这个时候可以使用切换到第二个进程，也就是第一个子进程，再打上断点进行调试，此时主进程是暂停状态：

  ```
  (gdb) inferior 2
  [Switching to inferior 2 [process 19626] ($HOME/demo)]
  (gdb) info inferiors
    Num  Description
    1    process 19613
  * 2    process 19626
  ```

  请注意，inferior后跟的数字是进程的序号，而不是进程号。

如果遇到同步阻塞，可以切换回主进程继续生成子进程，然后再切换到新的子进程进行调试，等到同步条件完成后，再切回第一个子进程继续执行。

如下是调试一个单独子进程的命令样例：

```
gdb --args add_custom_cpu
set follow-fork-mode child
break add_custom.cpp:45
run
list
backtrace
print i
break add_custom.cpp:56
continue
display xLocal
quit
```

使用 printf 打印命令打印

在代码中直接编写printf(...)来观察数值的输出。样例代码如下：

```
printf("xLocal size: %d\n", xLocal.GetSize());
printf("tileLength: %d\n", tileLength);
```

10.3 NPU 域上板调试

通过 DumpTensor、printf 打印进行调试

NPU域上板数据打印功能包括DumpTensor、printf两种，其中DumpTensor用于打印指定Tensor的数据，printf主要用于打印标量和字符串信息。

该功能仅在如下场景支持：

- 通过**Kernel直调**方式调用算子。
- 通过**单算子API调用**方式调用算子。
- 间接调用单算子API(aclnnxxx)接口：Pytorch框架单算子直调的场景。

具体的使用方法如下：

在算子kernel侧实现代码中需要输出日志信息的地方调用DumpTensor接口或者printf接口打印相关内容。

- DumpTensor示例，srcLocal表示待打印的Tensor；5表示用户的自定义附加信息，比如当前的代码行号；dataLen表示元素个数。DumpTensor接口的使用说明和具体约束请参考DumpTensor。

  ```
  DumpTensor(srcLocal,5, dataLen);
  ```

  Dump时，每个block核的dump信息前会增加对应信息头DumpHead（32字节大小），用于记录核号和资源使用信息；每次Dump的Tensor数据前也会添加信息头DumpTensorHead（32字节大小），用于记录Tensor的相关信息。打印结果的样例如下：

  ```
  DumpHead: block_id=0, total_block_num=16, block_remain_len=1048448,
  block_initial_space=1048576, magic=5aa5bccd
  DumpTensor: desc=5, addr=0, data_type=DT_FLOAT16, position=UB
  [40, 82, 60, 11, 24, 55, 52, 60, 31, 86, 53, 61, 47, 54, 34, 62, 84, 29, 48, 95, 16, 0, 20, 77, 3, 55, 69, 73,
  75, 40, 35, 13]
  DumpHead: block_id=1, total_block_num=16, block_remain_len=1048448,
  block_initial_space=1048576, magic=5aa5bccd
  DumpTensor: desc=5, addr=0, data_type=DT_FLOAT16, position=UB
  [58, 84, 22, 54, 41, 93, 1, 45, 50, 9, 72, 81, 23, 96, 86, 45, 36, 9, 36, 34, 78, 7, 2, 29, 47, 26, 13, 24, 27,
  55, 90, 5]
  ```



```
...
DumpHead: block_id=7, total_block_num=16, block_remain_len=1048448,
block_initial_space=1048576, magic=5aa5bccd
DumpTensor: desc=5, addr=0, data_type=DT_FLOAT16, position=UB
[28, 27, 79, 39, 86, 5, 23, 97, 89, 5, 65, 69, 59, 13, 49, 2, 34, 6, 52, 38, 4, 90, 11, 11, 61, 50, 71, 98, 19,
54, 54, 99]
```

- printf示例如下，printf接口的使用说明和具体约束请参考printf。

```
printf("fmt string %d", 0x123);
```

说明

DumpTensor、printf接口打印功能会对算子实际运行的性能带来一定影响，通常在调测阶段使用。开发者可以按需关闭打印功能。具体方法请参考DumpTensor、printf。

NPU 模式下的性能采集与分析

本节提供的调试方法基于**8.2.3 运行验证算子工程**章节中的算子程序进行调试，请先完成核函数运行验证章节的学习。

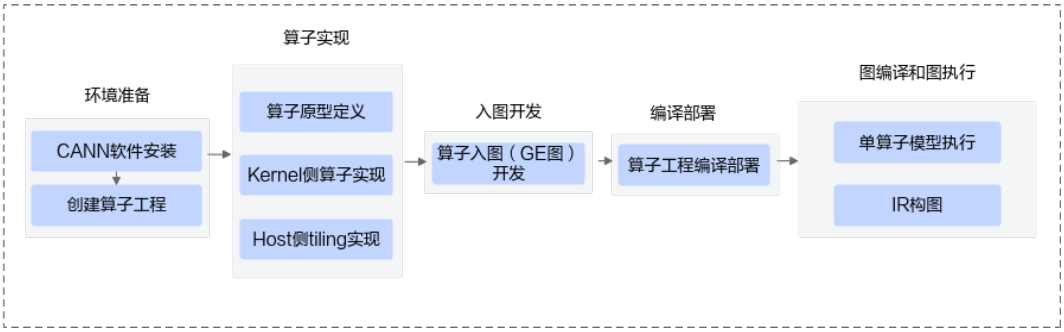
基于NPU域算子的调用接口（<<<>>>内核调用符）编写的算子程序，通过毕昇编译器编译后生成可执行程序，运行可执行程序，可以完成算子NPU域的运行验证。使用性能采集工具运行NPU模式下生成的可执行文件从而采集Ascend C算子在AI处理器上执行的性能数据，进行性能精细调优。性能分析工具的具体使用方法请参考《**性能调优工具指南**》。

11 算子入图（GE 图）开发

11.1 概述

图模式是神经网络模型的一种运行模式，在图模式下用户首先将模型的计算过程构造成一张图，然后通过GE将图下发到昇腾硬件执行。相对于单个算子依次下发的方式，图模式下，GE可以通过计算图优化、多流并行、内存复用、模型下沉等技术手段，加速模型执行效率，减少模型内存占用。

算子入图的开发流程如下图所示：算子工程创建完成后，基于工程代码框架完成算子原型定义、kernel侧算子实现、host侧tiling实现并完成算子入图开发，通过工程编译脚本完成算子的编译部署，之后即可基于图IR执行算子，比如单算子模型执行或者IR构图IR构图的方式调用自定义算子。该开发流程以**9 工程化算子开发**为基础，除了需要提供**工程化算子开发**中的算子实现文件外，还需要额外交付算子入图的代码文件。



步骤1 环境准备。

1. CANN软件安装请参考**2 环境准备**。
2. **创建算子工程**。使用msOpGen工具创建算子开发工程。

步骤2 算子实现。

- **算子原型定义**。通过原型定义来描述算子输入输出、属性等信息以及算子在AI处理器上相关实现信息，并关联tiling实现等函数。

- Kernel侧算子实现和host侧tiling实现请参考**7 算子实现**；工程化算子开发，支持开发者调用Tiling API基于CANN提供的编程框架进行tiling开发，kernel侧也提供对应的接口方便开发者获取tiling参数，具体内容请参考**9.3.2 Kernel侧算子实现**和**9.3.3 Host侧tiling实现**，由此而带来的额外约束也在上述章节说明。

步骤3 **算子入图（GE图）开发**。算子入图场景下，需要提供shape推导等算子入图适配函数的实现。

步骤4 **编译部署**。通过工程编译脚本完成算子的编译部署。

步骤5 **图编译和图执行**：基于图IR执行算子，比如单算子模型执行或者IR构图IR构图的方式调用自定义算子。

----结束

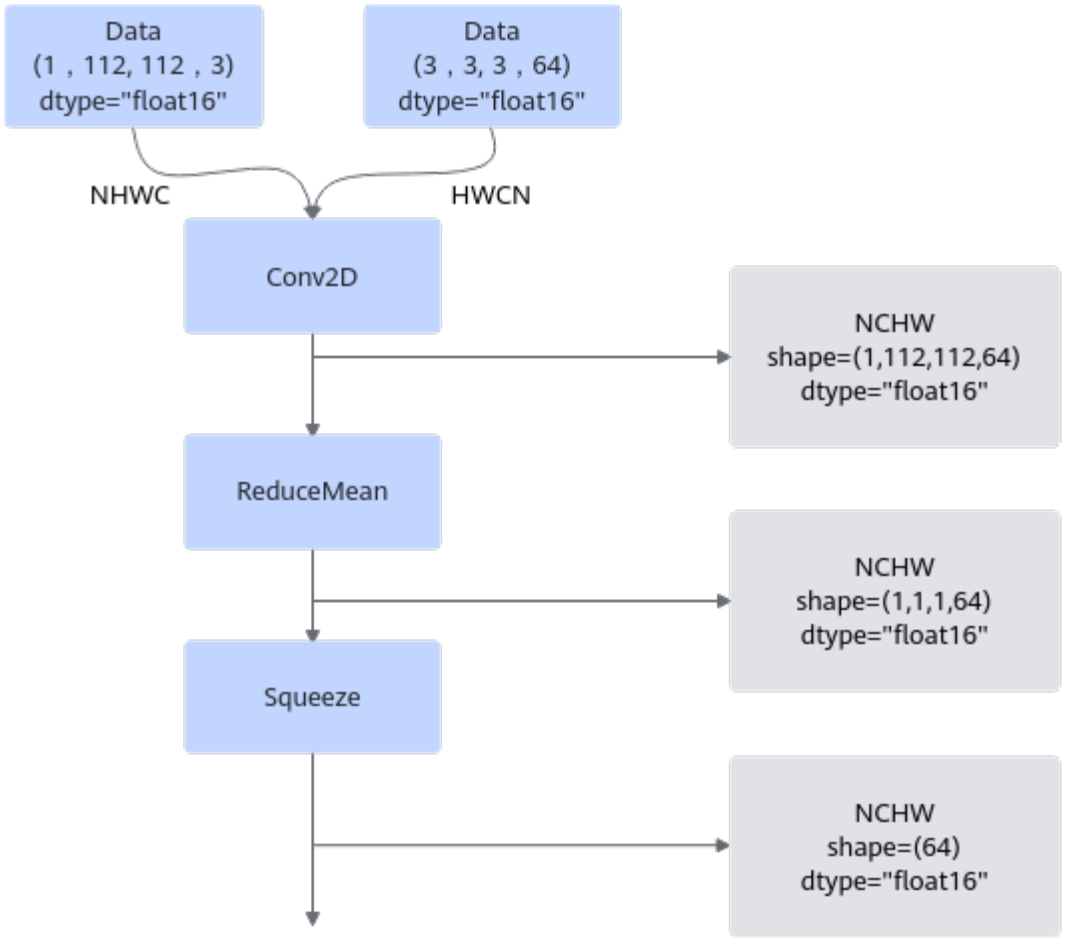
11.2 开发流程

该开发流程以**9 工程化算子开发**为基础，除了需要提供**工程化算子开发**中的算子实现文件外，还需要额外交付算子入图的代码文件。本节仅提供算子入图代码文件的开发指导。

假设下图是我们需要使用的网络模型，您可能会想直接逐个算子调用，根据输入tensor得到输出tensor就可以完成网络的运行，但在图模式场景下，实际的网络模型生成过程中，会先进行tensor shape以及datatype的推导。这样可以让我们在图执行之前，就知道各tensor的数据类型和形状，提前校验其正确性；同时提前推理出算子的输出张量描述，包括张量的形状、数据类型及数据排布格式等信息，算子构图准备阶段就可以为所有的张量静态分配内存，避免动态内存分配带来的开销。

下面的网络模型经过shape和datatype推导之后，可以得到灰色底纹框中的推导信息：

图 11-1 shape 与 datatype 推导示意图



除了tiling实现外，算子入图时需要额外提供的实现代码有以下几种：

- datatype推导：根据算子的输入datatype、算子逻辑及算子属性等信息，推理出算子的输出张量datatype。
- shape推导：根据算子的输入shape、算子逻辑及算子属性等信息，推理出算子的输出张量shape。
- 声明数据依赖：部分算子在InferShape时，需要依赖某个输入的具体值才可以进行，这类算子被称为“数据依赖算子”，对应的输入被称为“数据依赖输入”。该类算子在注册时，需要声明其数据依赖输入。

下表列出了不同类型的算子对上述实现代码的要求。

表 11-1 不同的类型的算子对入图实现代码的要求

分类	对入图实现代码的要求
根据输入shape可以推导出输出shape。	• shape推导 • datatype推导

分类	对入图实现代码的要求
依赖输入的value才能推导出输出shape，即数据依赖算子。 如Reshape算子，依赖shape输入的value才能推导出输出shape。	• shape推导 • datatype推导 • 声明数据依赖

实际开发时通过固定的datatype和shape推导原型实现推导函数，然后再通过SetInferShape、SetInferDataType接口来关联对应的shape推导函数，样例如下。

```
namespace ge {
static graphStatus InferShape(gert::InferShapeContext *context)
{
    ...
    return GRAPH_SUCCESS;
}

static graphStatus InferDataType(gert::InferDataTypeContext *context)
{
    ...
    return ge::GRAPH_SUCCESS;
}
} // namespace ge


namespace ops {
class AddCustom : public OpDef {
public:
    AddCustom(const char* name) : OpDef(name)
    {
        this->Input("x")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Input("y")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        this->Output("z")
            .ParamType(REQUIRED)
            .DataType({ge::DT_FLOAT16, ge::DT_FLOAT, ge::DT_INT32})
            .Format({ge::FORMAT_ND, ge::FORMAT_ND, ge::FORMAT_ND});
        // 根据用户的算子调用方式决定需不需要注册 图模式调用方式下需要
        this->SetInferShape(ge::InferShape);
       this->SetInferDataType(ge::InferDataType);
        this->AICore()
            .SetTiling(optiling::TilingFunc);
        // 请替换为实际的昇腾AI处理器型号
        this->AICore().AddConfig("ascendxxx");
    }
};
OP_ADD(AddCustom);
} // namespace ops
```

datatype 推导

以AddCustom算子为例，InferDataType的实现如下所示。该样例中输出tensor的数据类型与输入tensor的数据类型相同，所以直接将任意一个输入tensor的数据类型赋给输出tensor即可。

```
namespace ge {
static graphStatus InferDataType(gert::InferDataTypeContext* context)
```



```
{
    const auto inputDataType = context->GetInputDataType(0);
    context->SetOutputDataType(0, inputDataType);
    return ge::GRAPH_SUCCESS;
}
} // namespace ge
```

如下示例则给出了更灵活的datatype推导样例，当输入的数据类型为DT_INT4时，其输出的数据类型为DT_INT32。

```
ge::graphStatus InferDataTypeForFoo(gert::InferDataTypeContext* context) {

    if (context->GetInputDataType(0) == DT_INT4) {
        context->SetOutputDataType(0, DT_INT32);
    }
}
```

shape 推导

简单的shape推导逻辑可以使用Follow接口来表达，比如输出shape和输入shape相同的情况。示例如下：输出“y1”Follow输入“x1”场景，指定Follow模式为SHAPE，此时“y1”的shape将会和“x1”保持一致。

```
this->Input("x1")
    .ParamType(REQUIRED)
    .DataType({ge::DT_FLOAT, ge::DT_FLOAT})
    .Format({ge::FORMAT_ND, ge::FORMAT_ND});
this->Input("x2")
    .ParamType(REQUIRED)
    .DataType({ge::DT_FLOAT, ge::DT_FLOAT})
    .Format({ge::FORMAT_ND, ge::FORMAT_ND});
this->Output("y1")
    .ParamType(REQUIRED)
    .DataType({ge::DT_FLOAT, ge::DT_FLOAT})
    .Format({ge::FORMAT_ND, ge::FORMAT_ND})
    .Follow("x1", FollowType::SHAPE);
```

无法在原型定义中通过Follow表达的情况需要开发者编写InferShape函数，其原型是确定的，接受一个InferShapeContext作为输入，从此context上可以获取到输入、输出的shape指针等内容。输入shape为const类型，因此InferShape时，输入shape是只读、不允许修改的。InferShape成功后，返回ge::GRAPH_SUCCESS，其他返回值被认为推导失败。推导失败后，执行过程结束退出。

以ReShape算子为例，InferShape的实现如下所示。根据第1个输入（shape输入）的值，Reshape算子将第0个输入（x输入）的shape做变换，并输出到其第0个输出（y输出）上。Reshape的InferShape实现为：

```
ge::graphStatus InferShapeForReshape(InferShapeContext *context) {
const gert::Shape *x_shape = context->GetInputShape(0);        // 获取第0个输入的shape
const gert::Tensor *shape_tensor = context->GetInputTensor(1); // 获取第1个输入的tensor
gert::Shape *output_shape = context->GetOutputShape(0);
if (x_shape == nullptr || shape_tensor == nullptr || output_shape == nullptr) {
    // 防御式编程，不应该出现的场景，打印错误并返回失败
    return ge::GRAPH_FAILED;
}

auto reshape_size = static_cast<int32_t>(shape_tensor->GetShapeSize());
if (reshape_size < 1) {
    // 防御式编程，不应该出现的场景，打印错误并返回失败
    return ge::GRAPH_FAILED;
}

// 根据原型信息，Reshape的shape输入支持INT32与INT64两类，根据不同的类型进入对应的模板函数中做真正的shape变换操作
if (shape_tensor->GetDataType() == ge::DT_INT32) {
    int32_t *reshape_data = shape_tensor->GetData<int32_t>();
```



```
    return ReshapeInferShapeImpl<int32_t>(reshape_data, *x_shape, *output_shape, reshape_size);
} else {
    int64_t *reshape_data = shape_tensor->GetData<int64_t>();
    return ReshapeInferShapeImpl<int64_t>(reshape_data, *x_shape, *output_shape, reshape_size);
}
}
```

InferShapeContextpublic继承自ExtendedKernelContext，因此ExtendedKernelContext中提供的方法如获取算子type、name、属性等接口均可以在InferShapeContext实例中调用。

⚠ 注意

- InferShape推导函数和使用Follow接口去Follow shape不能混用，即不支持部分输出采用Infershape推导、部分输出采用Follow推导的情况。若用户同时使用了InferShape函数和Follow接口，以用户的InferShape函数为准，需要保证在InferShape函数中能够推导出所有的输出shape。
- 为了效率考虑，调用InferShape函数时，框架不会为输出shape做初始化，因此，在InferShape函数中，可以认为输出是**未初始化**的状态。如果在InferShape时，希望通过Append方式操作输出shape，需要先将输出shape的DimNum清零，以防止出现未定义行为。

InferShape 时获取属性、输入

在InferShape、Tiling时，可以通过context实例获取算子IR属性值，所谓IR属性，是指在IR注册时定义的属性，以TransData算子为例：

```
namespace ops {
class TransData : public OpDef {
public:
    explicit TransData(const char *name) : OpDef(name)
    {
        this->Input("src")
             ...
        this->Output("dst")
             ...
        this->Attr("src_format")
            .AttrType(REQUIRED)
            .String();
        this->Attr("dst_format")
            .AttrType(REQUIRED)
            .String();
        this->Attr("group")
            .AttrType(OPTIONAL)
           .Int(1);
        ...
    }
};
OP_ADD(TransData);
} // namespace ops
```

其原型定义中声明了src_format、dst_format、group三个属性，可以通过如下方式获取算子属性：

```
ge::graphStatus ExampleGetTransDataAttr(TilingContext *context) {
  // 获取所有属性
  const RuntimeAttrs *attrs = context->GetAttrs();
  ASSERT_NOT_NULL(attrs);

  // 按照在原型定义中的顺序，使用index获取属性，index从0开始计数
  const char *src_format = attrs->GetAttrPointer<char>(0);  // 获取src_format，src_format是第一个属性，因
```



```
此index为0
  const char *dst_format = attrs->GetAttrPointer<char>(1);  // 获取dst_format，dst_format是第二个属性，因此index为1
  const int64_t group = attrs->GetAttrPointer<int64_t>(2);  // 获取group，group是第三个属性，因此index为2

  return ge::GRAPH_SUCCESS;
}
```

通过index而不是字符串name来索引输入输出，对于带有OPTIONAL、DYNAMIC类型输入的算子，可能出现实例化后，单纯通过index无法索引到具体输入的问题，以DynamicRNNV3算子为例：

```
namespace ops {
class DynamicRNNV3 : public OpDef {
public:
    explicit DynamicRNNV3(const char *name) : OpDef(name)
    {
        this->Input("x")
            .ParamType(REQUIRED)
            ...
        this->Input("w")
            .ParamType(REQUIRED)
            ...
        this->Input("b")
            .ParamType(REQUIRED)
            ...
        this->Input("seq_length")
            .ParamType(OPTIONAL)
            ...
        this->Input("init_h")
            .ParamType(OPTIONAL)
            ...
        this->Input("init_c")
            .ParamType(OPTIONAL)
            ...
        this->Input("wci")
            .ParamType(OPTIONAL)
            ...
        this->Input("wcf")
            .ParamType(OPTIONAL)
            ...
        this->Input("mask")
            .ParamType(OPTIONAL)
            ...
        this->Input("mask")
            .ParamType(OPTIONAL)
            ...
        this->Input("project")
            .ParamType(OPTIONAL)
            ...


        ...
    }
};
OP_ADD(DynamicRNNV3);
} // namespace ops
```

由于DynamicRNNV3算子有连续的多个optional输入，这导致init_h及其后面的输入的实例化后index都是不确定的，对于这种类型的算子，可以通过GetOptionalInputShape传入原型对应的index来获取对应的输入shape等数据，以InferShape为例：

```
ge::graphStatus InferShapeForDynamicRNNV3(InferShapeContext *context) {
  // 对于前两个输入，不受到optional或dynamic的影响，可以按照常规方法获取输入shape
  auto x_shape = context->GetInputShape(0);
  auto w_shape = context->GetInputShape(1);
  if (x_shape == nullptr || w_shape == nullptr) {
    return ge::GRAPH_FAILED;
```



```
  }

  int64_t state_size = 0;
  // 在原型定义上，project是第11个输入(从0开始计数)
  constexpr int64_t kProjectInputIndex = 11;

  // 受到前面optional输入影响的，project实例化后输入的index是不确定的，通过GetOptionalInputShape来获取对应的输入shape，
  // GetOptionalInputShape的入参为原型上对应的index
  auto project_shape = context->GetOptionalInputShape(kProjectInputIndex);
  if (project_shape != nullptr) {
    if (project_shape->GetDimNum() < 2) {
      return ge::GRAPH_FAILED;
    }
    state_size = project_shape->GetDim(1);
  }
  // 更多的infershape逻辑...
  return ge::GRAPH_SUCCESS;
}
```

对于dynamic类型的输入，实例化后的输入可能是一到多个，对于此类输入，获取方式为：

```
// ir_index：此输入在原型定义中的index，从0开始计数
// relative_index：该输入实例化后的相对index，从0开始计数，例如某个DYNAMIC_INPUT实例化了3个，要取第二个，那么relatvie_index = 1
auto shape = context->GetDynamicInputShape(ir_index, relative_index);
```

本节举例的获取optional、dynamic输入的方式，在InferShape、Tiling函数中均可以调用。

数据依赖

一般来说，具备输入shape后，算子可以通过InferShape推导出输出shape。然而部分算子在InferShape时，需要依赖某个输入的具体值才可以进行，这类算子被称为“数据依赖算子”，对应的输入被称为“数据依赖输入”。以Reshape算子为例，其依据shape输入的描述，对输入的shape做调整，因此Reshape算子依赖shape输入的值。这类算子需要在原型定义时通过ValueDepend接口声明对应的输入为数据依赖输入。

```
namespace ops {
class Reshape : public OpDef {
public:
    explicit Reshape(const char *name) : OpDef(name)
    {
        ...
        this->Input("shape")
            .ParamType(REQUIRED)
            ...
            .ValueDepend(REQUIRED) // 声明 ReShape算子的shape输入为数据依赖输入
        ...
    }
};
OP_ADD(Reshape);
} // namespace ops
```

根据第1个输入（shape输入）的值，Reshape算子将第0个输入（x输入）的shape做变换，并输出到其第0个输出（y输出）上。Reshape的InferShape实现为：

```
ge::graphStatus InferShapeForReshape(InferShapeContext *context) {
const gert::Shape *x_shape = context->GetInputShape(0);        // 获取第0个输入的shape
const gert::Tensor *shape_tensor = context->GetInputTensor(1); // 获取第1个输入的tensor
gert::Shape *output_shape = context->GetOutputShape(0);
if (x_shape == nullptr || shape_tensor == nullptr || output_shape == nullptr) {
    // 防御式编程，不应该出现的场景，打印错误并返回失败
    return ge::GRAPH_FAILED;
}
```



```
auto reshape_size = static_cast<int32_t>(shape_tensor->GetShapeSize());
if (reshape_size < 1) {
    // 防御式编程，不应该出现的场景，打印错误并返回失败
    return ge::GRAPH_FAILED;
}

// 根据原型信息，Reshape的shape输入支持INT32与INT64两类，根据不同的类型进入对应的模板函数中做真正的shape变换操作
if (shape_tensor->GetDataType() == ge::DT_INT32) {
    int32_t *reshape_data = shape_tensor->GetData<int32_t>();
    return ReshapeInferShapeImpl<int32_t>(reshape_data, *x_shape, *output_shape, reshape_size);
} else {
    int64_t *reshape_data = shape_tensor->GetData<int64_t>();
    return ReshapeInferShapeImpl<int64_t>(reshape_data, *x_shape, *output_shape, reshape_size);
}
}
```

⚠ 注意

- 只有声明过数据依赖的输入，才可以在InferShape时调用GetInputTensor等获取tensor的接口获取其对应的tensor数据。若对一个未声明数据依赖的输入调用GetInputTensor等获取tensor的接口，只能在tensor中获取到正确的shape、format、datatype信息，无法获取到真实的tensor数据地址（获取到的地址为nullptr）。
- 从tensor中获取tensor_data时(GetData<int32_t>或GetData<int64_t>)，使用者需要保证获取的数据类型是正确的，否则行为是未定义的。

11.3 图编译和图执行

本节通过单算子模型执行的样例来介绍图模式下图编译和图执行流程。单算子模型执行是指基于图IR执行算子，先编译算子（例如，使用ATC工具将Ascend IR定义的单算子描述文件编译成算子om模型文件），再调用AscendCL接口加载算子模型，最后调用AscendCL接口执行算子。下文仅提供单算子模型执行的样例和基础内容讲解，详细内容请参考单算子模型执行。

环境要求

- 已参考**2 环境准备**，完成CANN驱动和软件的安装，配置CANN软件所需基本环境变量。

  安装CANN软件后，使用CANN运行用户进行编译、运行时，需要以CANN运行用户登录环境，执行**source ${install_path}/set_env.sh**命令设置环境变量。其中${install_path}为CANN软件的安装目录，例如：/usr/local/Ascend/ascend-toolkit。
- 已参考完成算子的开发和部署。

准备验证代码工程

代码工程目录结构如下，您可以单击**LINK**，获取样例工程的完整样例：

```
├──input                                  // 存放脚本生成的输入数据目录
├──output                                 // 存放算子运行输出数据和真值数据的目录
├── inc                          // 头文件目录
│   ├── common.h                 // 声明公共方法类，用于读取二进制文件
│   ├── operator_desc.h          // 算子描述声明文件，包含算子输入/输出，算子类型以及输入描述与输出描述
│   ├── op_runner.h              // 算子运行相关信息声明文件，包含算子输入/输出个数，输入/输出大小等
├── src
```



```
│   ├── CMakeLists.txt    // 编译规则文件
│   ├── common.cpp        // 公共函数，读取二进制文件函数的实现文件
│   ├── main.cpp    // 将单算子编译为om文件并加载om文件执行
│   ├── operator_desc.cpp     // 构造算子的输入与输出描述
│   ├── op_runner.cpp   // 单算子编译与运行函数实现文件
├── scripts
│   ├── verify_result.py    // 真值对比文件
│   ├── gen_data.py    // 输入数据和真值数据生成脚本文件
│   ├── acl.json    // acl配置文件
│   ├── add_custom_static_shape.json  // 算子描述文件，用于构造静态shape单算子模型文件
│   ├── add_custom_dynamic_shape.json // 算子描述文件，用于构造动态shape单算子模型文件
```

生成单算子离线模型文件

步骤1 构造静态shape单算子描述文件add_custom_static_shape.json，描述算子的输入、输出及属性等信息。

AddCustom静态shape算子的描述文件示例如下：

```
[
  {
    "op": "AddCustom",
    "input_desc": [
      {
        "name": "x",
        "param_type": "required",
        "format": "ND",
        "shape": [8, 2048],
        "type": "float16"
      },
      {
        "name": "y",
        "param_type": "required",
        "format":"ND",
        "shape": [8, 2048],
        "type": "float16"
      }
    ],
    "output_desc": [
      {
        "name": "z",
        "param_type": "required",
        "format":  "ND",
        "shape": [8, 2048],
        "type": "float16"
      }
    ]
  }
]
```

AddCustom动态shape算子的描述文件示例如下：

```
[
  {
    "op": "AddCustom",
    "input_desc": [
      {
        "name": "x",
        "param_type": "required",
        "format": "ND",
        "shape": [-1, -1],
        "shape_range": [[1,-1],[1,-1]],
        "type": "float16"
      },
      {
        "name": "y",
        "param_type": "required",
        "format":"ND",
        "shape": [-1, -1],
        "shape_range": [[1,-1],[1,-1]],
```

```
            "type": "float16"
        }
    ],
    "output_desc": [
        {
            "name": "z",
            "param_type": "required",
            "format":  "ND",
            "shape": [-1, -1],
            "shape_range": [[1,-1],[1,-1]],
            "type": "float16"
        }
    ]
  }
]
```

步骤2 使用ATC工具，将该算子描述文件编译成单算子模型文件（*.om文件）

ATC工具的命令示例如下：

```
atc --singleop=$HOME/op_verify/run/out/test_data/config/add_custom_static_shape.json --
output=op_models/ --soc_version=<soc_version>
```

关键参数解释如下（详细参数说明，请参见《**ATC工具使用指南**》。）：

- --singleop：单算子描述文件（json格式）的路径。
- --output：存放om模型文件的目录。
- --soc_version：配置为AI处理器的型号，请根据实际环境进行替换。

  说明

  AI处理器的型号请通过如下方式获取：

  - 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

以上命令执行后，会在output参数指定路径下生成*.om后缀的离线模型文件。

----结束

生成测试数据

在样例工程目录下，执行如下命令：

```
python3 scripts/gen_data.py
```

会在input目录下生成两个shape为(8,2048)，数据类型为float16的数据文件input_0.bin与input_1.bin，用于进行AddCustom算子的验证。

代码样例如下：

```
import numpy as np
a = np.random.randint(100, size=(8, 2048,)).astype(np.float16)
b = np.random.randint(100, size=(8, 2048,)).astype(np.float16)
a.tofile('input_0.bin')
b.tofile('input_1.bin')
```

编写验证代码

您可以参考如下样例编写单算子加载、执行的代码逻辑。

以下是关键步骤的代码示例，不可以直接拷贝编译运行，仅供参考，调用接口后，需增加异常处理的分支，并记录报错日志、提示日志，此处不一一列举。


```
// 1.AscendCL初始化
aclRet = aclInit("../scripts/acl.json");

// 2.运行管理资源申请
int deviceId = 0;
aclRet = aclrtSetDevice(deviceid);
// 获取软件栈的运行模式，不同运行模式影响后续的接口调用流程（例如是否进行数据传输等）
aclrtRunMode runMode;
bool g_isDevice = false;
aclError aclRet = aclrtGetRunMode(&runMode);
g_isDevice = (runMode == ACL_DEVICE);

// 3.加载单算子模型文件（*.om文件）
// 该目录相对可执行文件所在的目录，例如，编译出来的可执行文件存放在output目录下，此处就表示工程目录下的op_models目录
aclRet = aclopSetModelDir("../op_models");

// 4.设置算子的输入，申请内存，然后读取输入数据input_0.bin与input_1.bin并保存至申请的内存中
// ......

// 5.创建Stream流
aclrtStream stream = nullptr;
aclrtCreateStream(&stream)

// 6.执行算子
// opType表示算子类型名称，例如AddCustom
// numInputs表示算子输入个数，例如AddCustom算子是2个输入
// inputDesc表示算子输入tensor描述的数组，描述每个输入的format、shape、数据类型
// inputs表示算子输入tensor数据
// numOutputs表示算子输出个数，例如AddCustom算子是1个输出
// outputDesc表示算子输出tensor描述的数组，描述每个输出的format、shape、数据类型
// outputs表示算子输出tensor数据
// attr表示算子属性，如果算子没有属性，也需要调用aclopCreateAttr接口创建aclopAttr类型的数据
// stream用于维护一些异步操作的执行顺序

aclopExecuteV2(opType, numInputs, inputDesc, inputs,
               numOutputs, outputDesc, outputs, attr, nullptr);


// 7.阻塞应用运行，直到指定Stream中的所有任务都完成
aclrtSynchronizeStream(stream);

// 8.处理执行算子后的输出数据，例如在屏幕上显示、写入文件等，由用户根据实际情况自行实现，本示例会将结果写入output_z.bin文件中
// ......

// 9.释放stream流
aclrtDestroyStream(stream);

// 10.释放运行管理资源
aclRet = aclrtResetDevice(deviceid);
aclRet = aclFinalize();

// ....
```

运行和验证

1. 开发环境上，设置环境变量，配置AscendCL单算子验证程序编译依赖的头文件与库文件路径，如下为设置环境变量的示例。${INSTALL_DIR}表示CANN软件安装目录，例如，$HOME/Ascend/ascend-toolkit/latest。*{arch-os}*为运行环境的架构和操作系统，*arch*表示操作系统架构，*os*表示操作系统，例如x86_64-linux。

   ```
   export DDK_PATH=${INSTALL_DIR}
   export NPU_HOST_LIB=${INSTALL_DIR}/{arch-os}/devlib
   ```

2. 编译样例工程，生成单算子验证可执行文件。

   a. 切换到样例工程根目录，然后在样例工程根目录下执行如下命令创建目录用于存放编译文件，例如，创建的目录为“build”。

mkdir -p build

b. 进入build目录，执行cmake编译命令，生成编译文件

命令示例如下所示：

cd build

cmake ../src -DCMAKE_SKIP_RPATH=TRUE

c. 执行如下命令，生成可执行文件。

make

会在工程目录的output目录下生成可执行文件**execute_add_op**。

3. 执行单算子

a. 以运行用户（例如HwHiAiUser）拷贝开发环境中样例工程output下的**execute_add_op**到运行环境任一目录。

说明： 若您的开发环境即为运行环境，此拷贝操作可跳过。

b. 在运行环境中，执行**execute_add_op**文件，验证单算子模型文件。

chmod +x execute_add_op

./execute_add_op

会有如下屏显信息：

```
[INFO]  static op will be called
[INFO]  Set device[0] success
[INFO]  Get RunMode[1] success
[INFO]  aclopSetModelDir op model success
[INFO]  Init resource success
[INFO]  Set input success
[INFO]  Copy input[0] success
[INFO]  Copy input[1] success
[INFO]  Create stream success
[INFO]  Execute AddCustom success
[INFO]  Synchronize stream success
[INFO]  Copy output[0] success
[INFO]  Write output success
[INFO]  Run op success
[INFO]  Reset Device success
[INFO]  Destroy resource success
```

如果有Run op success，表明执行成功，会在outout目录下生成输出文件output_z.bin。

4. 比较真值文件

切换到样例工程根目录，然后执行如下命令：

```
python3 scripts/verify_result.py output/output_z.bin output/golden.bin
```

会有如下屏显信息，可见AddCustom算子验证结果正确。

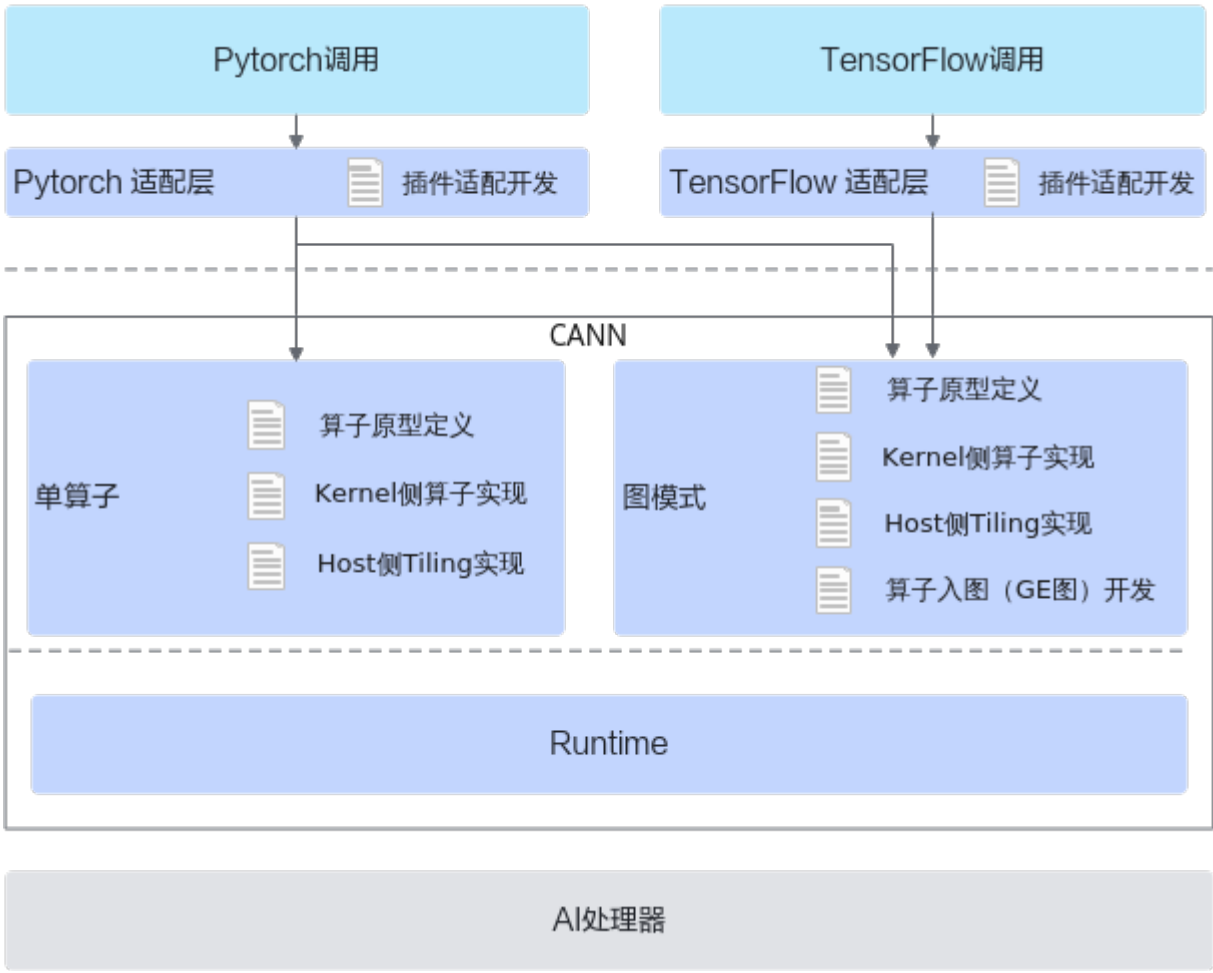
```
test pass
```


12 AI 框架算子适配

12.1 概述

本章节内容介绍AI框架调用自定义算子的方法。如下图所示，Pytorch支持单算子和图模式两种，TensorFlow仅支持图模式。

AI框架调用时，除了需要提供CANN框架调用时需要的代码实现文件，还需要进行插件适配开发。



12.2 PyTorch 框架

Pytorch框架单算子模式下的适配插件开发流程和具体样例请参见《**Ascend Extension for PyTorch 套件与三方库支持清单**》中的**昇腾自研插件 > 单算子适配OpPlugin插件开发**章节。图模式下自定义算子入图的开发指导请参见**PyTorch图模式使用指南（TorchAir）**中的**自定义算子插件化入图**章节。

12.3 ONNX 框架

12.3.1 适配插件开发

您可以参考本章节进行算子适配插件的开发，将ONNX框架的算子映射成适配昇腾AI处理器的算子（下文简称CANN算子），从而完成从ONNX框架调用Ascend C自定义算子的过程。如下样例展示了一个基础的开发流程：

```
#include "register/register.h"
#include "graph/operator.h"
#include "json.hpp"
namespace domi {
    Status ParseParamByOpFunc(const ge::Operator& op_src, ge::Operator& op_dest) {
        //...
    }
    REGISTER_CUSTOM_OP("OpType")
```

```
    .FrameworkType(ONNX)
    .OriginOpType("OriginOpType")
    .ParseParamsByOperatorFn(ParseParamByOpFunc)  //用来注册解析算子属性的函数
    .ImplyType(ImplyType::TVM);  // Ascend C算子实现类型设置为TVM
}
```

步骤1 包含所需头文件。

- register.h，存储在CANN软件安装后文件存储路径的“include/register/”目录下，包含该头文件，可使用算子注册相关类，调用算子注册相关的接口。
- operator.h（可选），存储在CANN软件安装后文件存储路径的“include/graph/”目录下，包含该头文件，可以使用Operator类相关接口，获取算子输入输出及属性等算子信息。
- json.hpp，用于进行ONNX数据定义的解析，将String类型的算子参数定义转换为json格式。若样例工程中未提供“json.hpp”文件，用户可以自行下载，并将“json.hpp”放在工程可以找到的任意路径下，然后包含此头文件即可，下载路径可参见**json.hpp**。

步骤2 使用REGISTER_CUSTOM_OP宏，完成CANN算子和ONNX框架的算子映射关系注册。使用方法如下：

- REGISTER_CUSTOM_OP：注册自定义算子，*OpType*为算子类型名称，需要与**算子原型注册**中的*OpType*保持一致。
- FrameworkType：ONNX代表原始框架为ONNX。
- OriginOpType：算子在原始框架中的类型。例如自定义算子*OpTypeA*，对应ONNX算子库版本opset_version=11，应传入“ai.onnx::11::*OpTypeA*”，当前支持的ONNX版本范围为9~15。
- ParseParamsByOperatorFn(*ParseParamByOpFunc*)：用来注册解析算子参数实现映射关系的回调函数，需要用户自定义实现回调函数ParseParamByOpFunc。具体实现方式参考**步骤3**。
- ImplyType：指定算子的实现方式。Ascend C算子实现类型设置为TVM。

步骤3 实现回调函数ParseParamByOpFunc。其函数声明如下所示：

```
Status ParseParamByOpFunc(const ge::Operator& op_src, ge::Operator& op_dest)
```

- *ParseParamByOpFunc*：函数名称，用户自定义。
- op_src：ONNX框架定义的Operator类对象，包含ONNX模型中自定义的算子属性信息，定义来源于ONNX框架的原始模型文件。
- op_dest：CANN算子数据结构，保存算子信息，Operator类的详细描述请参见Operator类。

开发者需要在回调函数中实现属性的解析和映射，具体实现方式如下：

ONNX原始模型中，属性为repeated message类型，对于repeated message类型的参数，可使用**GetAttr(const char *name, ge::AscendString &attr_value)**接口获取其属性值，然后将AscendString类型的属性值转换为String类型，再将其转换为json格式进行属性字段的解析。

实现如下所示：

```
Status ParseParamLeakyReluAscend(const ge::Operator& op_src, ge::Operator& op_dest) {
  float negative_slope = 0.01f;
  string negative_slope_str;
  AscendString attrs_string;
  // 使用固定属性名称“attribute”获取ONNX算子中的属性，并赋值给AscendString类型对象
  if (ge::GRAPH_SUCCESS == op_src.GetAttr("attribute", attrs_string)) {
    // 转换为json格式
    json attrs = json::parse(attrs_string.GetString());
```



```
    for (json attr : attrs["attribute"]) {
      if (attr["name"] == "alpha" && attr["type"] == kTypeFloat) {
        negative_slope_str = attr["f"];  // float type in json has accuracy loss, so we use string type to store it
        negative_slope = atof(negative_slope_str.c_str());
      }
    }
  }
  op_dest.SetAttr("negative_slope", negative_slope);
  return SUCCESS;
}
```

须知

- 当前版本GetAttr与SetAttr接口不支持对原始文件中数据类型为double和uint64的字段进行解析。
- 使用ATC工具执行模型转换时，对属性的获取情况不会进行强校验。所以进行算子适配插件实现时，若用户调用GetAttr失败，建议根据算子实际情况增加相应的处理逻辑，例如，针对必选属性，可返回失败，针对可选属性，可设置默认值。

----结束

12.3.2 调用样例

完成了ONNX框架的适配插件开发后，即可实现从ONNX框架调用Ascend C自定义算子。下面以一个仅包含LeakyRelu算子的ONNX框架网络为例（该网络中的LeakyRelu算子通过适配插件映射为自定义的LeakyRelu算子），呈现一个使用推理工具进行推理的过程，目的在于让您快速体验推理场景下网络中自定义算子调用的过程。

在完成如下步骤之前，您需要先参考上文内容完成自定义LeakyRelu算子kernel侧和host侧的开发、ONNX适配插件的开发，并完成算子的编译部署。

LeakyRelu算子实现的完整样例请参考**LINK**。ONNX框架调用的完整示例请参考**LINK**。

步骤1 通过如下命令获取ONNX框架网络模型。作为示例，该模型中仅包含一个LeakyRelu算子。

```
wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/AscendC/leaky_relu.onnx
```

步骤2 执行如下命令生成离线模型。（如下命令中使用的目录以及文件均为样例，请以实际为准）

```
atc --model=$HOME/module/leaky_relu.onnx --framework=5 --soc_version=<soc_version> --output=$HOME/module/out/leaky_relu --input_shape="X:8,16,1024" --input_format=ND
```

关键参数的解释如下：

- --model：ONNX框架网络模型文件（*.onnx）的路径。
- --framework：原始框架类型。5表示ONNX。
- --output：转换后的离线模型的路径以及文件名。请注意，记录保存该om模型文件的路径，后续开发应用时需要使用。
- --soc_version：昇腾AI处理器的型号。

说明

AI处理器的型号请通过如下方式获取：

- 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

- --input_shape：指定模型输入数据的shape，请基于算子支持的shape范围和实际使用场景进行设置，这里设置输入X为固定shape [8,16,1024]。
- --input_format：指定模型输入数据的格式，请基于算子支持的格式和实际使用场景进行设置，这里配置为ND。

步骤3 若提示有出现如下信息，则说明进入了Ascend C自定义算子编译流程且模型转换成功。

```
...
start compile Ascend C operator LeakyReluCustom. kernel name is leaky_relu_custom
compile Ascend C operator: LeakyReluCustom success!
...
ATC run success
```

成功执行命令后，在--output参数指定的路径下，可查看离线模型（如：leaky_relu.om）。

步骤4 准备符合模型输入要求的*.bin格式的输入数据，单击**LINK**，获取msame工具，参考该工具配套的README，使用推理工具快速完成推理体验，样例如下。

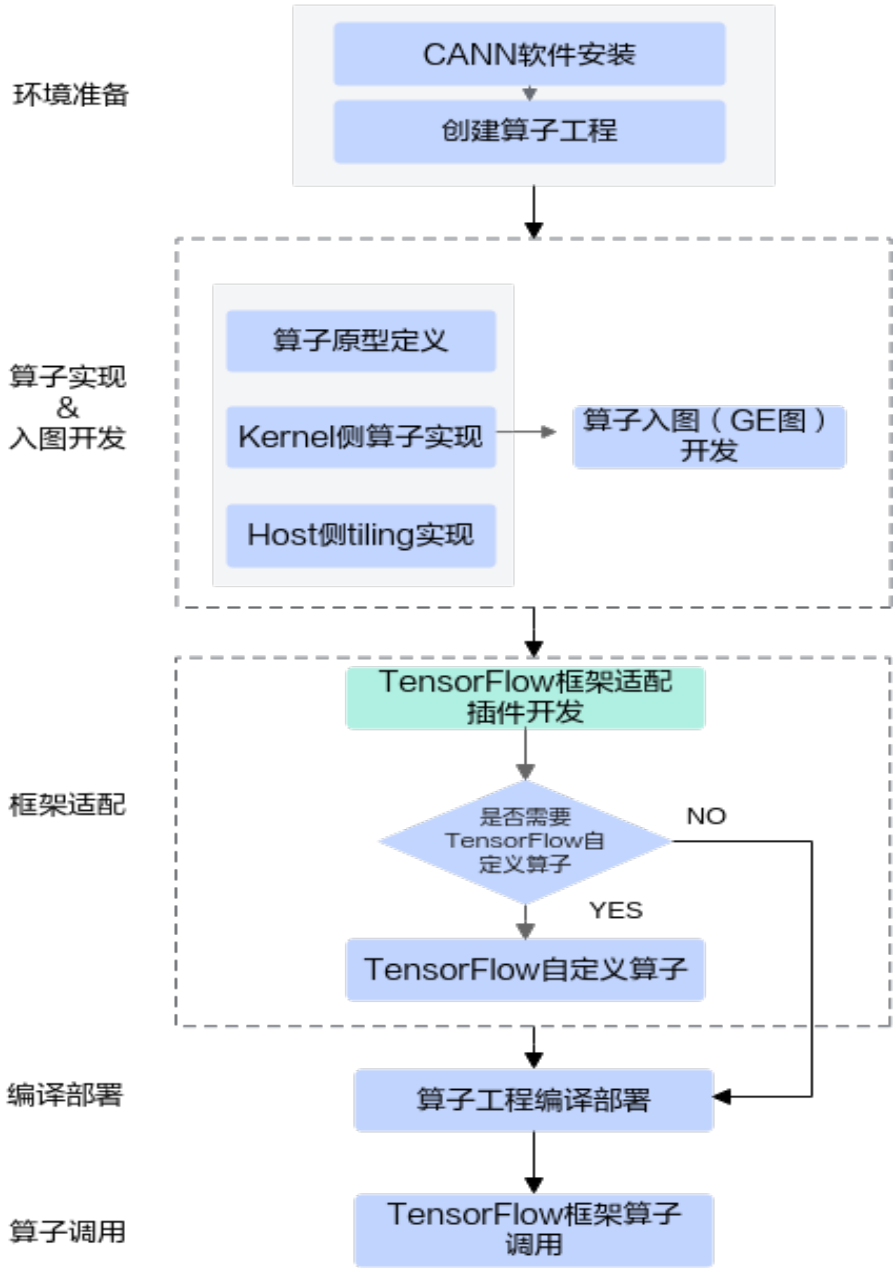
```
./msame --model "$HOME/module/out/leaky_relu.om"  --output "$HOME/module/out/" --outfmt TXT --loop 1
```

----结束

12.4 TensorFlow 框架

本章节介绍TensorFlow框架算子适配的流程，用于将TensorFlow框架的算子映射成CANN算子（开发者基于CANN框架自定义开发的算子），从而完成从TensorFlow 框架调用到CANN算子的过程。同时给出TensorFlow框架侧算子调用的示例，便于开发者了解完整流程。

下图展示了完整的开发流程图，关键步骤如下：首先开发者需要参考**9 工程化算子开发**和**11 算子入图（GE图）开发**完成算子实现和入图开发；然后进行**TensorFlow框架适配插件开发，**这是本节介绍的重点，用于将TensorFlow框架的算子映射成CANN算子；TensorFlow框架的算子包括TensorFlow自定义算子和原生算子，如果需要从TensorFlow自定义算子映射到CANN算子，还需要完成TensorFlow自定义算子的开发；最后进行TensorFlow框架侧算子调用代码编写。TensorFlow自定义算子和TensorFlow算子调用的相关内容您可以在TensorFlow的官方文档中找到更详细的介绍，本章节仅给出示例供参考。



具体步骤如下：

步骤1 环境准备。

1. CANN软件安装请参考**2 环境准备**。
2. **创建算子工程**。使用msOpGen工具创建算子开发工程。TensorFlow框架算子适配场景下，需要通过framework参数指定具体的框架为tf或者tensorflow，工具会自动生成框架适配代码。以自定义CANN算子AddCustom为例，使用msOpGen工具创建算子开发工程的具体命令如下：

```
${INSTALL_DIR}/python/site-packages/bin/msopgen gen -i $HOME/sample/add_custom.json -f tf -c ai_core-<soc_version> -lan cpp -out $HOME/sample/AddCustom
```

步骤2 算子实现。

- **算子原型定义**。通过原型定义来描述算子输入输出、属性等信息以及算子在AI处理器上相关实现信息，并关联tiling实现等函数。

- Kernel侧算子实现和host侧tiling实现请参考**7 算子实现**；工程化算子开发，支持开发者调用Tiling API基于CANN提供的编程框架进行tiling开发，kernel侧也提供对应的接口方便开发者获取tiling参数，具体内容请参考**9.3.2 Kernel侧算子实现**和**9.3.3 Host侧tiling实现**，由此而带来的额外约束也在上述章节说明。

步骤3 **算子入图（GE图）开发**。算子入图场景下，需要提供shape推导等算子入图适配函数的实现。

步骤4 TensorFlow框架适配插件开发。详细说明见**适配插件开发**。

步骤5 **编译部署**。通过工程编译脚本完成算子的编译部署。

步骤6 TensorFlow框架算子调用。详细说明见**样例（TensorFlow原生算子映射到CANN算子）**和**样例（TensorFlow自定义算子开发并映射到CANN算子）**。

本章节的完整样例请参考**LINK**。

----结束

适配插件开发

完成**算子工程创建**后，会在算子工程目录下生成framework/tf_plugin目录，用于存放TensorFlow框架适配插件实现文件。以自定义CANN算子AddCustom为例，算子工程目录如下：

```
AddCustom
├── build.sh            // 编译入口脚本
├── cmake
├── CMakeLists.txt      // 算子工程的CMakeLists.txt
├── CMakePresets.json   // 编译配置项
├── framework           // 框架适配插件实现文件目录
│   ├── tf_plugin       //  TensorFlow框架适配插件实现文件目录
│   │   ├── CMakeLists.txt
│   │   └── tensorflow_add_custom_plugin.cc  // TensorFlow框架适配插件实现文件
│   └── CMakeLists.txt
├── op_host                 // host侧实现文件
├── op_kernel               // kernel侧实现文件
└── scripts                 // 自定义算子工程打包相关脚本所在目录
```

TensorFlow框架适配插件实现代码如下：

```
#include "register/register.h"
namespace domi {
REGISTER_CUSTOM_OP("AddCustom")
    .FrameworkType(TENSORFLOW)
    .OriginOpType("AddCustom")
    .ParseParamsByOperatorFn(AutoMappingByOpFn);
}
```

- 包含插件实现函数相关的头文件。

  register.h存储在CANN软件安装后文件存储路径的“include/register/”目录下，包含该头文件，可使用算子注册相关类，调用算子注册相关的接口。

- REGISTER_CUSTOM_OP：注册自定义算子，传入算子的*OpType*，需要与算子原型注册中的*OpType*保持一致。
  - FrameworkType：TENSORFLOW代表原始框架为TensorFlow。
  - OriginOpType：算子在原始框架中的类型。对于TensorFlow自定义算子，还需要完成**TensorFlow自定义算子的开发**，这里的OriginOpType与REGISTER_OP注册算子名相同，对于TensorFlow原生算子， 即为原生算子名。
  - ParseParamsByOperatorFn：用来注册解析算子参数实现映射关系的回调函数，需要用户自定义实现回调函数ParseParamByOpFunc。原始TensorFlow

算子中参数与CANN算子中参数一一对应时，可直接使用自动映射回调函数AutoMappingByOpFn自动实现映射。

样例（TensorFlow 原生算子映射到 CANN 算子）

以自定义算子AddCustom为例，将该算子映射到TensorFlow内置算子Add上，需要先修改AddCustom自定义算子目录framework/tf_plugin下插件代码，完成算子名映射：

```
#include "register/register.h"
namespace domi {
REGISTER_CUSTOM_OP("AddCustom")   // 当前Ascend C自定义算子名
    .FrameworkType(TENSORFLOW)    // 第三方框架类型TENSORFLOW
    .OriginOpType("Add")          // 映射到TensorFlow原生算子Add
    .ParseParamsByOperatorFn(AutoMappingByOpFn);
}
```

完成算子工程的编译部署后，构造单算子的TensorFlow 1.15版本测试用例进行验证。

步骤1 编写测试用例“*tf_add*.py”。

步骤2 导入python库。

```
import logging            # Python标准库日志模块
import tensorflow as tf   # 导入TensorFlow开源库
from npu_bridge.estimator import npu_ops   # 导入TensorFlow开源库中的npu_ops模块
import numpy as np    # 导入Python的数学基础库
```

步骤3 通过config()定义昇腾AI处理器和CPU上的运行参数。

当“execute_type”为“ai_core”时，代表在昇腾AI处理器上运行单算子网络，最终会调用到Ascend C算子。

当“execute_type”为“cpu”时，代表在Host侧的CPU运行单算子网络，调用的是TensorFlow算子。

```
def config(execute_type):
    if execute_type == 'ai_core':
        session_config = tf.ConfigProto(
            allow_soft_placement=True,
            log_device_placement=False,)
        custom_op = session_config.graph_options.rewrite_options.custom_optimizers.add()
        custom_op.name = "NpuOptimizer"
        custom_op.parameter_map["enable_data_pre_proc"].b = True   # 开启数据预处理下沉到Device侧执行
        custom_op.parameter_map["mix_compile_mode"].b = True
        custom_op.parameter_map["use_off_line"].b = True     # True表示在昇腾AI处理器上执行训练

    elif execute_type == 'cpu':
        session_config = tf.ConfigProto(
            allow_soft_placement=True,
            log_device_placement=False)

    return session_config
```

步骤4 单算子网络测试用例主函数。

- 算子输入请根据算子实际输入个数及shape进行构造。
- 算子输出的计算，请根据算子逻辑调用TensorFlow相关接口进行实现。

```
设置np.allclose比较函数的公差参数。
#np.allclose比较函数的相对公差参数
atol = 0.001
#np.allclose比较函数的绝对公差参数
rtol = 0.001

def main(unused_argv):
    shape_params = (8, 2048)
    dtype_params = np.float16
```



```
    # 构造Add算子的两个输入数据,shape为shape_params，范围在[-2,2]之间的随机数
    x_data = np.random.uniform(-2, 2, size=shape_params).astype(dtype_params)
    y_data = np.random.uniform(-2, 2, size=shape_params).astype(dtype_params)
    # 分别对Add算子的两个输入数据进行占位
    x = tf.compat.v1.placeholder(dtype_params, shape=shape_params)
    y = tf.compat.v1.placeholder(dtype_params, shape=shape_params)
    # 计算算子输出
    out = tf.math.add(x, y)
    # 在Host侧CPU上运行单算子，得到期望运行结果
    with tf.compat.v1.Session(config=config('cpu')) as session:
        result_cpu = session.run(out, feed_dict={x: x_data, y: y_data})
    # 在昇腾AI处理器上运行单算子，得到实际运行结果
    with tf.compat.v1.Session(config=config('ai_core')) as session:
        result_ai_core = session.run(out, feed_dict={x: x_data, y: y_data})

    np.array(result_ai_core).astype(dtype_params)
    np.array(result_cpu).astype(dtype_params)
    print('====================================')
   # 通过np.allclose比较昇腾AI处理器上运行的实际结果和cpu上运行的期望结果，其中atol和rtol为np.allclose比较函数的相对公差参数和绝对公差参数
    cmp_result = np.allclose(result_ai_core, result_cpu, atol, rtol)
    print(cmp_result)
    print('====================================')
```

步骤5 运行单算子网络。

```
if __name__ == "__main__":
    tf.app.run()
```

----结束

样例（TensorFlow 自定义算子开发并映射到 CANN 算子）

步骤1 适配插件代码开发。以自定义算子AddCustom为例，将该算子映射到TensorFlow自定义算子AddCustom上，需要先修改CANN AddCustom自定义算子工程目录framework/tf_plugin下插件代码，完成算子名映射：

```
REGISTER_CUSTOM_OP("AddCustom")
  .FrameworkType(TENSORFLOW)
  .OriginOpType("AddCustom")
  .ParseParamsByOperatorFn(AutoMappingByOpFn);
```

步骤2 TensorFlow自定义算子的开发。本节仅给出示例说明，详细内容请参考TensorFlow官方文档。

创建TensorFlow原型注册文件custom_assign_add_custom.cc，内容如下：

```
#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/shape_inference.h"
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/common_shape_fns.h"
using namespace tensorflow;

// 通过TensorFlow提供的REGISTER_OP接口完成算子原型的注册
REGISTER_OP("AddCustom")        // TensorFlow 注册算子名
    .Input("x: T")              // 算子原型，输入参数x，类型为T
    .Input("y: T")              // 算子原型，输入参数y，类型为T
    .Output("z: T")             // 算子原型，输入参数z，类型为T
    .Attr("T: {half}")          // T类型支持范围
    .SetShapeFn(shape_inference::BroadcastBinaryOpShapeFn);  // 算子shape信息推导，BroadcastBinaryOpShapeFn为TensorFlow提供的内置函数，输出shape信息由输入shape传播推导，即输入和输出shape保持一致

// 实现一个CPU版本的kernel函数，因为Tensorflow的计算图在构建时会检查所有的算子是否有任意设备上的kernel函数（NPU Kernel无法被感知），如果没有将会报错。这里实现一个固定返回错误的CPU kernel函数：
class AddCustomOp : public OpKernel {
 public:
  explicit AddCustomOp(OpKernelConstruction* context) : OpKernel(context) {}
```



```
 void Compute(OpKernelContext* context) override {
   OP_REQUIRES_OK(context, errors::Unimplemented("AddCustomOp is not supported on CPU"));
 }
};

REGISTER_KERNEL_BUILDER(Name("AddCustom").Device(DEVICE_CPU), AddCustomOp);          // 注册AddCustom算子的CPU实现内核，该函数当前仅打印日志提示CPU不支持
```

使用如下命令对上述代码进行编译，产物为libcustom_ops.so，后续的算子调用脚本中可通过load_op_library接口加载该so为python模块，从而调用自定义算子。

```
TF_CFLAGS=( $(python3 -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get_compile_flags()))') )     // 获取TensorFlow编译选项
TF_LFLAGS=( $(python3 -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get_link_flags()))') )       // 获取TensorFlow链接选项
SOURCE_FILES=custom_assign_add_custom.cc                                                                    // 包含TensorFlow算子注册和CPU内核实现的cc文件
g++ -std=c++14 -shared $SOURCE_FILES -o ${Path}/libcustom_ops.so -fPIC ${TF_CFLAGS[@]} ${TF_LFLAGS[@]} -O2   // 编译命令，产物为libcustom_ops.so，TensorFlow即可通过load_op_library加载该so为python模块，调用自定义算子
```

步骤3 测试脚本中加载上一步骤编译好的动态库，实现自定义算子的调用。

- TensorFlow 1.15.0调用代码示例

```
import os
import tensorflow as tf
import numpy as np
from npu_bridge.npu_init import *
tf.enable_resource_variables()
#np.allclose比较函数的相对公差参数
atol = 0.001
#np.allclose比较函数的绝对公差参数
rtol = 0.001
def main(unused_argv):
    custom_op_lib = tf.load_op_library('./outputs/libcustom_ops.so')     # 加载so为python模块
    shape_params = (8, 2048)
    dtype_params = np.float16
    x_data = np.random.uniform(-2, 2, size=shape_params).astype(dtype_params)
    y_data = np.random.uniform(-2, 2, size=shape_params).astype(dtype_params)
    x = tf.compat.v1.placeholder(dtype_params, shape=shape_params)
    y = tf.compat.v1.placeholder(dtype_params, shape=shape_params)
    tf_z = tf.math.add(x, y)                                    # 调用TensorFlow原生算子
    ac_z = custom_op_lib.add_custom(x, y)                       # 调用AscendC AddCustom自定义算子
    config = tf.ConfigProto()
    custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
    custom_op.name = "NpuOptimizer"   # 配置在昇腾AI处理器上运行单算子
    config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
    config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF

    with tf.Session(config=config) as sess:
        sess.run(tf.global_variables_initializer())
        tf_golden = sess.run(tf_z, feed_dict={x: x_data, y: y_data})
    with tf.Session(config=config) as sess:
        sess.run(tf.global_variables_initializer())
        ascend_out = sess.run(ac_z, feed_dict={x: x_data, y: y_data})
    np.array(tf_golden).astype(dtype_params)
    np.array(ascend_out).astype(dtype_params)
    print('==================================')
    # 通过np.allclose比较昇腾AI处理器上运行的实际结果和使用TensorFlow原生算子运行的期望结果，其中atol和rtol为np.allclose比较函数的相对公差参数和绝对公差参数。
    cmp_result = np.allclose(tf_golden, ascend_out, atol, rtol)
    print(cmp_result)
    print('==================================')
if __name__ == "__main__":
    tf.app.run()
```

- TensorFlow 2.6.5调用代码

```
import os
import tensorflow as tf
import numpy as np
```



```
import npu_device
from npu_device.compat.v1.npu_init import *
npu_device.compat.enable_v1()
tf.compat.v1.enable_resource_variables()
#np.allclose比较函数的相对公差参数
atol = 0.001
#np.allclose比较函数的绝对公差参数
rtol = 0.001
def main(unused_argv):
    custom_op_lib = tf.load_op_library('./outputs/libcustom_ops.so')     # 加载so为python模块

    shape_params = (8, 2048)
    dtype_params = np.float16
    x_data = np.random.uniform(-2, 2, size=shape_params).astype(dtype_params)
    y_data = np.random.uniform(-2, 2, size=shape_params).astype(dtype_params)
    x = tf.compat.v1.placeholder(dtype_params, shape=shape_params)
    y = tf.compat.v1.placeholder(dtype_params, shape=shape_params)
    tf_z = tf.math.add(x, y)                                    # 调用TensorFlow原生算子
    ac_z = custom_op_lib.add_custom(x, y)                               # 调用AscendC AddCustom自定义算子

    config = tf.compat.v1.ConfigProto()
    custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
    custom_op.name = "NpuOptimizer"
    config.graph_options.rewrite_options.remapping = RewriterConfig.OFF
    config.graph_options.rewrite_options.memory_optimization = RewriterConfig.OFF

    with tf.compat.v1.Session(config=config) as sess:
        sess.run(tf.global_variables_initializer())
        tf_golden = sess.run(tf_z, feed_dict={x: x_data, y: y_data})
    with tf.compat.v1.Session(config=config) as sess:
        sess.run(tf.global_variables_initializer())
        ascend_out = sess.run(ac_z, feed_dict={x: x_data, y: y_data})
    np.array(tf_golden).astype(dtype_params)
    np.array(ascend_out).astype(dtype_params)
    print('====================================')
    # 通过np.allclose比较昇腾AI处理器上运行的实际结果和使用TensorFlow原生算子运行的期望结果，其
中atol和rtol为np.allclose比较函数的相对公差参数和绝对公差参数。
    cmp_result = np.allclose(tf_golden, ascend_out, atol, rtol)
    print(cmp_result)
    print('====================================')
if __name__ == "__main__":
    tf.app.run()
```

----结束

13 专题

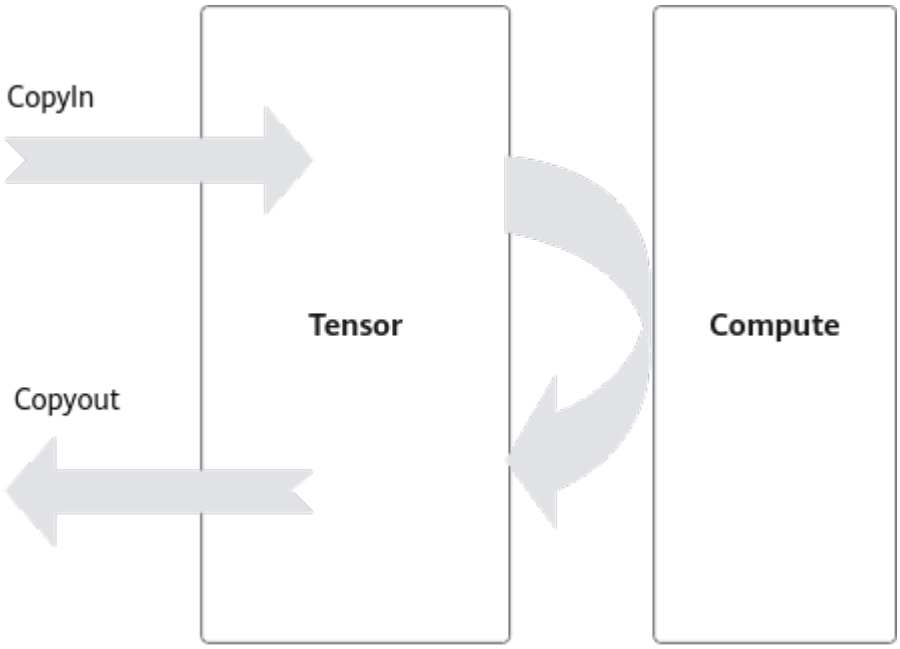
13.1 double buffer

执行于AI Core上的指令队列主要包括如下几类，即Vector指令队列（V）、Matrix指令队列（M）和存储移动指令队列（MTE2、MTE3）。不同指令队列间的相互独立性和可并行执行性，是double buffer优化机制的基石。

矢量计算CopyIn、CopyOut过程使用存储移动指令队列（MTE2、MTE3），Compute过程使用Vector指令队列（V），意味着CopyIn、CopyOut过程和Compute过程是可以并行的。

如**图13-1**所示，考虑一个完整的数据搬运和计算过程，CopyIn过程将数据从Global Memory搬运到Local Memory，Vector计算单元完成计算后，经过CopyOut过程将计算结果搬回Global Memory。

图 13-1 数据搬运与 Vector 计算过程



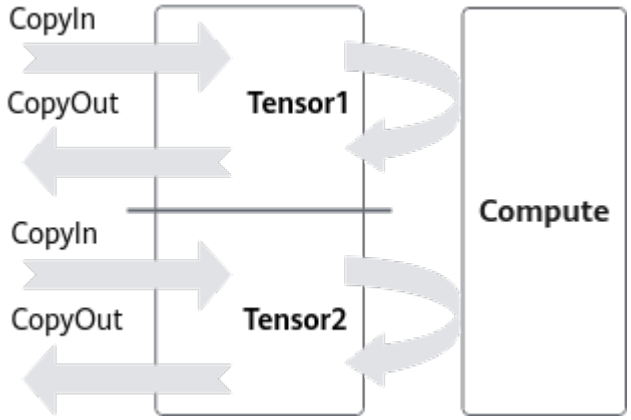
在此过程中，数据搬运与Vector计算串行执行，Vector计算单元无可避免存在资源闲置问题。举例而言，若CopyIn、Compute、CopyOut三阶段分别耗时*t*，则Vector的时间利用率仅为1/3，等待时间过长，Vector利用率严重不足。

为减少Vector等待时间，double buffer机制将待处理的数据一分为二，比如Tensor1、Tensor2。如图13-2所示，当Vector对Tensor1中数据进行Compute时，Tensor2可以执行CopyIn的过程；而当Vector切换到计算Tensor2时，Tensor1可以执行CopyOut的过程。由此，数据的进出搬运和Vector计算实现并行执行，Vector闲置问题得以有效缓解。

总体来说，double buffer是基于MTE指令队列与Vector指令队列的独立性和可并行性，通过将数据搬运与Vector计算并行执行以隐藏数据搬运时间并降低Vector指令的等待时间，最终提高Vector单元的利用效率，您可以通过为队列申请内存时设置内存块的个数来实现数据并行，简单代码示例如下：

```
pipe.InitBuffer(inQueueX, 2, 256);
```

图 13-2 double buffer 机制



需要注意：

多数情况下，采用double buffer能有效提升Vector的时间利用率，缩减算子执行时间。然而，double buffer机制缓解Vector闲置问题并不代表它总能带来整体的性能提升。例如：

- 当数据搬运时间较短，而Vector计算时间显著较长时，由于数据搬运在整个计算过程中的时间占比较低，double buffer机制带来的性能收益会偏小。
- 又如，当原始数据较小且Vector可一次性完成所有计算时，强行使用double buffer会降低Vector计算资源的利用率，最终效果可能适得其反。

因此，double buffer的性能收益需综合考虑Vector算力、数据量大小、搬运与计算时间占比等多种因素。

13.2 内存管理

13.2.1 workspace

workspace是设备侧Global Memory上的一块内存。workspace内存分为两部分：系统workspace和用户workspace。

- 系统workspace：Ascend C API需要预留的workspace内存

  API在计算过程需要一些workspace内存作为缓存，因此算子需要为API预留workspace内存，预留内存大小通过GetLibApiWorkSpaceSize接口获取。
- 用户workspace：算子实现使用到的workspace内存

  算子内部需要通过额外的device内存进行数据交换或者缓存的时候才需要分配，根据实际情况自行分配。使用场景如下：
  - 需要使用Unified Buffer和L1 Buffer上空间且空间不够用时，可以将数据暂存至workspace上。
  - 调用SyncAll等API接口时，可以使用workspace作为入参。
  - 其他需要使用Global Memory上内存空间的场景。

不同开发方式下，具体的使用方法如下：

- 工程化算子开发方式

  在tiling函数中先通过GetWorkspaceSizes接口获取workspace大小的存放位置，再设置workspace的大小，框架侧会为其在申请对应大小的设备侧Global Memory，在对应的算子kernel侧实现时可以使用这块workspace内存。在使用Matmul等需要系统workspace的高阶API时，设置的workspace空间大小为系统workspace和用户workspace之和。

```
// 用户自定义的tiling函数
static ge::graphStatus TilingFunc(gert::TilingContext* context)
{
    AddApiTiling tiling;
    ...
    size_t usrSize = 256; // 设置用户需要使用的workspace大小。
    // 如需要使用系统workspace需要调用GetLibApiWorkSpaceSize获取系统workspace的大小。
    auto ascendcPlatform = platform_ascendc:: PlatformAscendC(context->GetPlatformInfo());
    uint32_t sysWorkspaceSize = ascendcPlatform.GetLibApiWorkSpaceSize();
    size_t *currentWorkspace = context->GetWorkspaceSizes(1); // 通过框架获取workspace的指针，GetWorkspaceSizes入参为所需workspace的块数。当前限制使用一块。
    currentWorkspace[0] = usrSize + sysWorkspaceSize; // 设置总的workspace的数值大小，总的workspace空间由框架来申请并管理。
    ...
}
```

  在device侧kernel入口处的workspace为用户的workspace指针：


```
// 用户写的Kernel函数，核函数必须包括GM_ADDR workspace入参，位置需要放在tiling之前
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR
workspace, GM_ADDR tiling)
{
    ...

}
```

- Kernel直调算子开发场景

  需要使用workspace空间时，建议开启编译选项-DHAVE_WORKSPACE。host侧开发者仍需要自行申请workspace的空间，并传入。在使用Matmul等需要系统workspace的高阶API时，设置的workspace空间大小为系统workspace和用户workspace之和。系统workspce大小可以通过PlatformAscendCManager的GetLibApiWorkSpaceSize接口获取。开启-DHAVE_WORKSPACE后，开发者在kernel侧入参处获取的workspace为偏移了系统workspace后的用户workspace。

13.2.2 SPM Buffer

功能介绍

SPM Buffer功能支持对Unified Buffer的暂存操作，当Unified Buffer内存有溢出风险时，可以将Unified Buffer的数据先拷贝到SPM(Spill Memory) Buffer暂存，具体使用时再取回。

使用方法如下：

1. 调用InitSpmBuffer接口完成初始化操作。
2. 调用WriteSpmBuffer接口完成数据的拷贝暂存操作。
3. 需要使用暂存的数据时，调用ReadSpmBuffer接口完成数据的取回操作。

针对Atlas A2训练系列产品/Atlas 800I A2推理产品，仅支持暂存到workspace。

针对Atlas 训练系列产品、Atlas 推理系列产品，可以暂存到workspace或者L1 Buffer。

使用样例

1. 初始化操作。
   - 暂存到workspace初始化

     ```
     AscendC::TPipe pipe;
     int len = 1024;
     AscendC::GlobalTensor<half> workspace_gm;
     auto usrWorkspace = AscendC::GetUserWorkspace(workspace);
     // 此处的usrWorkspace为用户自定义的workspace,类型为half,有len个元素
     workspace_gm.SetGlobalBuffer((__gm__ half *)usrWorkspace, len);
     auto gm = workspace_gm[AscendC::GetBlockIdx() * len];
     pipe.InitSpmBuffer(gm, len * sizeof(half));
     ```

   - 暂存到L1 Buffer初始化

     ```
     TPipe pipe;
     int len = 1024; // 设置spm buffer为1024个类型为T的数据
     pipe.InitSpmBuffer(len * sizeof(T));
     ```

2. 暂存和取回操作。

   ```
   TQue<QuePosition::VECIN, 1> inQueueSrcVecIn;
   int dataSize = 32; // 假设T为half类型，从ub上申请一块内存32 * sizeof(half)字节
   int offset = 32; // 拷贝到spmBuffer时偏移32字节
   pipe.InitBuffer(inQueueSrcVecIn, 1, dataSize);
   LocalTensor<half> writeLocal = inQueueSrcVecIn.AllocTensor<half>();
   DataCopyParams copyParams{1, 2, 0, 0}; // 从ub上搬运一块长度为2个datablock的数据，一个
   datablock32byte
   ```



```
pipe.WriteSpmBuffer(writeLocal, copyParams, offset);
pipe.ReadSpmBuffer(writeLocal, copyParams, offset);
```

使用约束

无

13.3 通算融合算子

相比于一般的计算或搬运类算子，通算融合算子是融合集合通信任务和计算任务的算子，在算子执行过程中，计算和通信任务可以部分流水并行，以便提升性能。典型的应用场景如Matmul计算+集合通信。通算融合类算子的实现，请参考并使用Hccl高阶API。关于更多集合通信的内容和相关概念请参考HCCL概述。

通算融合算子的开发过程与一般算子相同，但请注意，当前通算融合算子暂不支持Kernel直调，暂不支持入图（GE图）开发。

算子原型定义

相比于一般算子，通算融合算子在实现**算子原型定义**时，有如下约束：

- 必须定义一个表示算子通信域名称的属性。通信域是集合通信执行的上下文，管理对应的通信实体（例如一个NPU就是一个通信实体）和通信所需的资源。
- 必须通过MC2接口注册该算子为通算融合算子，并通过HcclGroup接口配置该算子的通信域名称。

以算子属性"group"为例，"group"为该算子的通信域名称，在算子原型定义中，调用方式如下：

```
this->Attr("group").AttrType(REQUIRED).String(); // "group"为通算融合算子的属性，表示通信域名称
...
this->MC2().HcclGroup("group"); // 将"group"配置为该算子的通信域
```

调用示例

以AllGatherMatmul自定义算子为例，算子原型定义如下。

```
namespace ops {
    class AllGatherMatmulCustom : public OpDef {
    public:
        explicit AllGatherMatmulCustom(const char *name) : OpDef(name) {
            this->Input("x1")
                    .ParamType(REQUIRED)
                    .DataType({ge::DT_FLOAT16, ge::DT_BF16})
                    .Format({ge::FORMAT_ND, ge::FORMAT_ND})
                    .UnknownShapeFormat({ge::FORMAT_ND, ge::FORMAT_ND});
            this->Input("x2")
                    .ParamType(REQUIRED)
                    .DataType({ge::DT_FLOAT16, ge::DT_BF16})
                    .Format({ge::FORMAT_ND, ge::FORMAT_ND})
                    .UnknownShapeFormat({ge::FORMAT_ND, ge::FORMAT_ND})
                    .IgnoreContiguous();
            this->Input("bias")
                    .ParamType(OPTIONAL)
                    .DataType({ge::DT_FLOAT16, ge::DT_BF16})
                    .Format({ge::FORMAT_ND, ge::FORMAT_ND})
                    .UnknownShapeFormat({ge::FORMAT_ND, ge::FORMAT_ND});

            this->Output("y")
                    .ParamType(REQUIRED)
```



```
                .DataType({ge::DT_FLOAT16, ge::DT_BF16})
                .Format({ge::FORMAT_ND, ge::FORMAT_ND})
                .UnknownShapeFormat({ge::FORMAT_ND, ge::FORMAT_ND});
        this->Output("gather_out")
                .ParamType(REQUIRED)
                .DataType({ge::DT_FLOAT16, ge::DT_BF16})
                .Format({ge::FORMAT_ND, ge::FORMAT_ND})
                .UnknownShapeFormat({ge::FORMAT_ND, ge::FORMAT_ND});

        this->Attr("group").AttrType(REQUIRED).String();
        this->Attr("isTransA").AttrType(OPTIONAL).Bool(false);
        this->Attr("isTransB").AttrType(OPTIONAL).Bool(false);
        this->Attr("gatherIndex").AttrType(OPTIONAL).Int(0);
        this->Attr("commTurn").AttrType(OPTIONAL).Int(0);
        this->Attr("rank_size").AttrType(OPTIONAL).Int(8);
        this->Attr("is_gather_out").AttrType(OPTIONAL).Bool(true);

        this->AICore().SetTiling(optiling::AllGatherMatmulCustomTilingFunc);
        this->AICore().AddConfig("ascendxxx"); // ascendxxx请修改为对应的昇腾AI处理器型号。
        this->MC2().HcclGroup("group");
    }
};
```


14 附录

14.1 Tensor 基础知识参考

14.1.1 Tensor 基本概念

Tensor是算子计算数据的容器，TensorDesc（Tensor描述符）是对Tensor中数据的描述。TensorDesc数据结构包含如下属性如**表14-1**所示。

表 14-1 TensorDesc 属性解释

属性	定义
名称（name）	用于对Tensor进行索引，不同Tensor的name需要保持唯一。
形状（shape）	Tensor的形状，比如(10,)或者(1024, 1024)或者(2, 3, 4)等。如形状(3, 4)表示第一维有3个元素，第二维有4个元素，(3, 4)表示一个3行4列的矩阵数组。 形式：(i1, i2, …, in)，其中i1到in均为正整数。 **说明** 由于AI Core算子编译器限制，不支持传入shape中含0的tensor(空tensor)，开发者在构造输入时需要避免传入空tensor。
数据类型（dtype）	指定Tensor对象的数据类型。 取值范围：float16, float32, int8, int16, int32, uint8, uint16, bfloat16, bool等。

属性	定义
数据排布格式（format）	数据的物理排布格式，定义了解读数据的维度。 详细请参见14.1.2 数据排布格式。

14.1.2 数据排布格式

Format为数据的物理排布格式，定义了解读数据的维度，比如1D，2D，3D，4D，5D等。

NCHW 和 NHWC

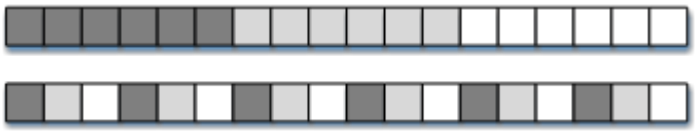
在深度学习领域，多维数据通过多维数组存储，比如卷积神经网络的特征图（Feature Map）通常用四维数组保存，即4D，4D格式解释如下：

- N：Batch数量，例如图像的数目。
- H：Height，特征图高度，即垂直高度方向的像素个数。
- W：Width，特征图宽度，即水平宽度方向的像素个数。
- C：Channels，特征图通道，例如彩色RGB图像的Channels为3。

由于数据只能线性存储，因此这四个维度有对应的顺序。不同深度学习框架会按照不同的顺序存储特征图数据，比如Caffe，排列顺序为[Batch, Channels, Height, Width]，即NCHW。TensorFlow中，排列顺序为[Batch, Height, Width, Channels]，即NHWC。

如图14-1所示，以一张格式为RGB的图片为例，NCHW中，C排列在外层，实际存储的是“RRRRRRGGGGGGBBBBBB”，即同一通道的所有像素值顺序存储在一起；而NHWC中C排列在最内层，实际存储的则是“RGBRGBRGBRGBRGBRGB”，即多个通道的同一位置的像素值顺序存储在一起。

图 14-1 NCHW 和 NHWC 存储示例



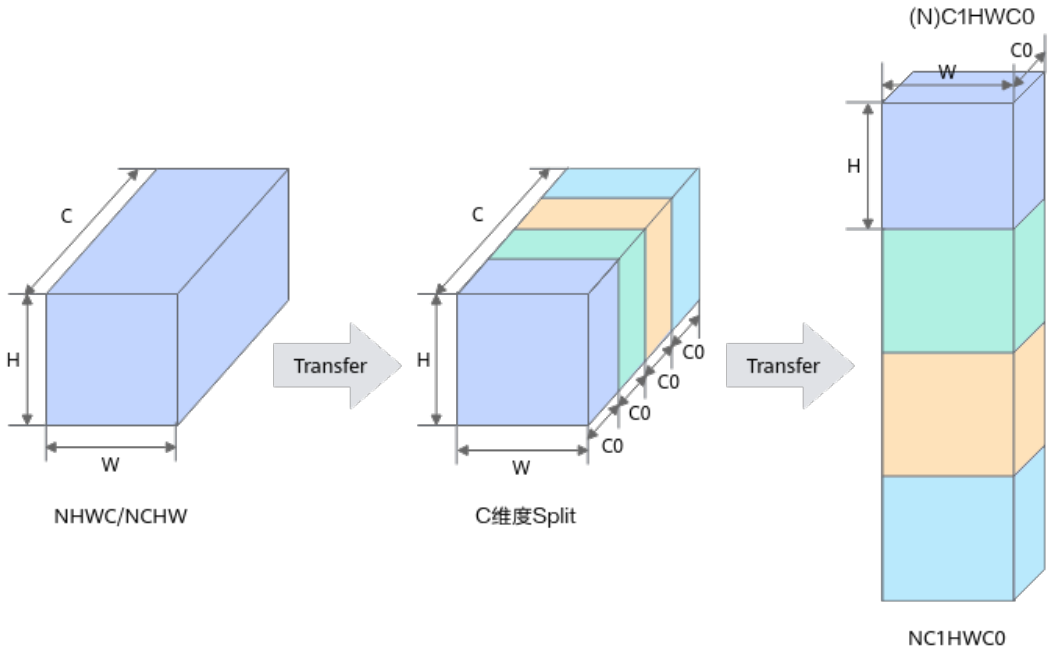
尽管存储的数据相同，但不同的存储顺序会导致数据的访问特性不一致，因此即便进行同样的运算，相应的计算性能也会不同。

NC1HWC0

昇腾AI处理器中，为了提高通用矩阵乘法（GEMM）运算数据块的访问效率，所有张量数据统一采用NC1HWC0的五维数据格式。其中C0与微架构强相关，等于AI Core中矩阵计算单元的大小。

C1=(C+C0-1)/C0。如果结果不整除，向下取整。

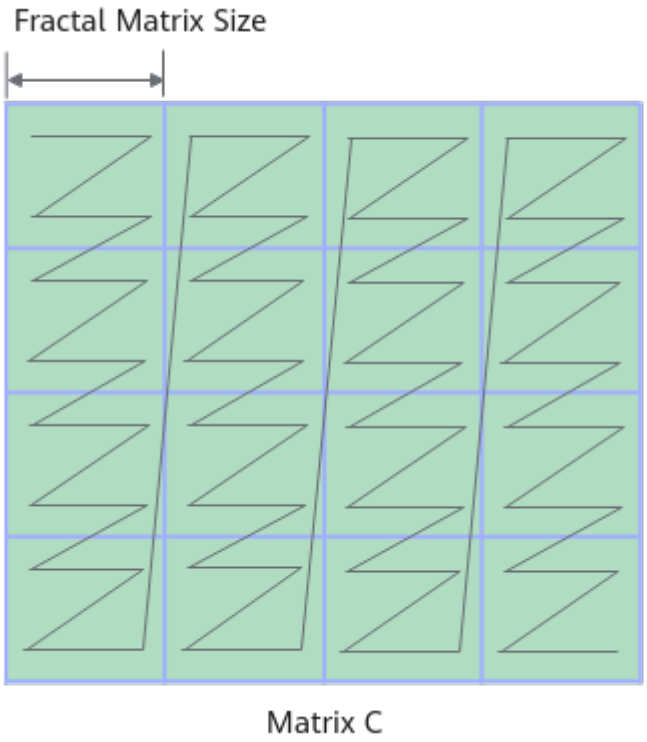
NHWC/NCHW -> NC1HWC0的转换过程为：将数据在C维度进行分割，变成C1份NHWC0/NC0HW，再将C1份NHWC0/NC0HW在内存中连续排列成NC1HWC0，其格式转换示意图如下图所示。



- NHWC -> NC1HWC0的转换公式如下：
  ```
  Tensor.reshape( [N, H, W, C1, C0]).transpose( [0, 3, 1, 2, 4] )
  ```
- NCHW -> NC1HWC0的转换公式如下：
  ```
  Tensor.reshape( [N, C1, C0, H, W]).transpose( [0, 1, 3, 4, 2] )
  ```

FRACTAL_NZ

FRACTAL_NZ是分形格式，如Feature Map的数据存储，在cube单元计算时，输出矩阵的数据格式为NW1H1H0W0。整个矩阵被分为（H1*W1）个分形，按照column major排布，形状如N字形；每个分形内部有（H0*W0）个元素，按照row major排布，形状如z字形。考虑到数据排布格式，将NW1H1H0W0数据格式称为Nz（大N小z）格式。其中，H0,W0表示一个分形的大小，示意图如下所示：



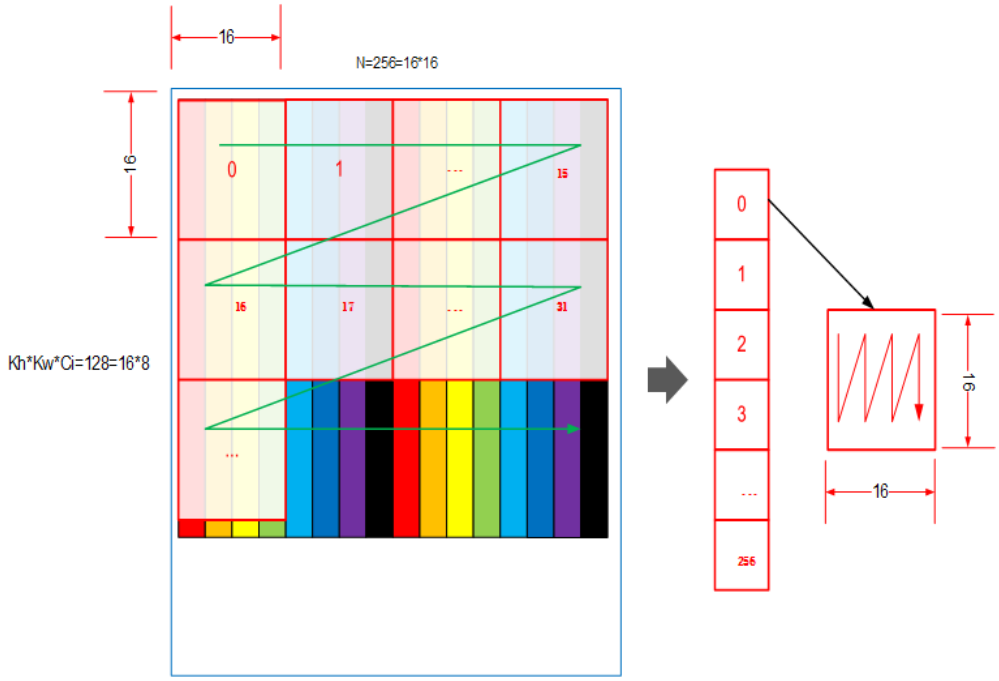
ND –> FRACTAL_NZ的变换过程为：

```
(..., N，H, W )->pad->(..., N, H1*H0, W1*W0)->reshape->(..., N, H1, H0, W1, W0)->transpose->(..., N, W1, H1, H0, W0)
```

FRACTAL_Z

FRACTAL_Z是用于定义卷积权重的数据格式，由FT Matrix（FT：Filter，卷积核）变换得到。FRACTAL_Z是送往Cube的最终数据格式，采用“C1HW,N1,N0,C0”的4维数据排布。

数据有两层Tiling，如下图所示：



第一层与Cube的Size相关，数据按照列的方向连续（小\n）；第二层与矩阵的Size相关，数据按照行的方向连续（大Z）。

例如：HWCN = (2, 2, 32, 32)，将其变成FRACTAL_Z(C1HW, N1, N0, C0) = (8, 2, 16, 16)。

HWCN变换FRACTAL_Z的过程为：

```
Tensor.padding([ [0,0], [0,0], [0,(C0–C%C0)%C0], [0,(N0–N%N0)%N0] ]).reshape( [H, W, C1, C0, N1, N0]).transpose( [2, 0, 1, 4, 5, 3] ).reshape( [C1*H*W, N1, N0, C0])
```

NCHW变换FRACTAL_Z的过程为：

```
Tensor.padding([ [0,(N0–N%N0)%N0], [0,(C0–C%C0)%C0], [0,0], [0,0] ]).reshape( [N1, N0, C1, C0, H, W,]).transpose( [2, 4, 5, 0, 1, 3] ).reshape( [C1*H*W, N1, N0, C0])
```

14.2 基于 VectorCore 编程

基于Ascend C开发的矢量计算相关的算子可以运行在Vector Core上，本节主要介绍如何基于Vector Core架构进行算子编程，该特性仅支持Atlas 推理系列产品。

学习本节内容之前，请先参考**算子实现**、**8 Kernel直调算子开发**、**9 工程化算子开发**学习基于AI Core的算子端到端的开发流程，下文内容会重点介绍基于VectorCore进行算子编程的差异点。具体差异点如下：

1. 完成算子kernel侧开发时，需要通过宏KERNEL_TASK_TYPE_DEFAULT开启支持Vector Core，算子执行时会同时启动AI Core和Vector Core， 此时AI Core会当成Vector Core使用。如下的代码样例展示了使能Vector Core的方法：

```
extern "C" __global__ __aicore__ void add_custom(__gm__ uint8_t *x, __gm__ uint8_t *y, __gm__ uint8_t *z, __gm__ uint8_t *workspace, __gm__ uint8_t *tiling)
{
    GET_TILING_DATA(tilingData, tiling);
    if (workspace == nullptr) {
        return;
    }
    GM_ADDR usr = AscendC::GetUserWorkspace(workspace);
    KernelAdd op;
```



```
    op.Init(x, y, z, tilingData.blockDim, tilingData.totalLength, tilingData.tileNum);
    KERNEL_TASK_TYPE_DEFAULT(KERNEL_TYPE_MIX_VECTOR_CORE); // 使能VectorCore
    if (TILING_KEY_IS(1)) {
        op.Process1();
    } else if (TILING_KEY_IS(2)) {
        op.Process2();
    }
    // ...
}
```

2. 完成host侧tiling开发时，设置的blockDim代表的是AI Core和Vector Core的总数，比如用户在host侧设置blockDim为10，则会启动总数为10的AI Core和Vector Core；为保证启动Vector Core，设置数值应大于AI Core的核数。您可以通过GetCoreNumAic接口获取AI Core的核数，GetCoreNumVector接口获取Vector Core的核数。 如下代码片段，分别为使用kernel直调工程和自定义算子工程时的设置样例，此处设置为AI Core和Vector Core的总和，表示所有AI Core和Vector Core都启动。
   - kernel直调工程
     ```
     auto ascendcPlatform = platform_ascendc::PlatformAscendCManager::GetInstance();
     auto totalCoreNum = ascendcPlatform.GetCoreNumAic();
     // ASCENDXXX请替换为实际的版本型号
     if (ascendcPlatform.GetSocVersion() == platform_ascendc::SocVersion::ASCENDXXX) {
       totalCoreNum = totalCoreNum + ascendcPlatform.GetCoreNumVector();
     }
     ...
     kernel_name<<<totalCoreNum , l2ctrl, stream>>>(argument list);
     ```
   - 自定义算子工程
     ```
     // 配套的host侧tiling函数示例：
     ge::graphStatus TilingFunc(gert::TilingContext* context)
     {
         // 使能VectorCore，将blockDim置为AI Core中vector核数 + Vector Core中的vector核数
         auto ascendcPlatform = platform_ascendc::PlatformAscendC(platformInfo);
         auto totalCoreNum = ascendcPlatform.GetCoreNumAic();
         // ASCENDXXX请替换为实际的版本型号
         if (ascendcPlatform.GetSocVersion() == platform_ascendc::SocVersion::ASCENDXXX) {
           totalCoreNum = totalCoreNum + ascendcPlatform.GetCoreNumVector();
         }
         context->SetBlockDim(totalCoreNum);
     }
     ```

说明

- 请参考Ascend C API中具体API支持的型号，来判断API接口是否支持Atlas推理系列产品 Vector Core。
- 支持VectorCore后，因为AICore和VectorCore会分别执行，通过不同的任务进行调度，所以不支持核间同步指令，如IBSet、IBWait、SyncAll等。
- 算子计算溢出（输入inf/nan或计算结果超出范围）时，需注意AICore和VectorCore结果表现不一致，AICore仅支持饱和模式，VectorCore仅支持inf/nan模式。

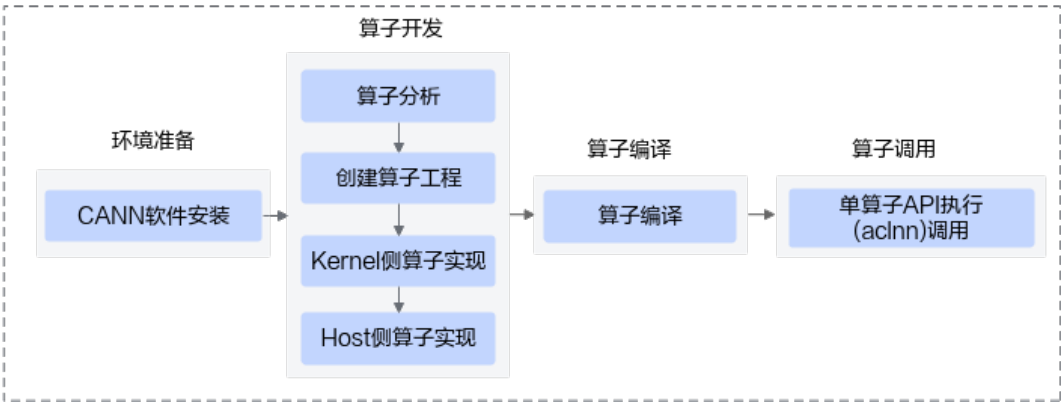
14.3 简易自定义算子工程

本章节介绍的简易自定义算子工程，是上文中介绍的自定义算子工程的简化版，对算子的编译、打包、部署过程进行简化，便于开发者将该工程集成到自己的算子工程。

说明

- 使用该工程，支持在如下平台进行自定义算子开发：
  - Atlas A2训练系列产品
  - Atlas 推理系列产品
- 使用本工程开发的算子，只支持通过单算子API执行（aclnn）方式进行调用。
- 本工程暂不支持算子的交叉编译功能。

基于简易自定义算子工程的算子开发流程图如下：



创建算子工程

和**9.2 创建算子工程**类似，简易自定义算子工程通过msOpGen生成，基于算子原型定义输出算子工程，包括**算子host侧代码实现文件**、**算子kernel侧实现文件**以及**工程编译配置文件等**。主要差异点在于：创建简易算子工程需要通过**-f**参数配置framework框架为aclnn。

说明

使用msOpGen工具创建算子工程之前，需要参考**2 环境准备**章节安装驱动固件和CANN软件包，完成开发环境和运行环境的准备。

使用msOpGen工具创建简易算子开发工程的步骤如下：

步骤1 编写算子的原型定义json文件，用于生成算子开发工程。

例如，AddCustom算子的json文件命名为add_custom.json，文件内容如下：

```
[
  {
    "op": "AddCustom",
    "input_desc": [
      {
        "name": "x",
        "param_type": "required",
        "format": [
          "ND",
          "ND",
          "ND"
        ],
        "type": [
          "fp16",
          "float",
          "int32"
        ]
      },
      {
```



```
            "name": "y",
            "param_type": "required",
            "format": [
               "ND",
               "ND",
               "ND"
            ],
            "type": [
               "fp16",
               "float",
               "int32"
            ]
         }
      ],
      "output_desc": [
         {
            "name": "z",
            "param_type": "required",
            "format": [
               "ND",
               "ND",
               "ND"
            ],
            "type": [
               "fp16",
               "float",
               "int32"
            ]
         }
      ]
   }
]
```

步骤2 使用msOpGen工具生成算子的开发工程。以生成AddCustom的算子工程为例，下文仅针对关键参数进行解释，详细参数说明请参见msOpGen工具。

```
${INSTALL_DIR}/python/site-packages/bin/msopgen gen -i $HOME/sample/add_custom.json -c ai_core-<soc_version> -lan cpp -out $HOME/sample/AddCustom -f aclnn
```

- **${INSTALL_DIR}**为CANN软件安装后文件存储路径，请根据实际环境进行替换。
- -i：指定算子原型定义文件*add_custom*.json所在路径，请根据实际情况修改。
- -c：ai_core-*<soc_version>*代表算子在AI Core上执行，*<soc_version>*为昇腾AI处理器的型号。

  说明

  AI处理器的型号*<soc_version>*请通过如下方式获取：

  - 在安装昇腾AI处理器的服务器执行**npu-smi info**命令进行查询，获取**Chip Name**信息。实际配置值为AscendChip Name，例如**Chip Name**取值为*xxxyy*，实际配置值为Ascend*xxxyy*。

  基于同系列的AI处理器型号创建的算子工程，其基础功能（基于该工程进行算子开发、编译和部署）通用。
- -lan：参数cpp代表算子基于Ascend C编程框架，使用C/C++编程语言开发。
- -out：生成文件所在路径，可配置为绝对路径或者相对路径，并且工具执行用户对路径具有可读写权限。若不配置，则默认生成在执行命令的当前路径。
- -f：表示框架类型，aclnn表示生成简易工程。

步骤3 命令执行完后，会在-out指定目录或者默认路径下生成算子工程目录，工程中包含算子实现的模板文件，编译脚本等，以AddCustom算子为例，目录结构如下所示：

```
AddCustom
├── build.sh                    // 编译入口脚本
├── cmake
│   ├── config.cmake            // 编译配置项
```



```
│   ├── func.cmake
│   ├── intf.cmake
│   └── util                    // 算子工程编译所需脚本及公共编译文件存放目录
├── CMakeLists.txt              // 算子工程的CMakeLists.txt
├── op_host                     // host侧实现文件
│   ├── add_custom_tiling.h     // 算子tiling定义文件
│   ├── add_custom.cpp          // 算子原型注册、tiling实现等内容文件
│   ├── CMakeLists.txt
├── op_kernel                   // kernel侧实现文件
│   ├── CMakeLists.txt
│   ├── add_custom.cpp          // 算子代码实现文件
```

说明

上述目录结构中的粗体文件为后续算子开发过程中需要修改的文件，其他文件无需修改。

----结束

算子实现

参考**9.3.2 Kernel侧算子实现**、**9.3.3 Host侧tiling实现**、**9.3.1 算子原型定义**完成算子实现。

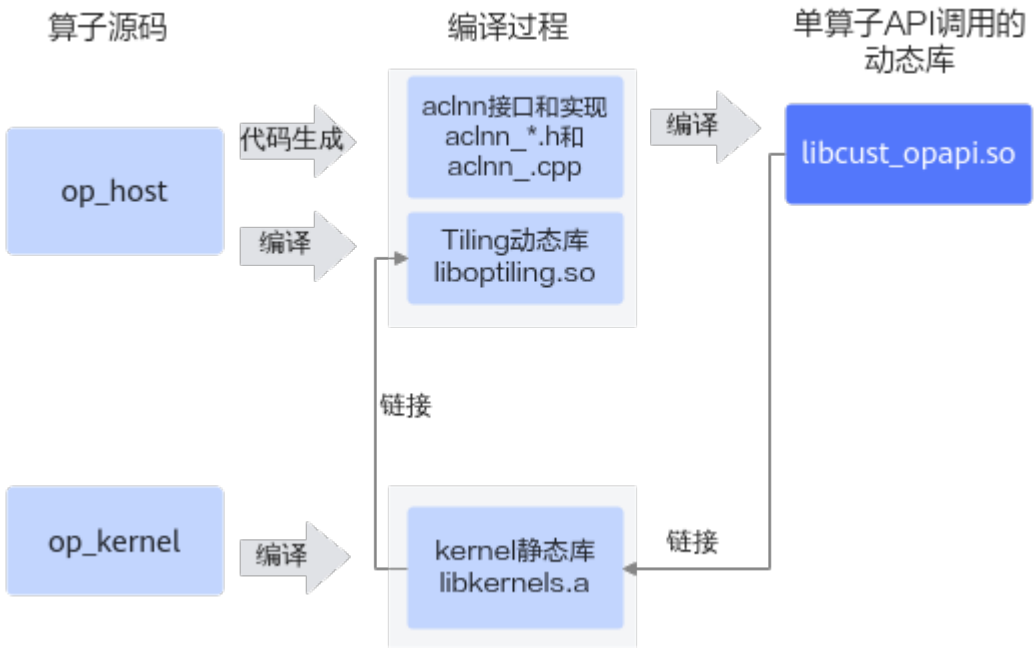
算子编译

算子kernel侧和host侧实现开发完成后，需要对算子进行编译，生成算子静态库；自动生成aclnn调用实现代码和头文件，链接算子静态库生成aclnn动态库，以支持后续的单算子API执行方式（aclnn）的算子调用。编译过程如下：

- 根据host侧算子实现文件自动生成aclnn接口aclnn_*.h和aclnn实现文件aclnn_.cpp。
- 编译Tiling实现和算子原型定义生成Tiling动态库liboptiling.so（libcust_opmaster_rt2.0）。
- 编译kernel侧算子实现文件，并加载Tiling动态库，生成kernel静态库libkernels.a。
- 编译aclnn实现文件，并链接kernel静态库libkernels.a生成单算子API调用的动态库libcust_opapi.so。

上述编译过程示意图如下：

图 14-2 编译过程示意图



上文描述的过程都封装在编译脚本中，开发者进行编译时参考如下的步骤进行操作：

步骤1 完成工程编译相关配置。

- 修改cmake目录下config.cmake中的配置项，config.cmake文件内容如下：

```
set(CMAKE_CXX_FLAGS_DEBUG "")
set(CMAKE_CXX_FLAGS_RELEASE "")

if (NOT DEFINED CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release CACHE STRING "")
endif()
if (CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT)
    set(CMAKE_INSTALL_PREFIX "${CMAKE_SOURCE_DIR}/build_out" CACHE PATH "" FORCE)
endif()
if (NOT DEFINED ASCEND_CANN_PACKAGE_PATH)
    set(ASCEND_CANN_PACKAGE_PATH /usr/local/Ascend/ascend-toolkit/latest CACHE PATH "") //请替换为CANN软件包安装后的实际路径
endif()
if (NOT DEFINED ASCEND_PYTHON_EXECUTABLE)
    set(ASCEND_PYTHON_EXECUTABLE python3 CACHE STRING "")
endif()
if (NOT DEFINED ASCEND_COMPUTE_UNIT)
    set(ASCEND_COMPUTE_UNIT ascendxxx CACHE STRING "")
endif()
if (NOT DEFINED ENABLE_TEST)
    set(ENABLE_TEST FALSE CACHE BOOL "")
endif()
if (NOT DEFINED ENABLE_CROSS_COMPILE)
    set(ENABLE_CROSS_COMPILE  FALSE CACHE BOOL "")
endif()
if (NOT DEFINED CMAKE_CROSS_PLATFORM_COMPILER)
    set(CMAKE_CROSS_PLATFORM_COMPILER "/your/cross/compiler/path" CACHE PATH "")
endif()
set(ASCEND_TENSOR_COMPILER_PATH ${ASCEND_CANN_PACKAGE_PATH}/compiler)
set(ASCEND_CCEC_COMPILER_PATH ${ASCEND_TENSOR_COMPILER_PATH}/ccec_compiler/bin)
set(ASCEND_AUTOGEN_PATH ${CMAKE_BINARY_DIR}/autogen)
file(MAKE_DIRECTORY ${ASCEND_AUTOGEN_PATH})
set(CUSTOM_COMPILE_OPTIONS "custom_compile_options.ini")
execute_process(COMMAND rm -rf ${ASCEND_AUTOGEN_PATH}/${CUSTOM_COMPILE_OPTIONS}
                COMMAND touch ${ASCEND_AUTOGEN_PATH}/${CUSTOM_COMPILE_OPTIONS})
```


表 14-2 需要开发者配置的常用参数列表

参数名称	参数描述	默认值
ASCEND_CANN_PACKAGE_PATH	CANN软件包安装路径，请根据实际情况进行修改。	“/usr/local/Ascend/ascend-toolkit/latest”
CMAKE_BUILD_TYPE	编译模式选项，可配置为： – “Release”，Release版本，不包含调试信息，编译最终发布的版本。 – “Debug”，“Debug”版本，包含调试信息，便于开发者开发和调试。	“Release”
CMAKE_INSTALL_PREFIX	编译产物存放的目录，不指定则为默认值。	${CMAKE_SOURCE_DIR}/build_out： 算子工程目录下的build_out目录

- 配置编译相关环境变量（可选）

表 14-3 环境变量说明

环境变量	配置说明

CMAKE_CXX_COMPILER_LAUNCHER	用于配置C++语言编译器（如g++）、毕昇编译器的启动器程序为ccache，配置后即可开启cache缓存编译，加速重复编译并提高构建效率。用法如下，在对应的CMakeLists.txt进行设置：
	`set(CMAKE_CXX_COMPILER_LAUNCHER <launcher_program>)`
	其中<launcher_program>是ccache的安装路径，比如ccache的安装路径为/usr/bin/ccache，示例如下：
	`set(CMAKE_CXX_COMPILER_LAUNCHER /usr/bin/ccache)`

步骤2 （可选）如果需要编译多个算子，在op_kernel目录下的CMakeLists.txt中增加要编译的算子。

```
# set custom compile options
if ("${CMAKE_BUILD_TYPE}x" STREQUAL "Debugx")
    add_ops_compile_options(ALL OPTIONS -g -O0)
endif()
#多算子编译通过add_kernel_compile命令增加算子源码文件即可
add_kernel_compile(AddCustom ${CMAKE_CURRENT_SOURCE_DIR}/add_custom.cpp)
add_kernel_compile(SubCustom ${CMAKE_CURRENT_SOURCE_DIR}/sub_custom.cpp)
if(ENABLE_TEST AND EXISTS ${CMAKE_CURRENT_SOURCE_DIR}/testcases)
    add_subdirectory(testcases)
endif()
```

步骤3 （可选）在算子工程中，如果开发者想对算子kernel侧代码增加一些自定义的编译选项，可以参考**支持自定义编译选项**进行编译选项的定制。

步骤4 在算子工程目录下执行如下命令，进行算子工程编译。

```
./build.sh
```

编译成功后，会在CMAKE_INSTALL_PREFIX/op_api目录存放生成的aclnn头文件和lib库，每一个算子都会对应一个单独的头文件。具体目录结构如下：

```
├── op_api
│   ├── include
│   │   ├── aclnn_optype1.h
│   │   └── aclnn_optype2.h
│   │   └── aclnn_optypexxx.h
│   ├── lib
│   │   ├── libcust_opapi.so
```

对于lib目录下生成的库文件可通过msobjdump工具进一步解析得到kernel信息，具体操作参见**14.5 msobjdump工具**。

----结束

14.4 show_kernel_debug_data 工具

静态图场景下，整图算子全部下沉到NPU侧执行，kernel侧单算子调试信息（通过printf接口）需要在模型执行结束后才能获取。本工具提供了离线解析能力，帮助用户获取并解析调试信息（将bin文件解析成可读格式）。

说明

show_kernel_debug_data支持多用户并发调用，但用户需要指定不同的落盘路径，否则可能出现落盘内容被覆盖等问题。

本工具支持的产品型号：

- Atlas A2训练系列产品/Atlas 800I A2推理产品
- Atlas 推理系列产品
- Atlas 200/500 A2推理产品

工具安装

步骤1 安装工具。

工具跟随CANN软件包发布（参考**2 环境准备**完成CANN安装），其路径默认为“${INSTALL_DIR}/tools/ascendc_tools/show_kernel_debug_data”，其中${INSTALL_DIR}请替换为CANN软件安装后文件存储路径。例如，若安装的Ascend-cann-toolkit软件包，则安装后文件存储路径为：$HOME/Ascend/ascend-toolkit/latest。

步骤2 设置环境变量。

- root用户安装Ascend-cann-toolkit包时
  ```
  source /usr/local/Ascend/ascend-toolkit/set_env.sh
  source /usr/local/Ascend/ascend-toolkit/latest/toolkit/bin/setenv.bash
  ```
- 非root用户安装Ascend-cann-toolkit包时
  ```
  source ${HOME}/Ascend/ascend-toolkit/set_env.sh
  source ${HOME}/Ascend/ascend-toolkit/latest/toolkit/bin/setenv.bash
  ```

步骤3 检查工具是否安装成功。

执行如下命令，若能正常显示--help或-h信息，则表示工具环境正常，功能可正常使用。


```
show_kernel_debug_data -h
```

----结束

使用方法

- **命令行方式**

  ```
  show_kernel_debug_data ${bin_file_path} ${output_path}
  ```

参数	说明	默认值	是否必选
${bin_file_path}	kernel侧调试信息落盘的bin文件路径。	-	是
${output_path}	解析结果的保存路径。	默认是当前命令行执行目录下。	否

  命令行示例如下：

  ```
  show_kernel_debug_data  ./input/dump_workspace.bin  ./output_dir
  ```

- **API方式**

  表 14-4 show_kernel_debug_data 接口说明表

函数原型	def show_kernel_debug_data(bin_file_path: str, output_path: str = './') -> None	
函数功能	获取kernel侧调试信息并解析成可读文件。	
参数（IN）	bin_file_path	kernel侧调试信息落盘的bin文件路径，字符串类型。
	output_path	解析结果的保存路径，字符串类型，默认是当前接口调用脚本所在目录下。
参数（OUT）	NA	-
返回值	NA	-
使用约束	无	
调用示例	from show_kernel_debug_data import show_kernel_debug_data show_kernel_debug_data(./input/dump_workspace.bin)	

产物说明

工具解析结果文件目录结构如下：

```
├ ${output_path}
├── PARSER_${timestamp}          // ${timestamp}表示时间戳。
│   ├── parser.log               // 工具解析的日志，包含kernel侧日常流程和printf打印信息。
```

14.5 msobjdump 工具

本工具针对任意场景下算子编译生成的ELF文件（Executable and Linkable Format）提供解析和解压功能，并将结果信息以可读格式呈现，方便开发者直观获得kernel文件信息。

说明

ELF文件是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储的文件格式，包括常见的*.a、*.so文件等。ELF文件构成如下：

- ELF头部：描述了整个文件的组织结构，包括文件类型、机器类型、版本号等信息。
- 程序头部表：描述了文件中各种段（segments）信息，包括程序如何加载到内存中执行的信息。
- 节区头部表：描述了文件中各个节（sections）信息，包括程序的代码、数据、符号表等。

ELF文件主要用于Linux和其他类Unix操作系统中，用于程序的执行和链接。

工具安装

步骤1 安装msobjdump工具。

工具跟随CANN软件包发布（参考**2 环境准备**完成CANN安装），其路径默认为“${INSTALL_DIR}/tools/ascendc_tools/msobjdump”，其中${INSTALL_DIR}请替换为CANN软件安装后文件存储路径。例如，若安装的Ascend-cann-toolkit软件包，则安装后文件存储路径为：$HOME/Ascend/ascend-toolkit/latest。

步骤2 设置环境变量。

- root用户安装Ascend-cann-toolkit包时
  ```
  source /usr/local/Ascend/ascend-toolkit/set_env.sh
  source /usr/local/Ascend/ascend-toolkit/latest/toolkit/bin/setenv.bash
  ```
- 非root用户安装Ascend-cann-toolkit包时
  ```
  source ${HOME}/Ascend/ascend-toolkit/set_env.sh
  source ${HOME}/Ascend/ascend-toolkit/latest/toolkit/bin/setenv.bash
  ```

步骤3 检查工具是否安装成功。

执行如下命令，若能正常显示--help或-h信息，则表示工具环境正常，功能可正常使用。

```
msobjdump -h
```

----结束

功能介绍

- **解析ELF文件**

  解析ELF文件内部信息，如文件长度、 文件类型信息、每个文件的段信息、符号表信息等，命令样式如下：

  ```
  msobjdump --dump-elf ${elf_file}
  ```

  其中${elf_file}为待解析ELF文件路径，如/home/op_api/lib_api.so。

- **解压ELF文件**

解压ELF文件并落盘到指定目录，命令样式如下：

```
msobjdump --extract-elf ${elf_file} --out-dir ${out_path}
```

其中${elf_file}为待解压ELF文件路径，如/home/op_api/lib_api.so；${out_path}为落盘文件目录，如/home/extract/。若不设置--out-dir，工具默认将解压文件落盘到当前执行路径下。

- **获取ELF文件列表**

  终端打印所有device.o文件列表，命令样式如下：

  ```
  msobjdump --list-elf ${elf_file}
  ```

  其中${elf_file}为待打印的ELF文件路径，如/home/op_api/lib_api.so。

工具全量命令参数说明请参考**表14-5**。

表 14-5 msobjdump 参数说明

功能参数（区分大小写）	功能说明	是否必选
-d，--dump-elf	解析ELF文件中包含的每个device.o文件的文件长度、 文件类型、符号表等信息，并在终端屏显，关键字段含义参见**表14-6**。	三选一，必选
-e，--extract-elf	解压ELF文件中包含的每个device.o/device.json文件，并按原始文件夹规则落盘到输出路径下。 若未指定--out-dir参数，默认落盘到当前执行路径下。	
-l，--list-elf	打屏显示输ELF文件中包含的device.o文件列表。	
-o，--out-dir	解压文件的落盘路径， 需要与--extract-elf参数配套使用。 **说明** msobjdump支持多用户并发调用，但需要指定不同的--out-dir，否则可能出现落盘内容被覆盖的问题。	否

表 14-6 ELF 解析字段说明

关键字段	说明	是否必选
VERSION	版本号。	否
TYPE COUNT	当前section段包含的kernel文件个数。	
ELF FILE ${id}	罗列的kernel文件，文件名是按照 ${sec_prefix}_${file_index}_${kernel_type}.o 拼接而成，其中${sec_prefix}为section段名（工具根据“.ascend.kernel”关键字搜索获取），${file_index}表示文件编号，${kernel_type}表示kernel类型。	

关键字段	说明	是否必选
KERNEL TYPE	当前kernel文件的类型，映射关系为{0 : 'mix', 1: 'aiv', 2: 'aic'}。	
KERNEL LEN	当前kernel文件的长度。	
elf heard infos	包括ELF Header、Section Headers、Key to Flags、Program Headers、Symbol表等信息。	是

使用样例（Kernel 直调算子工程）

以MatMulInvocationNeo算子为例，完整的算子工程请参考**matmul多核Kernel直调样例**。假设${cmake_install_dir}为算子Cmake编译产物根目录，产物目录结构如下，类似**CMake编译配置文件编写**。

```
out
├── lib
│   ├── libascendc_kernels_npu.so
├── include
│   ├── ascendc_kernels_npu
│       ├── aclrtlaunch_matmul_custom.h
│       ├── aclrtlaunch_triple_chevrons_func.h
```

工具对编译生成的库文件（如*.so、*.a等）进行解析和解压，功能实现命令样例如下：

- **解析每个device.o文件**

  ```
  msobjdump --dump-elf ${cmake_install_dir}/out/libascendc_kernels_npu.so
  ```

  执行命令后，终端会打印所有device信息，示例如下，字段含义参见**表14-6**。

  ```
  ============================
  [VERSION]: 1
  [TYPE COUNT]: 1
  ============================
  [ELF FILE 0]: ascendxxxb1_ascendc_kernels_npu_0_mix.o
  [KERNEL TYPE]: mix
  [KERNEL LEN]: 511560
  ====== [elf heard infos] ======
  ELF Header:
    Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
    Class:                             ELF64
    Data:                              2's complement, little endian
    Version:                           1 (current)
    OS/ABI:                            UNIX - System V
    ABI Version:                       0
    Type:                              EXEC (Executable file)
    Machine:                           <unknown>: 0x1029
    Version:                           0x1
    Entry point address:               0x0
    Start of program headers:          64 (bytes into file)
    Start of section headers:          510280 (bytes into file)
    Flags:                             0x940000
    Size of this header:               64 (bytes)
    Size of program headers:           56 (bytes)
    Number of program headers:         2
    Size of section headers:           64 (bytes)
    Number of section headers:         20
    Section header string table index: 18

  Section Headers:
    [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  ```



```
  [ 0]                NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text          PROGBITS        0000000000000000 0000b0 010a08 00  AX  0   0  4
   ..................................................................................
  [19] .strtab        STRTAB          0000000000000000 071278 00b6cb 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x0000b0 0x0000000000000000 0x0000000000000000 0x010aa8 0x010aa8 R E 0x1000
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0
......
```

- **解压每个device.o文件并落盘**

  ```
  msobjdump --extract-elf ${cmake_install_dir}/out/libascendc_kernels_npu.so
  ```

  执行命令后，默认在当前执行路径下落盘ascendxxxb1_ascendc_kernels_npu_0_mix.o文件。

- **获取device.o文件列表**

  ```
  msobjdump --list-elf ${cmake_install_dir}/out/libascendc_kernels_npu.so
  ```

  执行命令后，终端会打印所有文件，屏显样例如下：

  ```
  ELF file    0: ascendxxxb1_ascendc_kernels_npu_0_mix.o
  ```

使用样例（简易自定义算子工程）

以下面的算子工程为例，假设${cmake_install_dir}为算子Cmake编译产物根目录，产物目录结构如下，类似**9.4.1 算子工程编译**。

```
├── op_api
│   ├── include
│   │   ├── aclnn_acos_custom.h
│   │   ├── aclnn_matmul_leakyrelu_custom.h
│   │   ├── .........
│   ├── lib
│   │   ├── libcust_opapi.so
```

工具对编译生成的库文件（如*.so、*.a等）进行解析和解压，功能实现命令样例如下：

- **解析每个device.o文件**

  ```
  msobjdump --dump-elf ${cmake_install_dir}/op_api/lib/libcust_opapi.so
  ```

  执行命令后，终端会打印每个算子device.o的section段、符号表信息，示例如下，字段含义参见**表14-6**。

  ```
  ===== [elf heard infos] in
  ascendxxx_acos_custom_AcosCustom_da824ede53d7e754f85c14b9446ec2fc.o =====:
  ELF Header:
    Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
    Class:                             ELF64
    Data:                              2's complement, little endian
    Version:                           1 (current)
    OS/ABI:                            UNIX - System V
    ..................................................
    Size of program headers:           56 (bytes)
    Number of program headers:         3
    Size of section headers:           64 (bytes)
    Number of section headers:         9
    Section header string table index: 7
  Section Headers:
    [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
    [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
     ..................................................................................
  ```

```
  [ 8] .strtab            STRTAB          0000000000000000 00529b 000119 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
.....................

===== [elf heard infos] in
ascendxxx_matmul_leakyrelu_custom_MatmulLeakyreluCustom_e052bee3255764ac919095f3bdf83389.
o =====:
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  ..........................................
  Section header string table index: 6
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 0000e8 007ed8 00  AX  0   0  4
  [ 2] .data             PROGBITS        0000000000008000 0080e8 000008 00  WA  0   0 256
  [ 3] .comment          PROGBITS        0000000000000000 0080f0 000043 01  MS  0   0  1
  [ 4] .bl_uninit        NOBITS          0000000000000000 008133 000020 00      0   0  1
  [ 5] .symtab           SYMTAB          0000000000000000 008138 0000c0 18      7   1  8
  [ 6] .shstrtab         STRTAB          0000000000000000 0081f8 00003b 00      0   0  1
  [ 7] .strtab           STRTAB          0000000000000000 008233 0000ec 00      0   0  1
.....................................
```

- **解压每个device.o文件并落盘**

```
msobjdump --extract-elf ${cmake_install_dir}/op_api/lib/libcust_opapi.so
```

执行命令后，默认在当前执行路径下保存解压文件，产物目录如下：

```
|-- config                                                    // 算子原型配置文件目录
|   ├── ${soc_version}
|       ├── acos_custom.json
|       ├── matmul_leakyrelu_custom.json
|       ├── .......
|-- ${soc_version}                                            // 昇腾AI处理器名
|   ├── acos_custom                                           // 基础单算子编译文件*.o和对应的*.json文件
|       ├── AcosCustom_da824ede53d7e754f85c14b9446ec2fc.json      // 命名规则：${op_type}_$
{parm_info}.json或${op_type}_${parm_info}.o，${parm_info}是基于算子输入/输出dtype、shape等信息
生成的标识码
|       ├── AcosCustom_da824ede53d7e754f85c14b9446ec2fc.o
|       ├── AcosCustom_dad9c8ca8fcbfd789010c8b1c0da8e26.json
|       ├── AcosCustom_dad9c8ca8fcbfd789010c8b1c0da8e26.o
|   ├── matmul_leakyrelu_custom
|       ├── MatmulLeakyreluCustom_e052bee3255764ac919095f3bdf83389.json
|       ├── MatmulLeakyreluCustom_e052bee3255764ac919095f3bdf83389.o
|   ├── axpy_custom
|       ├── .....
```

以acos_custom算子编译产物解压为例：

- 查看算子原型（acos_custom.json）

```
{
   "binList": [
      {
         "implMode": "high_performance",
         "int64Mode": false,
         "simplifiedKeyMode": 0,
         "simplifiedKey": [...],
         "staticKey":
"96b2b4bb2e35fa3db8c2714828d6e4f367b0731c9514401e1a2d41ebd63833c7,ee37ce8796ef139d
ed8f105c019b2819f38447bdeeb8ff7f58430228e6d9a361",
         "inputs": [
            {
               "name": "x",
               "index": 0,
               "dtype": "float32",
```



```
                "format": "ND",
                "paramType": "required",
                "shape": [
                    -2
                ],
                "format_match_mode": "FormatAgnostic"
            }
        ],
        "outputs": [
            {
                "name": "y",
                "index": 0,
                "dtype": "float32",
                "format": "ND",
                "paramType": "required",
                "shape": [
                    -2
                ],
                "format_match_mode": "FormatAgnostic"
            }
        ],
        "attrs": [
            {
                "name": "tmp",
                "dtype": "int",
                "value": 0
            },
        ..............
}
```

- 解析${op_type}_${parm_info}.o获取.ascend.meta section段信息。

```
msobjdump --dump-elf ./AcosCustom_da824ede53d7e754f85c14b9446ec2fc.o
```

执行命令后，终端屏显信息如下，参数说明参见**表14-7**、**表14-8**、**表14-9**。

```
.ascend.meta. [0]: AcosCustom_da824ede53d7e754f85c14b9446ec2fc_1
T: 1  L: 4  V: 3
F_TYPE_KTYPE: K_TYPE_AIV
.ascend.meta. [0]: AcosCustom_da824ede53d7e754f85c14b9446ec2fc_2_mix_aiv
T: 1  L: 4  V: 5
F_TYPE_KTYPE: K_TYPE_MIX_AIV_MAIN
T: 3  L: 4  V: [0:1]
F_TYPE_MIX_TASK_RATION: [0:1]
.ascend.meta. [0]: AcosCustom_da824ede53d7e754f85c14b9446ec2fc_3_mix_aiv
T: 1  L: 4  V: 5
F_TYPE_KTYPE: K_TYPE_MIX_AIV_MAIN
T: 3  L: 4  V: [0:1]
F_TYPE_MIX_TASK_RATION: [0:1]
```

表 14-7 类型映射关系

解析类型	解析类型映射	说明
1	F_TYPE_KTYPE	表示kernel type。
2	F_TYPE_CROSS_CORE_SYNC	表示硬同步syncall类型。
3	F_TYPE_MIX_TASK_RATION	表示核函数运行时的core分配类型。

表 14-8 kernel type 映射关系

kernel type数值	对应的kernel type	说明
1	K_TYPE_AICORE	**该参数为预留参数，当前版本暂不支持。** 算子执行时仅会启动AI Core，比如用户在host侧设置blockdim为5，则会启动5个AI Core。
2	K_TYPE_AIC	算子执行时仅启动AI Core上的Cube核：比如用户在host侧设置blockdim为10，则会启动10个Cube核。
3	K_TYPE_AIV	算子执行时仅启动AI Core上的Vecor核：比如用户在host侧设置blockdim为10，则会启动10个Vecor核。
4	K_TYPE_MIX_AIC_MAIN	AIC、AIV混合场景下，设置核函数的类型为MIX ，算子执行时会同时启动AI Core上的Cube核和Vector核，比如用户在host侧设置blockdim为10，且设置task_ration为1：2，则会启动10个Cube核和20个Vector核。
5	K_TYPE_MIX_AIV_MAIN	AIC、AIV混合场景下，使用了多核控制相关指令时，设置核函数的类型为MIX，算子执行时会同时启动AI Core上的Cube核和Vector核，比如用户在host侧设置blockdim为10，且设置task_ration为1：2，则会启动10个Vector核和20个Cube核。
6	K_TYPE_AIC_ROLLBACK	算子执行时会同时启动AI Core和Vector Core，此时AI Core会当成Cube Core使用。
7	K_TYPE_AIV_ROLLBACK	算子执行时会同时启动AI Core和Vector Core，此时AI Core会当成Vector Core使用。

表 14-9 硬同步映射关系

硬同步数值	对应的硬同步类型	说明
1	C_TYPE_USE_SYNC	使用硬同步。
0	C_TYPE_NO_USE_SYNC	不使用硬同步。

- 查看${op_type}_${parm_info}.json，直观获取device.o算子信息。

```
{
    "binFileName": "AcosCustom_da824ede53d7e754f85c14b9446ec2fc",
    "binFileSuffix": ".o",
```



```
    "blockDim": -1,
    "coreType": "MIX",
    "intercoreSync": 1,
    "kernelName": "AcosCustom_da824ede53d7e754f85c14b9446ec2fc",
    "magic": "RT_DEV_BINARY_MAGIC_ELF",
    "memoryStamping": [],
    "opParaSize": 24,
    "parameters": [],
    "sha256": "94e32d04fcaf435411af194c8b4c85447744879aacec198448b820246f96d310",
    "workspace": {
        "num": 1,
        "size": [
            -1
        ],
        "type": [
            0
        ]
    },
    "kernelList": [
        {
            "tilingKey": 1,
            "kernelType": "MIX_AIC",
            "taskRation": "0:1",
            "crossCoreSync": 0,
            "kernelName": "AcosCustom_da824ede53d7e754f85c14b9446ec2fc_1"
        },
        ..............
    ],
    "taskRation": "tilingKey",
    "optionalInputMode": "gen_placeholder",
    "debugOptions": "printf",
    "debugBufSize": 78643200,
    "compileInfo": {},
    "supportInfo": {                                                    // 算子原型信息
        "implMode": "high_performance",
        "int64Mode": false,
        "simplifiedKeyMode": 0,
        "simplifiedKey": [...],
        "staticKey":
"96b2b4bb2e35fa3db8c2714828d6e4f367b0731c9514401e1a2d41ebd63833c7,ee37ce8796ef139d
ed8f105c019b2819f38447bdeeb8ff7f58430228e6d9a361",
        "inputs": [
            {
                "name": "x",
                "index": 0,
                "dtype": "float32",
                "format": "ND",
                "paramType": "required",
                "shape": [
                    -2
                ],
                "format_match_mode": "FormatAgnostic"
            }
        ],
         ..........
}
```

- **获取device.o文件列表**

  ```
  msobjdump --list-elf ${cmake_install_dir}/op_api/lib/libcust_opapi.so
  ```

  执行命令后，终端会打印所有文件，屏显样例如下：

  ```
  ELF file    0: ascendxxx_acos_custom_AcosCustom_dad9c8ca8fcbfd789010c8b1c0da8e26.json
  ELF file    1: ascendxxx_acos_custom_AcosCustom_dad9c8ca8fcbfd789010c8b1c0da8e26.o
  ..................
  ELF file    2: ascendxxx_acos_custom_AcosCustom_da824ede53d7e754f85c14b9446ec2fc.json
  ELF file    3: ascendxxx_acos_custom_AcosCustom_da824ede53d7e754f85c14b9446ec2fc.o
  ```


15 FAQ

15.1 核函数运行验证时算子存在精度问题

现象描述

在进行算子NPU域的运行验证时，通过md5sum等方式进行算子精度比对，实际数据和真值数据不一致，算子存在精度问题。本示例中通过md5sum来进行精度比对，打印出的真值数据和实际输出数据的md5值不一致，具体打印信息如下：

```
md5sum:
45e17ee4c068a655be2af4d8c3a1f191  output/golden.bin
6a99e41a84b14dd04f32730ceb9a3988  output/output_y.bin
```

问题根因

算子出现精度问题，一般是由于算子的实现逻辑有误。

定位步骤

Ascend C提供孪生调试的功能，通过CPU域的功能验证、gdb单步调试、printf数值打印来定位算子的实现逻辑问题。本样例仅展示了可能会出现的场景，便于演示定位步骤。实际使用过程中，请根据代码情况进行调试。

步骤1 进行CPU域的功能验证，观察是否有日志报错。

参考**8.2.3 运行验证算子工程**章节，编写CPU侧的运行验证代码，并进行运行验证。得到CPU域的精度比对结果如下：


```
md5sum:
45e17ee4c068a655be2af4d8c3a1f191  output/golden.bin
5d6e1aec686b28bd3839dbcd5caaa8b2  output/output_y.bin
```

可以看出CPU域的精度比对也存在不一致的问题，然后观察打屏日志中是否有报错信息，可搜索关键词"failed"。比如，下图的报错示例指示，错误出现在代码中调用LeakyRelu接口的地方。

```
leakyrelu_custom_cpu: /usr/local/Ascend/CANN-7.0/x86_64-linux/tikcpp/tikcfw/interface/
kernel_operator_vec_binary_scalar_intf.h:447: void AscendC::LeakyRelu(const AscendC::LocalTensor<T>&,
const AscendC::LocalTensor<T>&, const T&, const int32_t&) [with T = float16::Fp16T; int32_t = int]:
Assertion `false && "check vlrelu instr failed"' failed
```

通过上述报错日志，一般只能定位到报错的代码行，无法明确具体错误，接下来需要通过gdb调试的方式或者printf打印的方式进一步精确定位。

步骤2 gdb调试。下面的样例展示了拉起leakyrelu算子CPU侧运行程序的样例，该样例程序会直接抛出异常，直接gdb运行，查看调用栈信息分析定位即可。其他场景下您可以使用gdb打断点等基本操作进行调试。使用gdb调试Ascend C程序的详细内容请参考**10.2 CPU域调试**。

1. 使用gdb拉起待调试程序，进入gdb界面进行debug。
   ```
   gdb leakyrelu_custom_cpu
   ```
2. 单独调试一个子进程。
   ```
   (gdb) set follow-fork-mode child
   ```
3. 运行程序。
   ```
   (gdb) r
   ```
4. 通过bt查看程序调用栈。
   ```
   (gdb) bt
   ```
5. 查看具体层的堆栈信息，打印具体变量的值。本示例中，打印了tileLength为1024，该程序中表示需要处理1024个half类型的数，大小为1024*sizeof(half)=2048字节；输入Tensor xLocal的值，其中dataLen表示LocalTensor的size大小为1024字节，只能计算1024字节的数据。可以看出两者的长度不匹配，由此可以定位问题。
   ```
   (gdb) f 5
   #5  0x000055555555d364 in KernelLeakyRelu::Compute (this=0x7fffffffd7d0, progress=0) at /root/
   AscendC_DemoCode-master/precision-error/vector/leakyrelu_custom.cpp:59
   59              LeakyRelu(yLocal, xLocal, scalar, tileLength);
   (gdb) p tileLength
   $1 = 1024
   (gdb) p xLocal
   $1 = {<AscendC::BaseTensor<float16::Fp16T>> = {<No data fields>}, address_ = {logicPos = 9 '\t',
   bufferHandle = 0x7fffffffd930 "\003\005\377\377", dataLen = 1024,bufferAddr = 0,absAddr = ...}
   ```

步骤3 printf打印。在合适的位置增加变量打印。样例代码如下：

```
printf("xLocal size: %d\n", xLocal.GetSize());
printf("tileLength: %d\n", tileLength);
```

可以看到有如下打屏日志输出，打印了tileLength为1024，该程序中表示需要处理1024个half类型的数；输入Tensor xLocal的size大小，为512，表示只能计算512个half类型的数。可以看出两者的长度不匹配，由此可以定位问题。

```
xLocal size: 512
tileLength: 1024
```

----结束

15.2 运行验证时 AllocTensor/FreeTensor 失败

现象描述

通过NPU进行核函数的运行验证时，出现挂死现象；通过CPU进行核函数的运行验证时，出现AllocTensor/FreeTensor失败的报错，日志报错和调用栈打印如下：

```
[ERROR][Core_0][/usr/local/Ascend/latest/x86_64-linux/tikcpp/tikcfw/interface/kernel_tpipe.h:730]
[AllocEventID][321678] current size is 4, max buffer number in same queue position is 4
[ERROR][CORE_0][pid 321674] error happened! =========
SIGABRT Signal (Abort Signal from abort) catched, backtrace info:
[#0] 0x000000000001e7c0: handler(int) at /usr/local/Ascend/latest/tools/tikicpulib/lib/include/
kern_fwk.h:105
[#1] 0x0000000000017c4f: signed char AscendC::TPipe::AllocEventID<(AscendC::HardEvent)5>() at /usr/
local/Ascend/latest/x86_64-linux/tikcpp/tikcfw/interface/kernel_tpipe.h:733
[#2] 0x000000000001426d: AscendC::TQueBind<(AscendC::TPosition)0, (AscendC::TPosition)9, 4,
0>::FreeBuffer(unsigned char*) at /usr/local/Ascend/latest/x86_64-linux/tikcpp/tikcfw/interface/
kernel_tpipe.h:1217
[#3] 0x0000000000011058: void AscendC::TQueBind<(AscendC::TPosition)0, (AscendC::TPosition)9, 4,
0>::FreeTensor<float16::Fp16T>(AscendC::LocalTensor<float16::Fp16T>&) at /usr/local/Ascend/latest/x86_64-
linux/tikcpp/tikcfw/interface/kernel_tpipe.h:1237
[#4] 0x000000000000dfde: KernelAdd::Compute(int) at /home/xxxx/xxxx.cpp:59
[#5] 0x000000000000dd1c: KernelAdd::Process() at /home/xxxx/xxxx.cpp:37 (discriminator 2)
...
```

问题根因

根据日志信息“current size is 4, **max buffer number in same queue position** is 4”可以明确该问题是因为同一个TPosition上QUE Buffer的数量超出限制导致。

同一个TPosition上QUE Buffer的数量根据AI处理器型号的不同，有数量约束。申请Buffer时，需要满足该约束。

Atlas 训练系列产品、Atlas推理系列产品AI Core不超过4块。

Atlas A2训练系列产品/Atlas 800I A2推理产品不超过8块。

不满足该约束，在后续使用AllocTensor/FreeTensor可能会出现分配资源失败。比如：

```
TQue<TPosition::VECIN, 1> que0;
TQue<TPosition::VECIN, 1> que1;
// 不建议：
// 比如，算子有6个输入，需要申请6块buffer
// 通过2个队列为其申请内存，分别为que0、que1分配3块,申请VECIN position上的buffer总数为6
// 针对Atlas 训练系列产品、Atlas推理系列产品AI Core同一个TPosition上QUE Buffer的数量限制为4，超出该限制，在后续使用AllocTensor/FreeTensor可能会出现分配资源失败。
pipe.InitBuffer(que0, 3, len);
pipe.InitBuffer(que1, 3, len);
```

处理步骤

如果确实有多块buffer使用，可以将多个buffer合并到一块buffer，通过偏移使用。样例如下：

```
// 此时建议通过以下方法解决：
// 如果确实有多块buffer使用，可以将多个buffer合并到一块buffer，通过偏移使用
pipe.InitBuffer(que0, 1, len * 3)
pipe.Initbuffer(que1, 1, len * 3)
/*
 * 分配出3块内存大小的LocalTensor，local1的地址为que0中buffer的起始地址，
 * local2的地址为local1的地址偏移len后的地址，local3的地址为local1的地址偏移
```



```
 * len * 2的地址
 */
int32_t offset1 = len;
int32_t offset2 = len * 2;
LocalTensor<T> local1 = que0.AllocTensor<T>();
LocalTensor<T> local2 = local1[offset1];
LocalTensor<T> local3 = local1[offset2];
```

15.3 kernel 侧获取 Tiling 信息不正确

现象描述

通过算子在kernel侧实现代码中添加PRINTF打印发现kernel侧获取的Tiling信息不正确。

比如下文样例，增加的打印代码如下：

```
PRINTF("tiling_data.totalLength: %d tiling_data.tileNum: %d.\n", tiling_data.totalLength,
tiling_data.tileNum);
```

打印的Tiling数据如下，全为0：

```
tiling_data.totalLength: 0 tiling_data.tileNum: 0.
```

问题根因

kernel侧获取Tiling信息不正确的原因一般有以下两种：

- host侧计算Tiling的逻辑不正确
- kernel侧核函数的参数未按照正确顺序填写

处理步骤

步骤1 参考如下示例，打印TilingData的数据，确认host侧序列化保存的TilingData是否正确。如果此时打印值有误，说明Tiling的计算逻辑可能不正确，需要进一步检查host侧Tiling实现代码，排查计算逻辑是否有误。

```
std::out<<*reinterpret_cast<uint32_t *>(context->GetRawTilingData()->GetData())<<std::endl; //按照实际数据类型打印TilingData第一个参数值，如需确认其他值，取值指针向后偏移即可
```

步骤2 如果上一步骤中打印的TilingData正确，需要排查kernel侧核函数的参数是否按照正确顺序填写。

使用msOpGen工具创建算子工程，并基于工程进行kernel侧算子开发时，核函数的定义模板已通过msOpGen工具自动生成，样例如下所示。参数按照 “输入、输出、workspace、tiling”的顺序排布。请检查是否调整过参数顺序导致和正确顺序不一致。

```
#include "kernel_operator.h"
extern "C" __global__ __aicore__ void add_custom(GM_ADDR x, GM_ADDR y, GM_ADDR z, GM_ADDR
workspace, GM_ADDR tiling) {
    GET_TILING_DATA(tiling_data, tiling);// 获取Tiling参数
    // TODO: user kernel impl
}
```

----结束

15.4 Kernel 编译时报错“error: out of jump/jumpc imm range”

现象描述

使用工程化算子开发方式，基于自定义算子工程进行算子开发。编译算子时失败，报如下错误：

```
[ERROR] [ascendxxxx] PowerCustom_88a695f03edfbc0af76b9eaae9e4556c error: out of jump/jumpc imm range
```

问题根因

该编译错误的原因是算子kernel代码过大，导致在编译时跳转指令跳转的偏移值超过了限定的大小(int16_t的数据范围)，可通过添加编译选项“-mllvm -cce-aicore-jump-expand=true”通过间接跳转的方式来避免该问题，让编译器能够正常编译。

处理步骤

步骤1 在kernel侧的CMakeLists中通过add_ops_compile_options针对报错算子添加编译选项“-mllvm -cce-aicore-jump-expand=true”，示例如下：

```
add_ops_compile_options(PowerCustom OPTIONS -mllvm -cce-aicore-jump-expand=true)
```

add_ops_compile_options的具体使用方法请参考**支持自定义编译选项**。

步骤2 重新编译该算子。正常编译无报错。

----结束

15.5 使用跨版本的自定义算子包时，含有 Matmul 高阶 API 的算子存在编译或执行报错

现象描述

1. 基于CANN-7.2及之前版本（<=7.2）的CANN开发套件包，编译含有Matmul高阶API的自定义算子包，将编译后的自定义算子包安装至CANN-7.3及之后版本（>=7.3）的CANN包环境，然后对该含有Matmul高阶API的算子，执行图模式在线编译时，报如下错误：
   ```
   res = struct.unpack_from(fmt_str, tiling_data, offset + unpack_size)
   struct.error: unpack_from requires a buffer of at least 52 bytes for unpacking 4 bytes at offset 48
   ```
2. 基于CANN-7.2及之前版本（<=7.2）的CANN开发套件包，编译**sample样例仓**中含有Matmul高阶API的算子，例如**MatmulLeakyReluCustomSample**，将编译后的自定义算子包安装至CANN-7.3及之后版本（>=7.3）的CANN包环境，然后对该含有Matmul高阶API的算子，执行单算子API的调用时，报如下错误：
   ```
   ERROR：acl executable run faile! please check your project!
   ```

问题根因

该错误的原因是编译自定义算子包的软件版本过老，可通过更新自定义算子包编译环境上的CANN开发套件包版本，然后重新编译和部署自定义算子包，来避免出现该问题。

处理步骤

步骤1 查看自定义算子包编译时使用的CANN开发套件包版本号，示例如下：

```
cd ${CANN包安装路径}
cat version.cfg

# version: 1.0
runtime_running_version=[7.2.T11.0.B218:8.0.RC2.alpha001]
runtime_upgrade_version=[7.2.T11.0.B218:8.0.RC2.alpha001]
runtime_installed_version=[7.2.T11.0.B218:8.0.RC2.alpha001]
```

步骤2 基于CANN-7.3及之后版本（>=7.3）的CANN开发套件包，重新编译该自定义算子包。部署编译生成的自定义算子包后，正常编译或者执行算子，无报错。重新编译和部署自定义算子包的具体方法可参考**9.4 编译部署**。

----结束

15.6 含有 Matmul 高阶 API 的算子精度问题

本节针对含有Matmul高阶API的算子，为排查在开发过程中遇到的精度问题，是否为算子中Matmul高阶API调用方式导致，提供初步的问题定界和定位指导。如未特殊说明，下面均以Atlas A2训练系列产品/Atlas 800I A2推理产品上的案例为例。

主要介绍根据如下六个步骤，开展具体排查：

1. CPU域调试，观察报错信息；
2. Matmul Tiling是否有修改，修改是否合理；
3. 算子隐藏Vector计算，仅调用Matmul API，算子功能是否正确；
4. 单核执行，算子功能是否正确；
5. 排查Matmul API的使用是否正确；
6. 用于算子调测的golden脚本是否正确。

步骤1 CPU域调试，观察报错信息

在完成算子代码的开发后，优先通过**CPU调测工程**，调试算子的功能。在CPU域调试时，若编译或执行报错，日志中一般会有明显的报错信息。根据报错信息的提示内容，通常可以快速定位到问题所对应的代码位置。这种方法尤其对DataCopy参数设置错误导致的地址越界、算子Tiling参数设置不正确、其他内存越界访问等基础参数的使用问题，可以快速定位到具体原因。

- 案例：

  以下为matmul算子核函数的代码片段。

```
extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c,
GM_ADDR workspace, GM_ADDR tilingGm)
{
    using A_T = half;
    using B_T = half;
    using C_T = float;

    AscendC::TPipe pipe;
    TCubeTiling tiling;
    CopyTiling(&tiling, tilingGm);

    AscendC::GlobalTensor<A_T> aGlobal;
    AscendC::GlobalTensor<B_T> bGlobal;
    AscendC::GlobalTensor<C_T> cGlobal;
    aGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ A_T *>(a), tiling.M * tiling.Ka);
    bGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ B_T *>(b), tiling.Ka * tiling.N);
```



```
    cGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ C_T *>(c), tiling.M * tiling.N);

    int offsetA = 0;
    int offsetB = 0;
    int offsetC = 0;
    bool isTransA = false;
    bool isTransB = true;

    int tailM = 0;
    int tailN = 0;
    CalcGMOffset(GetBlockIdx(), tiling, offsetA, offsetB, offsetC, tailM, tailN, isTransA, isTransB);

    auto gmA = aGlobal[offsetA];
    auto gmB = bGlobal[offsetB];
    auto gmC = cGlobal[offsetC];

    Matmul<MatmulType<AscendC::TPosition::GM, CubeFormat::ND, A_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, B_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, C_T>> mm;
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm, &tiling);

    mm.SetTensorA(gmA, isTransA);
    mm.SetTensorB(gmB, isTransB);
    mm.SetTail(tailM, tailN);
    mm.IterateAll(gmC);
    mm.End();
}
```

```
[ASSERT] /home/cma/Ascend/CANN-7.5/x86_64-linux/ascendc/include/highlevel_api/lib/matmul/
matmul_client.h:268: Assertion `isTransposeB <= B_TYPE::isTrans && "It is not allowed to do B
transpose when matmul B transpose is not defined."'
[ASSERT] /home/cma/Ascend/CANN-7.5/x86_64-linux/ascendc/include/highlevel_api/lib/matmul/
matmul_client.h:268: Assertion `isTransposeB <= B_TYPE::isTrans && "It is not allowed to do B
transpose when matmul B transpose is not defined."'
[ERROR][AIV_1][pid 1010818] error happened! ==========
SIGABRT Signal (Abort Signal from abort) catched, backtrace info:
[#0] 0x0000000000009cd2: Handler(int) at /home/cma/Ascend/latest/tools/tikicpulib/lib/include/
kern_fwk.h:106
[#1] 0x00000000000060b7: main at /home/cma/samples/Precision_Check_Guide/samples-master/
operator/MatmulCustomSample/KernelLaunch/MatmulInvocationNeo-cpu_check/main.cpp:50
(discriminator 126)
[#2] 0x00000000000086de: _start at ??:?

[ERROR][AIV_0][pid 1010817] error happened! ==========
SIGABRT Signal (Abort Signal from abort) catched, backtrace info:
[#0] 0x0000000000009cd2: Handler(int) at /home/cma/Ascend/latest/tools/tikicpulib/lib/include/
kern_fwk.h:106
[#1] 0x00000000000060b7: main at /home/cma/samples/Precision_Check_Guide/samples-master/
operator/MatmulCustomSample/KernelLaunch/MatmulInvocationNeo-cpu_check/main.cpp:50
(discriminator 126)
[#2] 0x00000000000086de: _start at ??:?
```

本案例中的算子有精度问题，于是使用CPU调测该算子功能，CPU运行后，根据报错信息提示的矩阵B的transpose未定义，查看矩阵B的相关设置代码，发现Matmul对象定义时未设置矩阵B的B_TYPE::isTrans，而SetTensorB接口设置了isTransB = true，导致执行报错。所以，此问题的根因为SetTensorB设置的isTransB值与B_TYPE不符。

步骤2 Matmul Tiling是否有修改，修改是否合理

一般含有Matmul的算子Tiling实现中，Matmul Tiling的结构体TCubeTiling，通过调用GetTiling接口返回，这时这组Tiling值是合法的。某些情况下，用户自定义了一组TCubeTiling参数值，或者，基于GetTiling接口返回的TCubeTiling，自行修改了其中的部分Tiling值，这样的修改需要满足参数间的制约条件。

为获取所有Tiling参数值，需要打印Tiling参数相关的日志。设置日志环境变量，获取MatmulTiling参数值。设置环境变量的命令如下：


```
export ASCEND_GLOBAL_LOG_LEVEL=1
export ASCEND_SLOG_PRINT_TO_STDOUT=1
```

在日志中搜索“MatmulTiling”关键字，参照TCubeTiling约束条件，检查Tiling取值是否合法。若不满足某条约束条件，需要修改对应的相关参数，使该组TCubeTiling参数值均合法。

```
root@ubuntu:/home/cma/samples/Precision_Check_Guide/samples-master/operator/
MatmulLeakyReluCustomSample/KernelLaunch/MatmulLeakyReluInvocation-golden# cat test_tiling2.log |
grep MatmulTiling
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.864
[matmul_tiling_base.cpp:697][PrintTilingDataInfo] MatmulTiling: M              = 1024
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.870
[matmul_tiling_base.cpp:698][PrintTilingDataInfo] MatmulTiling: N              = 640
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.873
[matmul_tiling_base.cpp:699][PrintTilingDataInfo] MatmulTiling: Ka             = 256
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.876
[matmul_tiling_base.cpp:700][PrintTilingDataInfo] MatmulTiling: Kb             = 256
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.879
[matmul_tiling_base.cpp:701][PrintTilingDataInfo] MatmulTiling: singleCoreM    = 512
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.882
[matmul_tiling_base.cpp:702][PrintTilingDataInfo] MatmulTiling: singleCoreN    = 640
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.884
[matmul_tiling_base.cpp:703][PrintTilingDataInfo] MatmulTiling: singleCoreK    = 256
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.887
[matmul_tiling_base.cpp:704][PrintTilingDataInfo] MatmulTiling: baseM          = 256
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.890
[matmul_tiling_base.cpp:705][PrintTilingDataInfo] MatmulTiling: baseN          = 128
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.893
[matmul_tiling_base.cpp:706][PrintTilingDataInfo] MatmulTiling: baseK          = 64
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.896
[matmul_tiling_base.cpp:707][PrintTilingDataInfo] MatmulTiling: depthA1        = 8
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.899
[matmul_tiling_base.cpp:708][PrintTilingDataInfo] MatmulTiling: depthB1        = 2
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.902
[matmul_tiling_base.cpp:709][PrintTilingDataInfo] MatmulTiling: depthAL1CacheUB     = 0
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.905
[matmul_tiling_base.cpp:710][PrintTilingDataInfo] MatmulTiling: depthBL1CacheUB     = 0
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.908
[matmul_tiling_base.cpp:711][PrintTilingDataInfo] MatmulTiling: stepM          = 2
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.912
[matmul_tiling_base.cpp:712][PrintTilingDataInfo] MatmulTiling: stepN          = 1
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.915
[matmul_tiling_base.cpp:713][PrintTilingDataInfo] MatmulTiling: isBias         = 1
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.917
[matmul_tiling_base.cpp:714][PrintTilingDataInfo] MatmulTiling: transLength    = 0
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.920
[matmul_tiling_base.cpp:715][PrintTilingDataInfo] MatmulTiling: iterateOrder   = 0
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.923
[matmul_tiling_base.cpp:716][PrintTilingDataInfo] MatmulTiling: shareMode      = 0
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.926
[matmul_tiling_base.cpp:717][PrintTilingDataInfo] MatmulTiling: usedL1Size     = 295424
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.929
[matmul_tiling_base.cpp:718][PrintTilingDataInfo] MatmulTiling: usedL0CSize    = 131072
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.932
[matmul_tiling_base.cpp:719][PrintTilingDataInfo] MatmulTiling: usedUBSize     = 0
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.935
[matmul_tiling_base.cpp:720][PrintTilingDataInfo] MatmulTiling: batchM         = 1
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.938
[matmul_tiling_base.cpp:721][PrintTilingDataInfo] MatmulTiling: batchN         = 1
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.941
[matmul_tiling_base.cpp:722][PrintTilingDataInfo] MatmulTiling: singleBatchM   = 1
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.943
[matmul_tiling_base.cpp:723][PrintTilingDataInfo] MatmulTiling: singleBatchN   = 1
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.946
[matmul_tiling_base.cpp:724][PrintTilingDataInfo] MatmulTiling: stepKa         = 4
[INFO] ASCENDCKERNEL(1202803,ascendc_kernels_bbit):2024-10-12-08:53:59.636.949
[matmul_tiling_base.cpp:725][PrintTilingDataInfo] MatmulTiling: stepKb         = 1
```


步骤3 算子隐藏Vector计算，仅调用Matmul API，算子功能是否正确

融合算子的代码既包含Matmul API，也包含Vector计算API。通过在算子代码中删除Vector计算API，只保留Matmul API，快速定界是否为Matmul API的错误使用，导致了融合算子的精度问题。具体排查过程为，同步修改算子代码逻辑和golden脚本，删除Vector计算的代码，完成适配修改后，CPU域或NPU域上执行算子，观察算子结果是否正确。若算子结果正确，说明代码中Matmul API的使用方式正确，定位算子精度问题需要继续排查Vector计算；反之，若算子结果不正确，需要继续排查Matmul API的使用是否正确。

- 案例：

  以融合算子matmul_leakyrelu为例，执行算子后，出现如下图所示的精度问题。

```
data index: 000195, expected: -0.693000019, actual: -69.300003052, rdiff: -99.000000
data index: 000196, expected: -0.209000006, actual: -20.899999619, rdiff: -99.000000
data index: 000197, expected: -0.517000020, actual: -51.700000763, rdiff: -99.000000
data index: 000200, expected: -0.193000004, actual: -19.300001144, rdiff: -99.000000
data index: 000202, expected: -0.684000015, actual: -68.400001526, rdiff: -99.000000
data index: 000204, expected: -0.422000021, actual: -42.200000763, rdiff: -98.999992
data index: 000209, expected: -0.109000005, actual: -10.900000572, rdiff: -99.000000
error ratio: 0.4517, tolrence: 0.0001
[ERROR] result error
```

  修改算子代码，注释屏蔽LeakyRelu API计算，同时，需要适配修改相应的内存分配或涉及的同步等操作；然后，注释golden脚本中LeakyRelu计算，具体修改示例如下。

  以下代码为算子核函数的代码片段。

```
template <typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::Process(AscendC::TPipe
*pipe)
{
    uint32_t computeRound = 0;

    matmulObj.SetTensorA(aGlobal);
    matmulObj.SetTensorB(bGlobal);
    matmulObj.SetBias(biasGlobal);
    while (matmulObj.template Iterate<true>()) {
        MatmulCompute();
        // LeakyReluCompute(); // 注释LeakyReluCompute Vector计算
        CopyOut(computeRound);
        computeRound++;
    }
    matmulObj.End();
}

template <typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::MatmulCompute()
{
    reluOutLocal = reluOutQueue_.AllocTensor<cType>();
    matmulObj.template GetTensorC<true>(reluOutLocal, false, true);
    reluOutQueue_.EnQue(reluOutLocal); // 将LeakyReluCompute()接口里的reluOutLocal结果输出提前
到这里
}

template <typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::LeakyReluCompute()
{
    LeakyRelu(reluOutLocal, reluOutLocal, (cType)0.1, tiling.baseM * tiling.baseN);
    reluOutQueue_.EnQue(reluOutLocal);
}

template <typename aType, typename bType, typename cType, typename biasType>
__aicore__ inline void MatmulLeakyKernel<aType, bType, cType, biasType>::CopyOut(uint32_t count)
{
    reluOutQueue_.DeQue<cType>();
    const uint32_t roundM = tiling.singleCoreM / tiling.baseM;
```



```
    const uint32_t roundN = tiling.singleCoreN / tiling.baseN;
    uint32_t startOffset = (count % roundM * tiling.baseM * tiling.N + count / roundM * tiling.baseN);
    AscendC::DataCopyParams copyParam = {(uint16_t)tiling.baseM, (uint16_t)(tiling.baseN *
sizeof(cType) / AscendC::DEFAULT_C0_SIZE), 0,
                                         (uint16_t)((tiling.N - tiling.baseN) * sizeof(cType) /
AscendC::DEFAULT_C0_SIZE)};
    DataCopy(cGlobal[startOffset], reluOutLocal, copyParam);
    reluOutQueue_.FreeTensor(reluOutLocal);
}
```

以下代码为golden生成脚本的代码片段。

```
def gen_golden_data():
    M = 1024
    N = 640
    K = 256

    input_a = np.random.randint(-10, 10, [M, K]).astype(np.float16)
    input_b = np.random.randint(-10, 10, [K, N]).astype(np.float16)
    input_bias = np.random.randint(-10, 10, [N]).astype(np.float32)
    alpha = 0.001
    golden = (np.matmul(input_a.astype(np.float32), input_b.astype(np.float32)) +
input_bias).astype(np.float32)
    # golden = np.where(golden >= 0, golden, golden * alpha) # 与kernel保持一致，golden生成也需注
释相应的LeakyRelu计算
    os.system("mkdir -p input")
    os.system("mkdir -p output")
    input_a.tofile("./input/x1_gm.bin")
    input_b.tofile("./input/x2_gm.bin")
    input_bias.tofile("./input/bias.bin")
    golden.tofile("./output/golden.bin")
```

删除LeakyRelu计算后，执行算子，算子结果比对正确。如此可确定，算子代码中已正确使用Matmul API，并得到了正确的Matmul API计算结果，需要继续定位LeakyReluCompute函数内LeakyRelu接口的使用。

```
-- Installing: /home/cma/samples/Precision_Check_Guide/samples-master/operator/
MatmulLeakyReluCustomSample/KernelLaunch/MatmulLeakyReluInvocation_cube_vec/out/bin/
ascendc_kernels_bbit
8901941eee314bcd64d24ff5f8d21247  output/golden.bin
8901941eee314bcd64d24ff5f8d21247  output/output.bin
error ratio: 0.0000, tolrence: 0.0001
test pass
```

步骤4 单核执行，算子功能是否正确

验证单核场景下，算子的功能是否正确，可以帮助快速定界是Matmul API的计算结果不符合预期，还是算子代码中错误调用Matmul API导致。由于Matmul API内部实现管理的是单核的计算逻辑，所以单核上的计算结果正确，而多核的计算结果错误的情况，说明单核上的Matmul API的使用及计算正确，这时需要排查与多核切分相关的代码逻辑是否正确，比如多核的输入和输出地址偏移是否正确，每个核上的尾块地址设置是否正确。如果验证单核场景下，算子精度不正确，需要排查Matmul API的使用是否正确，具体可参考**步骤5**。

提示，包含Matmul的算子的Tiling实现中，Matmul的多核Tiling需要使用MultiCoreMatmulTiling构造多核Tiling对象，通过SetDim接口设置Matmul计算所用的核数。注意：这里设置的核数为Matmul计算所用的核数，仅在多核场景下设置，用于计算tiling参数。如下两个案例为MIX模式的算子，SetDim的设置规则请参考**MIX场景核数设置规则**。

- 案例1：多核切分场景，输出地址偏移不正确

  以M=512, N=1024, K=512的Matmul为例，MIX模式的算子代码中设置AIC核数为4，AIV核数为8，因为本案例以分离架构为例，所以SetDim设置为AIV核数的取值8。多核场景下执行该算子，计算结果精度错误。

  以下为算子Tiling计算的代码片段。


```
uint8_t *GenerateTiling(const char *socVersion)
{
    int M = 512;
    int N = 1024;
    int K = 512;

    TPosition leftPosition = TPosition::GM;
    CubeFormat leftFormat = CubeFormat::ND;
    DataType leftDtype = DataType::DT_FLOAT16;
    bool isTransA = false;

    TPosition rightPosition = TPosition::GM;
    CubeFormat rightFormat = CubeFormat::ND;
    DataType rightDtype = DataType::DT_FLOAT16;
    bool isTransB = false;

    TPosition resultPosition = TPosition::GM;
    CubeFormat resultFormat = CubeFormat::ND;
    DataType resultDtype = DataType::DT_FLOAT;

    bool isBias = false;

    int usedCoreNum = 8;
    int32_t baseM = 128;
    int32_t baseN = 256;

    optiling::TCubeTiling tilingData;
    auto ascendcPlatform = platform_ascendc::PlatformAscendCManager::GetInstance(socVersion);
    MultiCoreMatmulTiling tilingApi(*ascendcPlatform);

    tilingApi.SetDim(usedCoreNum); // 设置为AIV核数8
    tilingApi.SetAType(leftPosition, leftFormat, leftDtype, isTransA);
    tilingApi.SetBType(rightPosition, rightFormat, rightDtype, isTransB);
    tilingApi.SetCType(resultPosition, resultFormat, resultDtype);

    tilingApi.SetOrgShape(M, N, K);
    tilingApi.SetShape(M, N, K);
    tilingApi.SetFixSplit(baseM, baseN, -1);
    tilingApi.SetBias(isBias);
    tilingApi.SetBufferSpace(-1, -1, -1);

    int64_t res = tilingApi.GetTiling(tilingData);
    if (res == -1) {
        std::cout << "gen tiling failed" << std::endl;
    }
    return GetTilingBuf(&tilingData);
}
```

以下为算子核函数的代码片段。

```
__aicore__ inline void CalcGMOffset(int blockIdx, const TCubeTiling &tiling, int &offsetA, int &offsetB,
int &offsetC,
                                    int &tailM, int &tailN, bool isTransA, bool isTransB)
{
    uint32_t mSingleBlocks = Ceiling(tiling.M, tiling.singleCoreM);
    uint32_t mCoreIndx = blockIdx % mSingleBlocks;
    uint32_t nCoreIndx = blockIdx / mSingleBlocks;

    offsetA = mCoreIndx * tiling.Ka * tiling.singleCoreM;
    if (isTransA) {
        offsetA = mCoreIndx * tiling.singleCoreM;
    }
    offsetB = nCoreIndx * tiling.singleCoreN;
    if (isTransB) {
        offsetB = nCoreIndx * tiling.Kb * tiling.singleCoreN;
    }
    offsetC = mCoreIndx * tiling.singleCoreN * tiling.singleCoreM + nCoreIndx * tiling.singleCoreN; //此处
的tiling.singleCoreN参数错误，应为tiling.N

    tailM = tiling.M - mCoreIndx * tiling.singleCoreM;
    tailM = tailM < tiling.singleCoreM ? tailM : tiling.singleCoreM;
```



```
    tailN = tiling.N - nCoreIndx * tiling.singleCoreN;
    tailN = tailN < tiling.singleCoreN ? tailN : tiling.singleCoreN;
}

extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c,
GM_ADDR workspace,
                                        GM_ADDR tilingGm)
{
    using A_T = half;
    using B_T = half;
    using C_T = float;

    AscendC::TPipe pipe;
    TCubeTiling tiling;
    CopyTiling(&tiling, tilingGm);

    AscendC::GlobalTensor<A_T> aGlobal;
    AscendC::GlobalTensor<B_T> bGlobal;
    AscendC::GlobalTensor<C_T> cGlobal;
    aGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ A_T *>(a), tiling.M * tiling.Ka);
    bGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ B_T *>(b), tiling.Ka * tiling.N);
    cGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ C_T *>(c), tiling.M * tiling.N);

    int offsetA = 0;
    int offsetB = 0;
    int offsetC = 0;
    bool isTransA = false;
    bool isTransB = false;

    int tailM = 0;
    int tailN = 0;
    CalcGMOffset(GetBlockIdx(), tiling, offsetA, offsetB, offsetC, tailM, tailN, isTransA, isTransB);

    auto gmA = aGlobal[offsetA];
    auto gmB = bGlobal[offsetB];
    auto gmC = cGlobal[offsetC];

    Matmul<MatmulType<AscendC::TPosition::GM, CubeFormat::ND, A_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, B_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, C_T>> mm;
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm, &tiling);

    mm.SetTensorA(gmA, isTransA);
    mm.SetTensorB(gmB, isTransB);
    mm.SetTail(tailM, tailN);
    mm.IterateAll(gmC);
    mm.End();
}
```

执行算子，精度校验失败：

```
data index: 000609, expected: 12979.000000000, actual: 0.000000000, rdiff: 1.000000
data index: 000610, expected: 12931.000000000, actual: 0.000000000, rdiff: 1.000000
data index: 000611, expected: 13120.000000000, actual: 0.000000000, rdiff: 1.000000
data index: 000612, expected: 12275.000000000, actual: 0.000000000, rdiff: 1.000000
error ratio: 0.8750, tolrence: 0.0001
[ERROR] result error
```

修改测试脚本和算子Tiling的代码，通过验证单核上的算子执行结果，快速定界。具体如下：

修改算子调测代码，为只启动单核，CPU调测代码中将ICPU_RUN_KF宏接口中的blockDim设置为1（AIC AIV的组合数）；算子的TIling实现中，设置单核场景，AIC核数为1，AIV核数为2，SetDim设置为AIV核数的取值2。如下代码所示。

以下为调测脚本的代码片段。

```
uint32_t blockDim = 1;
ICPU_RUN_KF(matmul_custom, blockDim, a, b, c, workspace, tiling);
```

以下为算子Tiling计算的代码片段。


```
int usedCoreNum = 2;
tilingApi.SetDim(usedCoreNum);
```

修改为单核场景后，执行算子：

```
-- Installing: /home/cma/samples/Precision_Check_Guide/samples-master/operator/
MatmulCustomSample/KernelLaunch/MatmulInvocationNeo-muticore/out/bin/ascendc_kernels_bbit
efaf4dc1e484bc3778cac65f56244e59  output/golden.bin
efaf4dc1e484bc3778cac65f56244e59  output/output.bin
error ratio: 0.0000, tolrence: 0.0001
test pass
```

从上述比对结果可看出，单核验证结果正确，此时可以定界导致精度的问题为多核相关的问题。

首先排查多核切分后的输入和输出地址偏移。分析CalcGMOffset函数，定位到矩阵C的偏移地址offsetC计算错误，正确的偏移应该是mCoreIndx * tiling.N * tiling.singleCoreM + nCoreIndx * tiling.singleCoreN。将offsetC修改为正确的偏移地址后，执行算子，结果比对正确。

提示，在上述单核场景的修改验证中，AIC核数为1，AIV核数为2；若想进一步验证，不引入任何多核切分，AIC核数和AIV核数均修改为1，代码修改示例如下：

- 在核函数中REGIST_MATMUL_OBJ接口后，利用判断代码，BlockIdx不为0的AIV核退出。

  以下为算子核函数的代码片段。

```
extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c,
GM_ADDR workspace,
                                        GM_ADDR tilingGm)
{
    using A_T = half;
    using B_T = half;
    using C_T = float;

    AscendC::TPipe pipe;
    TCubeTiling tiling;
    CopyTiling(&tiling, tilingGm);

    AscendC::GlobalTensor<A_T> aGlobal;
    AscendC::GlobalTensor<B_T> bGlobal;
    AscendC::GlobalTensor<C_T> cGlobal;
    aGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ A_T *>(a), tiling.M * tiling.Ka);
    bGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ B_T *>(b), tiling.Ka * tiling.N);
    cGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ C_T *>(c), tiling.M * tiling.N);

    int offsetA = 0;
    int offsetB = 0;
    int offsetC = 0;
    bool isTransA = false;
    bool isTransB = false;

    int tailM = 0;
    int tailN = 0;
    CalcGMOffset(GetBlockIdx(), tiling, offsetA, offsetB, offsetC, tailM, tailN, isTransA, isTransB);

    auto gmA = aGlobal[offsetA];
    auto gmB = bGlobal[offsetB];
    auto gmC = cGlobal[offsetC];

    Matmul<MatmulType<AscendC::TPosition::GM, CubeFormat::ND, A_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, B_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, C_T>> mm;
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm, &tiling);
    if ASCEND_IS_AIV {
        if (GetBlockIdx() != 0) {
            return;
        }
    }
```



```
    mm.SetTensorA(gmA, isTransA);
    mm.SetTensorB(gmB, isTransB);
    mm.SetTail(tailM, tailN);
    mm.IterateAll(gmC);
    mm.End();
}
```

- 算子调测脚本的ICPU_RUN_KF中blockDim和算子Tiling中SetDim的usedCoreNum均设置为1。

  以下为算子调测代码片段。

  ```
  uint32_t blockDim = 1;
  ICPU_RUN_KF(matmul_custom, blockDim, a, b, c, workspace, tiling);
  ```

  以下为算子Tiling计算的代码片段。

  ```
  int usedCoreNum = 1;
  tilingApi.SetDim(usedCoreNum);
  ```

- 案例2：尾块设置不正确

  多核场景下，当最后一个核的singleCoreM/singleCoreN/singleCoreK值与前面的核取值不同时，需要在最后一个核上，即尾核，调用SetTail接口，调整singleCoreM/singleCoreN/singleCoreK为实际尾核上的对应取值；若尾核未设置这些参数值，或者设置的参数值大小不正确，也会导致多核精度错误，单核精度正确。

  ```
  data index: 100254, expected: 13605.000000000, actual: 13137.000000000, rdiff: 0.034399
  data index: 101277, expected: 13268.000000000, actual: 13419.000000000, rdiff: 0.011381
  data index: 102300, expected: 13509.000000000, actual: 13114.000000000, rdiff: 0.029240
  data index: 103323, expected: 13526.000000000, actual: 13400.000000000, rdiff: 0.009315
  error ratio: 0.0010, tolrence: 0.0001
  [ERROR] result error
  ```

  以下为算子核函数的代码片段。

  ```
  __aicore__ inline void CalcGMOffset(int blockIdx, const TCubeTiling &tiling, int &offsetA, int &offsetB,
  int &offsetC,
                                      int &tailM, int &tailN, bool isTransA, bool isTransB)
  {
      uint32_t mSingleBlocks = Ceiling(tiling.M, tiling.singleCoreM);
      uint32_t mCoreIndx = blockIdx % mSingleBlocks;
      uint32_t nCoreIndx = blockIdx / mSingleBlocks;

      offsetA = mCoreIndx * tiling.Ka * tiling.singleCoreM;
      if (isTransA) {
          offsetA = mCoreIndx * tiling.singleCoreM;
      }
      offsetB = nCoreIndx * tiling.singleCoreN;
      if (isTransB) {
          offsetB = nCoreIndx * tiling.Kb * tiling.singleCoreN;
      }
      offsetC = mCoreIndx * tiling.N * tiling.singleCoreM + nCoreIndx * tiling.singleCoreN;

      // 尾核对应的M/N计算，此处为正确的计算方式
      tailM = tiling.M - mCoreIndx * tiling.singleCoreM;
      tailM = tailM < tiling.singleCoreM ? tailM : tiling.singleCoreM;

      tailN = tiling.N - nCoreIndx * tiling.singleCoreN;
      tailN = tailN < tiling.singleCoreN ? tailN : tiling.singleCoreN;
  }

  extern "C" __global__ __aicore__ void matmul_custom(GM_ADDR a, GM_ADDR b, GM_ADDR c,
  GM_ADDR workspace,
                                                      GM_ADDR tilingGm)
  {
      using A_T = half;
      using B_T = half;
      using C_T = float;

      AscendC::TPipe pipe;
      TCubeTiling tiling;
  ```

```
    CopyTiling(&tiling, tilingGm);

    AscendC::GlobalTensor<A_T> aGlobal;
    AscendC::GlobalTensor<B_T> bGlobal;
    AscendC::GlobalTensor<C_T> cGlobal;
    aGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ A_T *>(a), tiling.M * tiling.Ka);
    bGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ B_T *>(b), tiling.Ka * tiling.N);
    cGlobal.SetGlobalBuffer(reinterpret_cast<__gm__ C_T *>(c), tiling.M * tiling.N);

    int offsetA = 0;
    int offsetB = 0;
    int offsetC = 0;
    bool isTransA = false;
    bool isTransB = false;

    int tailM = 0;
    int tailN = 0;
    CalcGMOffset(GetBlockIdx(), tiling, offsetA, offsetB, offsetC, tailM, tailN, isTransA, isTransB);

    auto gmA = aGlobal[offsetA];
    auto gmB = bGlobal[offsetB];
    auto gmC = cGlobal[offsetC];

    Matmul<MatmulType<AscendC::TPosition::GM, CubeFormat::ND, A_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, B_T>,
        MatmulType<AscendC::TPosition::GM, CubeFormat::ND, C_T>> mm;
    REGIST_MATMUL_OBJ(&pipe, GetSysWorkSpacePtr(), mm, &tiling);

    mm.SetTensorA(gmA, isTransA);
    mm.SetTensorB(gmB, isTransB);
    // mm.SetTail(tailM, tailN); 尾核设置接口，若次处未更新尾块会导致单核精度正确，多核失败
    mm.IterateAll(gmC);
    mm.End();
}
```

步骤5 排查Matmul API的使用是否正确

经过上述步骤，可定界出是否为Matmul API使用问题。如果由于Matmul API使用错误导致了算子的精度问题，需要根据Matmul各接口的使用说明、约束条件等，检查接口的使用是否正确。

- 案例1：不支持的输入数据类型

  A矩阵、B矩阵和Bias的数据类型均设置为int8_t。由于Bias不支持int8_t类型，算子执行后精度比对失败。

  此类问题，应根据Matmul使用说明中支持的POSITION/CubeFormat/TYPE等信息进行排查。

- 案例2：未遵循接口约束条件

  在Matmul MDL模板下，调用IterateBatch接口，导致算子执行失败。这是由于不满足该接口的约束条件，IterateBatch接口仅支持Norm模板。

  此类问题，应仔细阅读Matmul各接口中的约束条件，并排查算子实现使用的相关接口，是否满足对应接口的约束条件。

- 案例3：未遵循模板约束条件

  在使能doMTE2Preload预加载模板时，若K方向非全载，不满足模板约束条件，则会导致精度比对失败。

  除了满足函数接口约束条件外，也需要满足模板参数相应的约束条件，排查模板参数的使用。

步骤6 用于算子调测的golden脚本是否正确

算子的golden生成脚本，是根据自定义算子的功能逻辑，自行实现的、用于比对算子执行结果是否正确的脚本。因此，该golden脚本的逻辑需要与算子的实现逻辑保持一

致，如果golden脚本实现错误，会导致算子计算结果的精度比对失败，这种情况是golden数据不可信。

所以，在算子精度定界定位的过程中，用户需要自行根据自定义算子的逻辑，检查golden脚本的正确性，尤其是对于复杂计算逻辑的算子，建议此排查优先进行。

----结束

