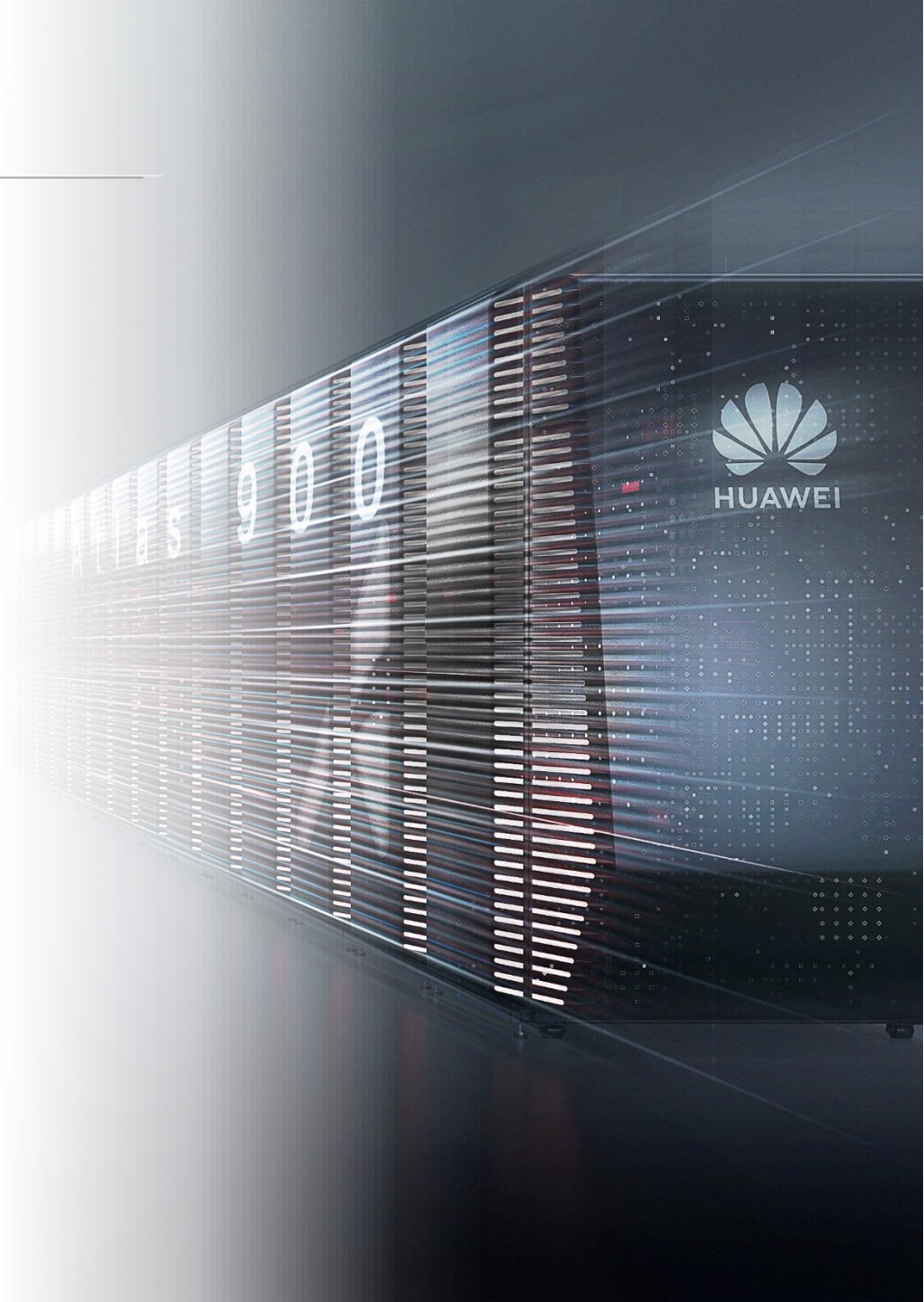
Ascend C编程快速入门

HUAWEI

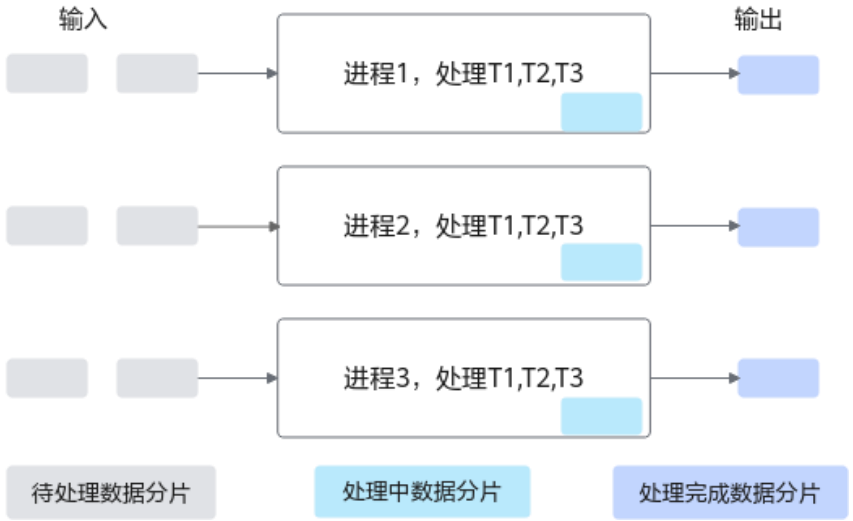
TABLE OF CONTENTS



Ascend C编程模型

Ascend C算子编程是SPMD（Single-Program Multiple-Data）编程，将需要处理的数据拆分并分布在多个计算核心上运行，多个AI Core共享相同的指令代码。

SPMD（Single-Program Multiple-Data）数据并行



输入数据切分后分发给不同进程处理，

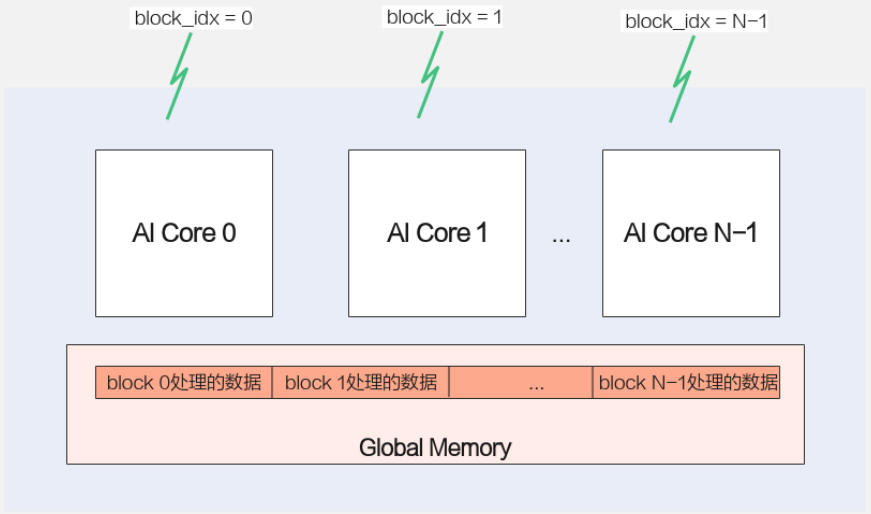
每个进程对自己的数据分片进行3个任务（ T1、T2、T3 ）的处理

开发者仅需关注单核算子实现

每个核上的运行实例唯一的区别是block_idx不同

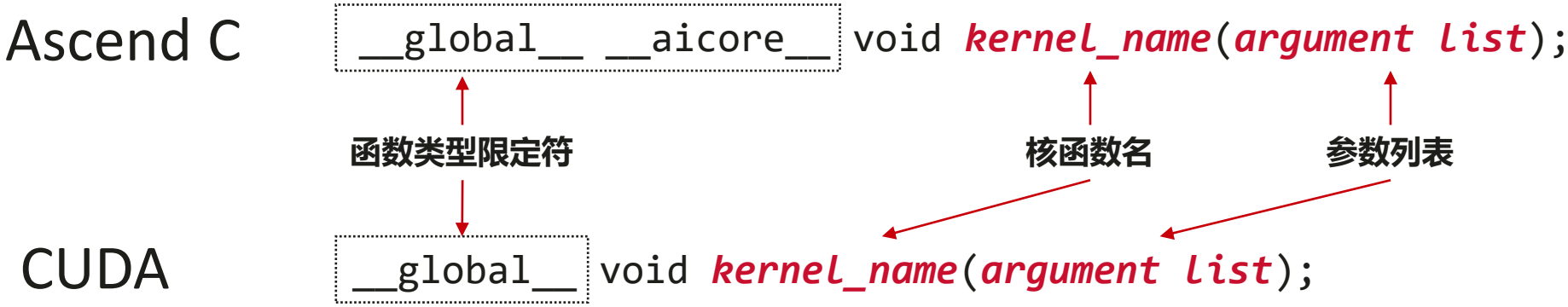
每个block用内嵌变量block_idx来识别自己的身份

```
18  class KernelAdd {
19  public:
20      __aicore__ inline KernelAdd() {}
21      __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z)
22      {
23          // 获得本block要计算的数据位置
24          xGm.SetGlobalBuffer((__gm__ half*)x + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
25          yGm.SetGlobalBuffer((__gm__ half*)y + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
26          zGm.SetGlobalBuffer((__gm__ half*)z + BLOCK_LENGTH * GetBlockIdx(), BLOCK_LENGTH);
27          // 初始化对列的内存资源
28          pipe.InitBuffer(inQueueX, BUFFER_NUM, TILE_LENGTH * sizeof(half));
29          pipe.InitBuffer(inQueueY, BUFFER_NUM, TILE_LENGTH * sizeof(half));
30          pipe.InitBuffer(outQueueZ, BUFFER_NUM, TILE_LENGTH * sizeof(half));
31      }
32      ...
33  }
```



什么是核函数

核函数（Kernel Function）是Ascend C算子设备侧的入口。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侧的运行代码，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。



核函数是直接在设备侧执行的代码。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**，SPMD编程模型允许核函数调用时，都执行相同的核函数代码，具有相同的参数，并行执行。

如何编写核函数

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侧AI Core上执行：

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

表1 函数类型限定符

函数类型限定符	执行	调用	备注
__global__	在设备侧执行	由<<<...>>>来调用	必须为void返回值类型
__aicore__	在设备侧执行	仅从设备端调用	-

如何编写核函数

使用变量类型限定符

为了方便：指针入参变量统一的类型定义为*__gm__ uint8_t**

用户可统一使用*uint8_t*类型的指针，并在使用时转化为实际的指针类型；亦可直接传入实际的指针类型

表2 变量类型限定符

变量类型限定符	内存空间	意义
__gm__	驻留在Global Memory上	表明该指针变量指向Global Memory上某处内存地址

规则或建议

1. 核函数必须具有void返回类型
2. 仅支持入参为指针类型或C/C++内置数据类型(Primitive Data Types)，如：half* s0、float* s1、int32_t c
3. 提供了一个封装的宏GM_ADDR来避免过长的函数入参列表

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

如何调用核函数

核函数的调用语句是C/C++函数调用语句的一种扩展

常见的C/C++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);
```

注：内核调用符仅可在NPU模式下编译时调用，CPU模式下编译无法识别该符号

- *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，表现为内置变量*block_idx*，编号从0开始，可为不同的逻辑核定义不同的行为，可以在算子实现中使用*GetBlockIdx()*函数来获得
- *l2ctrl*，保留参数，暂时设置为固定值*nullptr*
- *stream*，类型为*aclrtStream*，*stream*是一个任务队列，应用程序通过*stream*来管理任务的并行

HelloWorld——Ascend C核函数的基本写法

核函数定义

核函数实现

核函数调用

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

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

HelloWorld——调用

重要接口：

- aclInit
- aclrtCreateStream
- aclrtMallocHost
- aclrtMalloc
- aclrtMemcpy
- <<<…>>>
- aclrtSynchronizeStream
- aclrtFree
- aclrtFreeHost
- aclrtDestroyStream
- aclFinalize

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

int32_t main(int argc, char const *argv[]) {
    aclInit(nullptr);
    aclrtContext context;
    int32_t deviceId = 0;
    aclrtSetDevice(deviceId);
    aclrtCreateContext(&context,deviceId);
    aclrtStream stream = nullptr;
    aclrtCreateStream(&stream);

    constexpr uint32_t blockDim = 8;
    hello_world_do(blockDim, stream);
    aclrtSynchronizeStream(stream);

    aclrtDestroyStream(stream);
    aclrtDestroyContext(context);
    aclrtResetDevice(deviceId);
    aclFinalize();
    return 0;
}
```

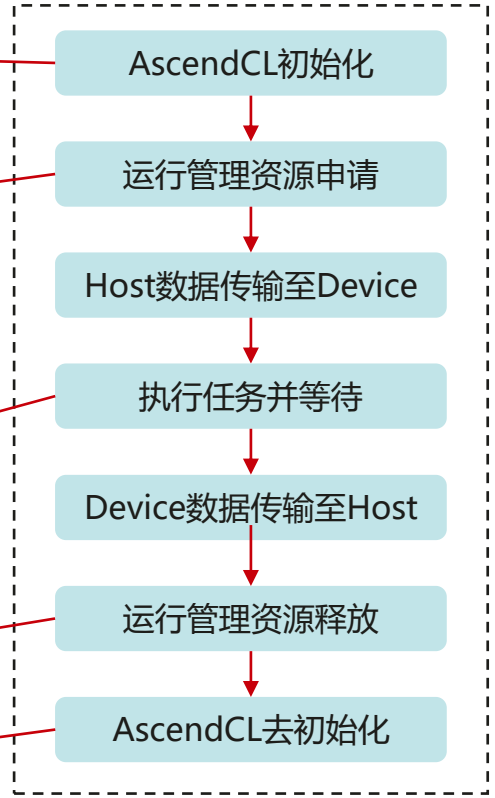
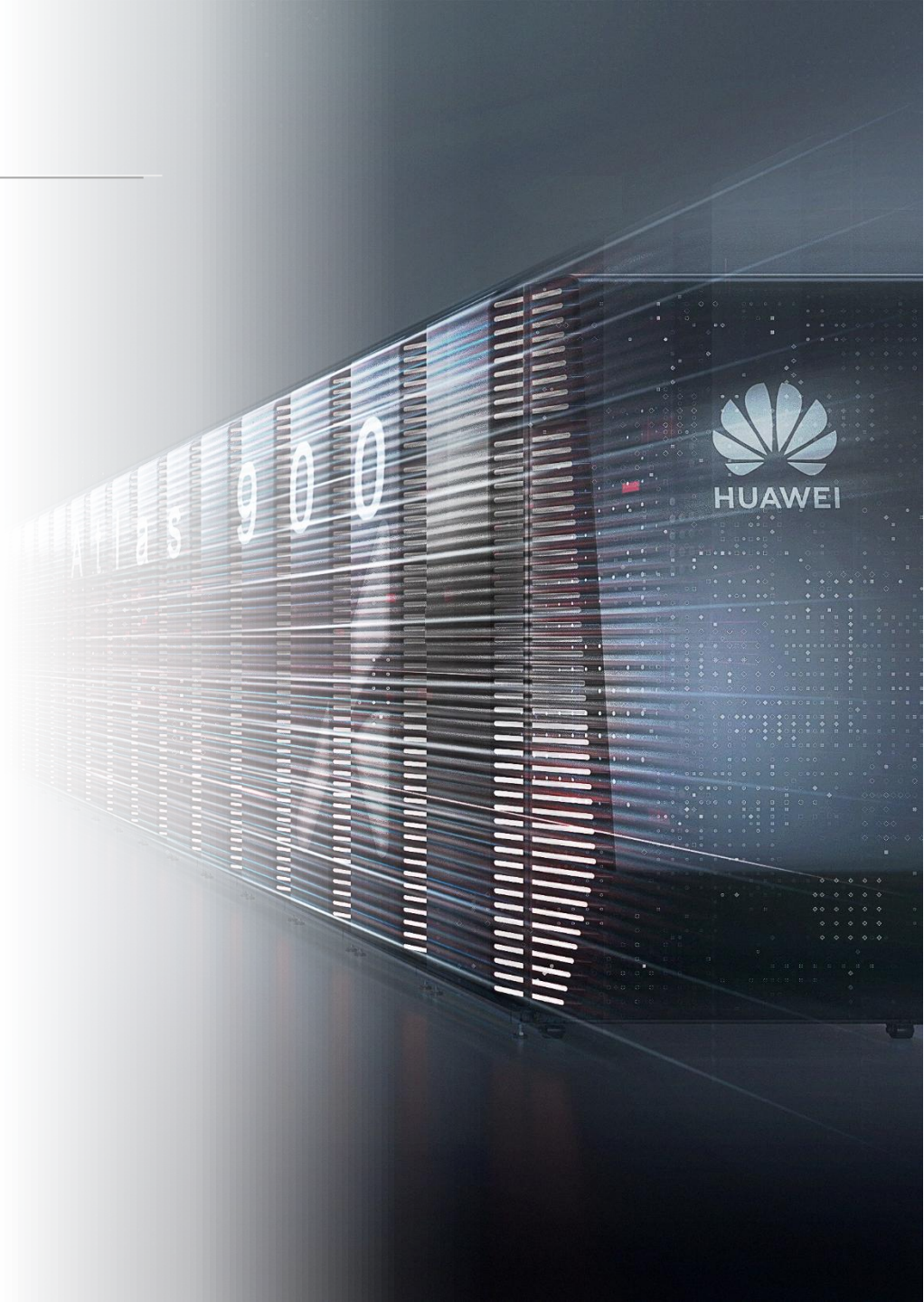
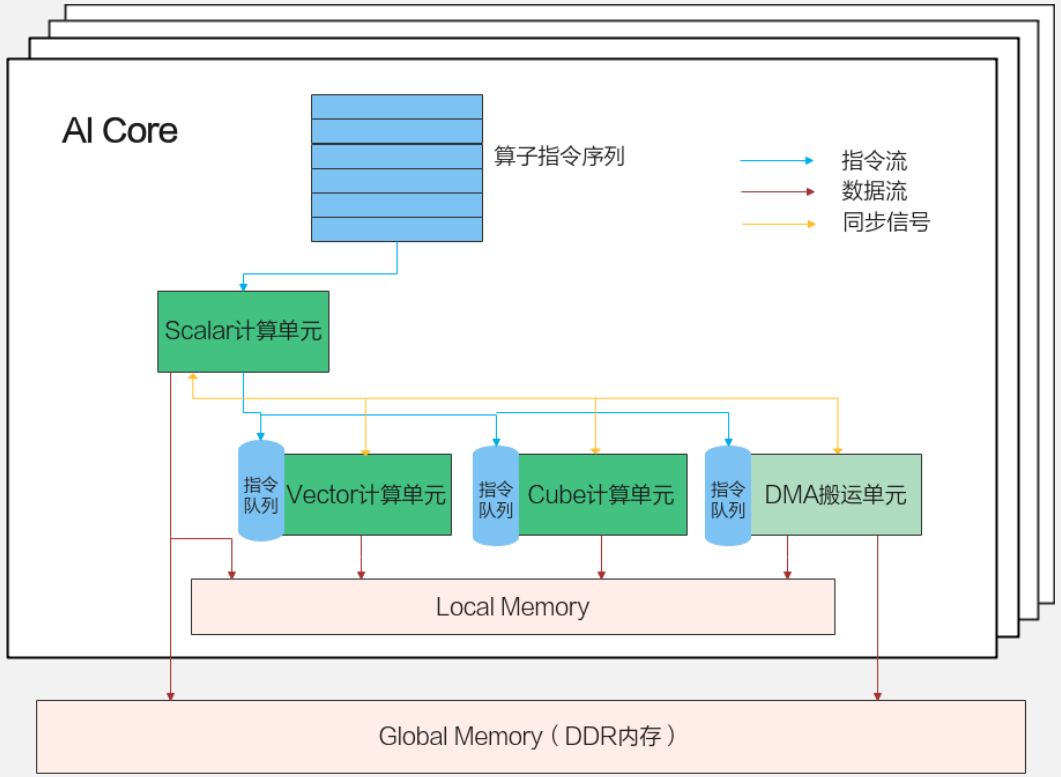


TABLE OF CONTENTS



硬件架构抽象

Ascend C基于硬件抽象架构进行编程，进而屏蔽不同硬件之间的差异。在理解硬件架构的抽象时，需要重点关注**异步指令流、同步信号流、计算数据流**三个过程：



AI Core内部计算架构示意图

核心组件　计算单元　存储单元　搬运单元

AI Core内异步计算过程
（指令流）

标量计算单元
读取指令序列
↓
标量计算单元
发射指令到对应单元
↓
各处理单元
并行执行指令

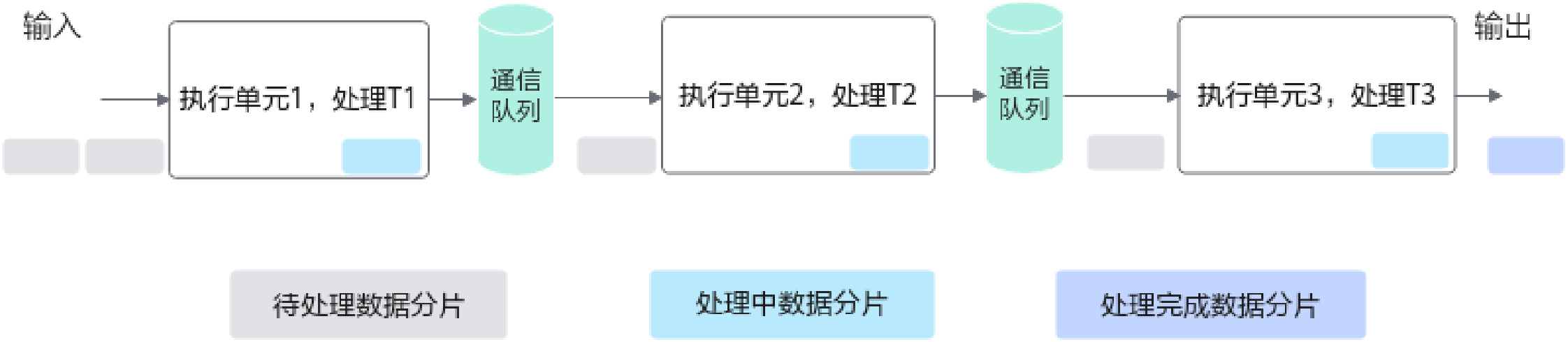
AI Core内存搬运过程
（数据流）

DMA
数据搬入LocalMem
↓
计算单元
数据完成计算，回写LocalMem
↓
DMA
数据搬出到GlobalMem

HUAWEI

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。

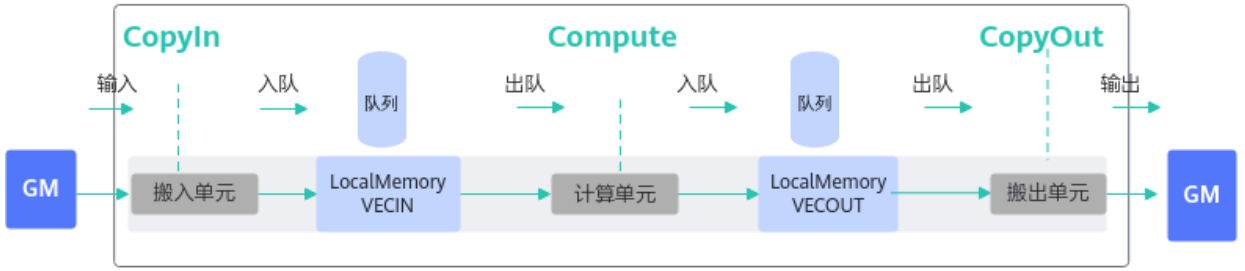


Ascend C编程范式就是这样一种流水线式的编程范式，把算子核内的处理程序，分成多个流水任务（Stage），以张量（Tensor）为数据载体，以队列（Queue）进行任务之间的通信与同步，以内存管理模块（Pipe）管理任务间的通信内存。

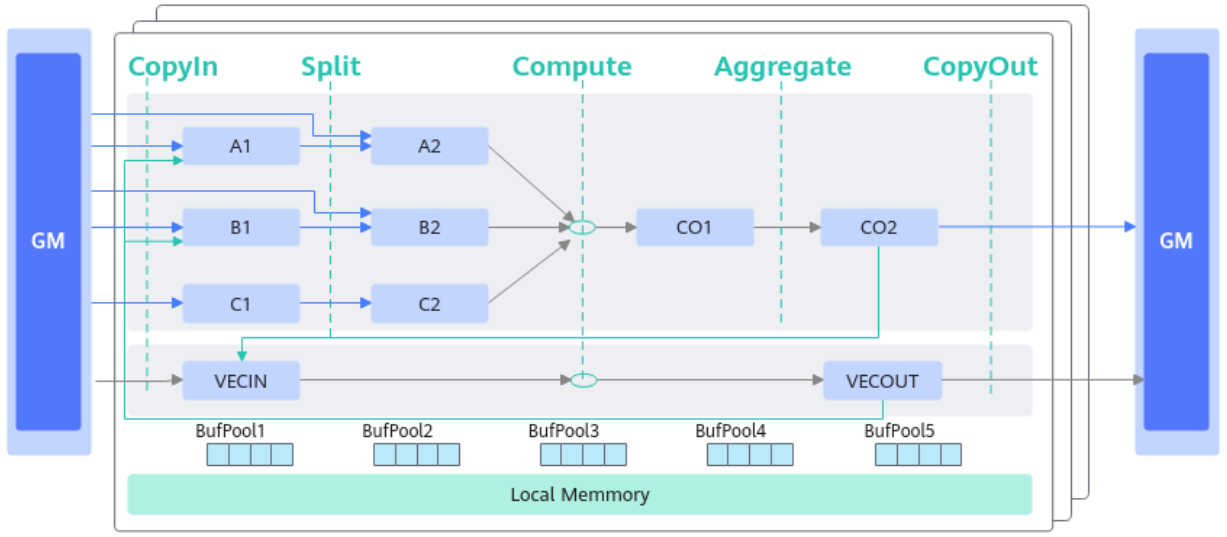
编程范式

1. 针对各代Davinci芯片的复杂数据流，根据实际计算需求，抽象出并行编程范式，简化流水并行
2. Ascend C的并行编程范式核心要素
    1. 一组并行计算任务
    2. 通过队列实现任务之间的通信和同步
    3. 程序员自主表达对并行计算任务和资源的调度
3. 典型的计算范式
    1. 基本的矢量编程范式：计算任务分为CopyIn, Compute, CopyOut
    2. 基本的矩阵编程范式：计算任务分为CopyIn, Split, Compute, Aggregate, CopyOut
    3. 复杂的矢量/矩阵编程范式，通过将矢量/矩阵的Out/In 组合在一起的方式来实现复杂计算数据流

Vector编程范式



Cube编程范式

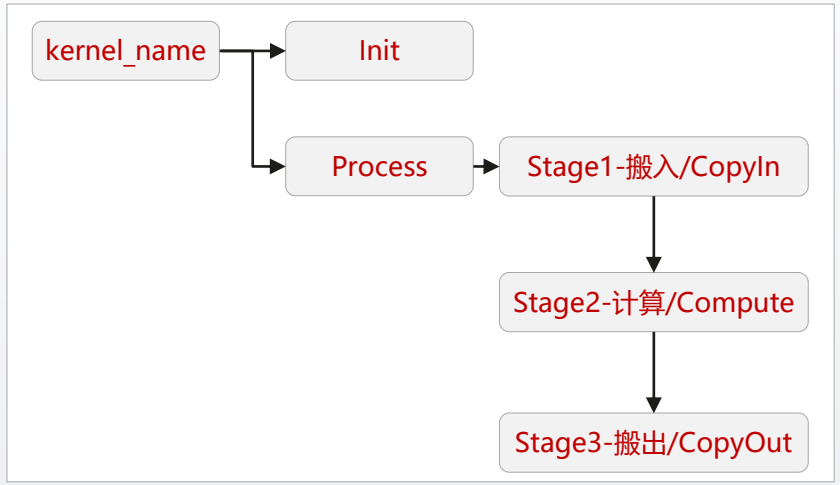


HUAWEI

结构化核函数编程，快速搭建算子实现代码框架

Ascend C流水编程范式将单核算子处理逻辑划分为多个流水任务“搬入、计算、搬出”，基于该编程范式，可**快速搭建算子实现的代码框架**

矢量算子编程基本任务设计



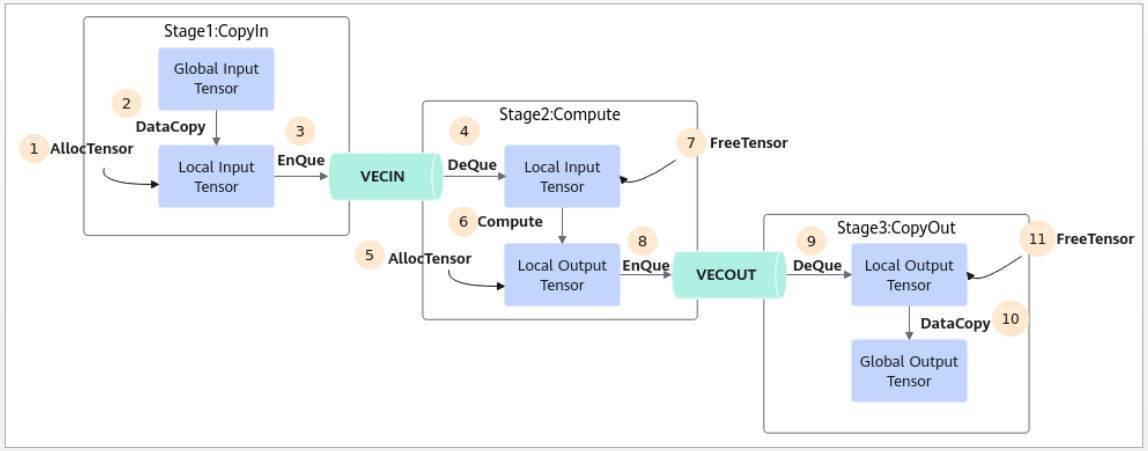
内存初始化、创建队列资源

输入数据从Global内存搬运到Local内存

使用Local内存数据进行计算

计算结果从Local内存搬运到Global内存

矢量算子编程任务间通信和同步



单核处理程序（核函数）定义

核函数（Kernel Function）是Ascend C算子Device侧实现的入口，
在核函数中，需要为在AI Core上执行的代码规定要进行的数据访问和计算操作

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

函数类型限定符
标识该核函数在AI Core上执行

核函数名

参数列表

核函数实现

```
class KernelAdd {
public:
    __aicore__ inline KernelAdd(){}
    __aicore__ inline void Init(GM_ADDR x, GM_ADDR y, GM_ADDR z)
    {
        ...
    }                                                                    Init
    __aicore__ inline void Process()
    {
        constexpr int32_t loopCount = TILE_NUM * BUFFER_NUM;
        for (int32_t i = 0; i < loopCount; i++) {
            CopyIn(i);
            Compute(i);
            CopyOut(i);
        }
    }
private:
    __aicore__ inline void CopyIn(int32_t progress)
    {
        LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
        LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
        DataCopy(xLocal, xGm[progress * TILE_LENGTH], TILE_LENGTH);
        DataCopy(yLocal, yGm[progress * TILE_LENGTH], TILE_LENGTH);
        inQueueX.EnQue(xLocal);
        inQueueY.EnQue(yLocal);
    }                                                                    CopyIn
    __aicore__ inline void Compute(int32_t progress)
    {
        LocalTensor<half> xLocal = inQueueX.DeQue<half>();
        LocalTensor<half> yLocal = inQueueY.DeQue<half>();
        LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
        Add(zLocal, xLocal, yLocal, TILE_LENGTH);
        outQueueZ.EnQue<half>(zLocal);
        inQueueX.FreeTensor(xLocal);
        inQueueY.FreeTensor(yLocal);
    }                                                                    Compute
    __aicore__ inline void CopyOut(int32_t progress)
    {
        LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
        DataCopy(zGm[progress * TILE_LENGTH], zLocal, TILE_LENGTH);
        outQueueZ.FreeTensor(zLocal);
    }                                                                    CopyOut
private:
    TPipe pipe;
    TQue<QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;
    TQue<QuePosition::VECOUT, BUFFER_NUM> outQueueZ;
    GlobalTensor<half> xGm, yGm, zGm;
};
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();
}                                                                        核函数定义
```

编程范式——内存管理

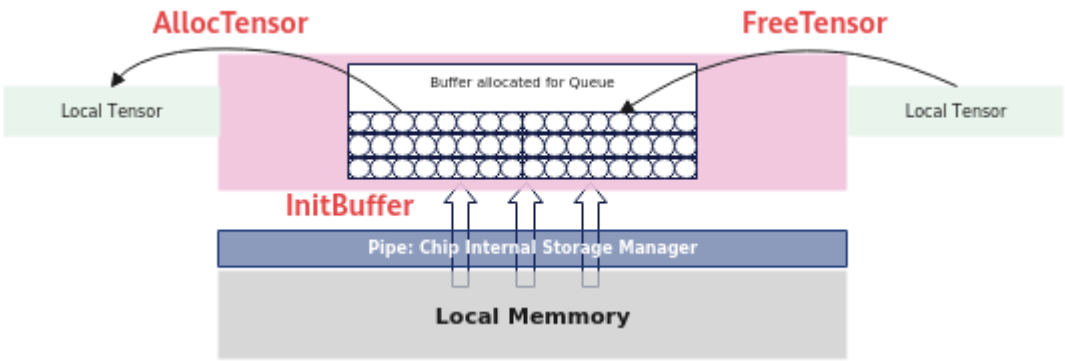
任务间数据传递使用到的内存统一由内存管理模块*Pipe*进行管理

*Pipe*作为片上内存管理者，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*分配内存给*Tensor*，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

InitBuffer、AllocTensor和FreeTensor的示例

```
// 使用AllocTensor分配Tensor
TPipe pipe;
TQue<TPosition::VECOUT, 2> que;
int num = 4;
int len = 1024;
// InitBuffer分配内存块数为4，每块大小为1024Bytes
pipe.InitBuffer(que, num, len);
// AllocTensor分配Tensor长度为1024Bytes
LocalTensor<half> tensor1 = que.AllocTensor();
// 使用FreeTensor释放通过AllocTensor分配的Tensor，注意配对使用
que.FreeTensor<half>(tensor1);
```



内存管理示意图

编程范式 —— 任务间通信和同步

数据通信与同步的管理者

不同的流水任务之间存在数据依赖，需要进行数据传递

Ascend C中使用*Queue*队列完成任务之间的数据通信和同步，

*Queue*提供了*EnQue*、*DeQue*等基础API

*Queue*队列管理NPU上不同层级的物理内存时，用一种抽象的**逻辑位置**（*QuePosition*）来表达各个级别的存储（Storage Scope），代替了片上物理存储的概念，开发者无需感知硬件架构

TPosition类型包括：*VECIN*、*VECCALC*、*VECOUT*、*A1*、*A2*、*B1*、*B2*、*CO1*、*CO2*。其中VECIN、VECCALC、VECOUT主要用于向量编程；A1、A2、B1、B2、CO1、CO2用于矩阵编程

数据的载体

Ascend C使用*GlobalTensor*和*LocalTensor*作为数据的基本操作单元，它是各种指令API直接调用的对象，也是数据的载体

TPosition	具体含义
GM	Global Memory，对应AI Core的外部存储。
VECIN	用于向量计算，搬入数据的存放位置，在数据搬入Vector计算单元时使用此位置
VECOUT	用于向量计算，搬出数据的存放位置，在将Vector计算单元结果搬出时使用此位置
VECCALC	用于向量计算/矩阵计算，在计算需要临时变量时使用此位置
A1	用于矩阵计算，存放整块A矩阵，可类比CPU多级缓存中的二级缓存
B1	用于矩阵计算，存放整块B矩阵，可类比CPU多级缓存中的二级缓存
A2	用于矩阵计算，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
B2	用于矩阵计算，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
CO1	用于矩阵计算，存放小块结果C矩阵，可理解为Cube Out
CO2	用于矩阵计算，存放整块结果C矩阵，可理解为Cube Out

编程范式 —— 矢量编程任务间通信和同步

矢量编程中的**逻辑位置（QuePosition）**：搬入数据的存放位置：*VECIN*、搬出数据的存放位置：*VECOUT*

矢量编程主要分为**CopyIn**、**Compute**、**CopyOut**三个任务：

- **CopyIn**任务中将输入数据从*GlobalTensor*搬运至*LocalTensor*后，需要使用*EnQue*将*LocalTensor*放入*VECIN*的*Queue*中
- **Compute**任务等待*VECIN*的*Queue*中*LocalTensor*出队之后才可以进行矢量计算，计算完成后使用*EnQue*将计算结果*LocalTensor*放入到*VECOUT*的*Queue*中
- **CopyOut**任务等待*VECOUT*的*Queue*中*LocalTensor*出队，再将其拷贝到*GlobalTensor*

Stage1：CopyIn任务

使用*DataCopy*接口将*GlobalTensor*拷贝到*LocalTensor*

使用*EnQue*将*LocalTensor*放入*VECIN*的*Queue*中

Stage2：Compute任务

使用*DeQue*从*VECIN*中取出*LocalTensor*

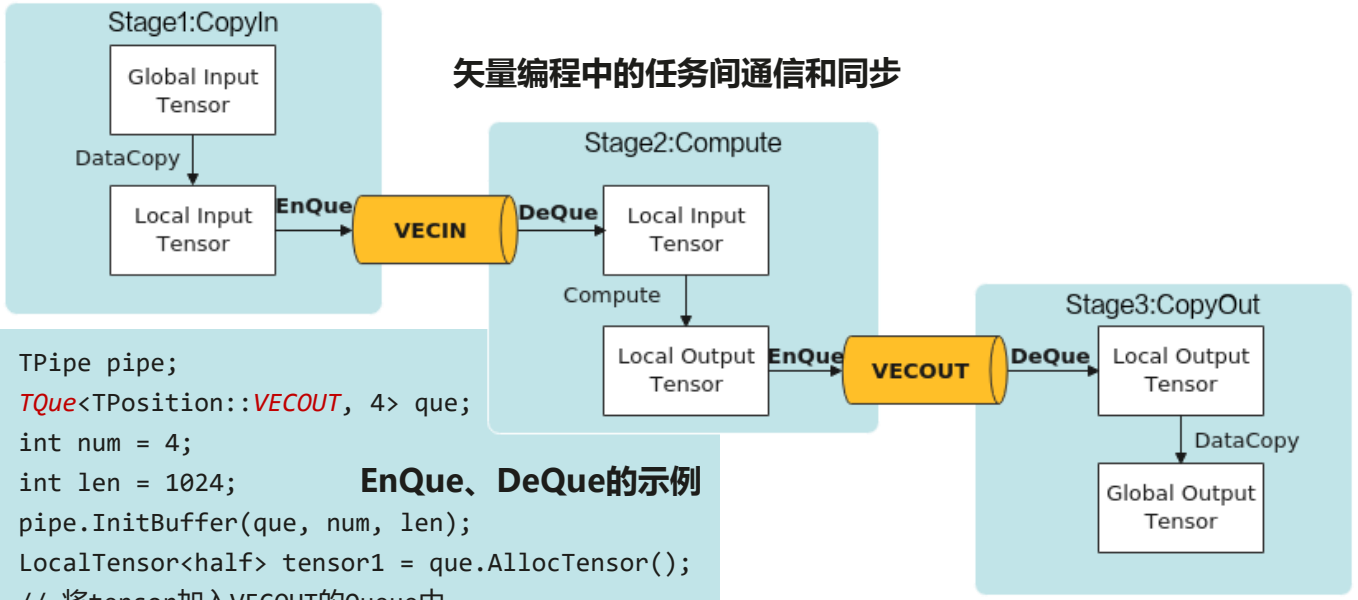
使用**Ascend C指令API**完成矢量计算：*Add*

使用*EnQue*将结果*LocalTensor*放入*VECOUT*的*Queue*中

Stage3：CopyOut任务

使用*DeQue*接口从*VECOUT*的*Queue*中取出*LocalTensor*

使用*DataCopy*接口将*LocalTensor*拷贝到*GlobalTensor*



EnQue、DeQue的示例

```
TPipe pipe;
TQue<TPosition::VECOUT, 4> que;
int num = 4;
int len = 1024;
pipe.InitBuffer(que, num, len);
LocalTensor<half> tensor1 = que.AllocTensor();
// 将tensor加入VECOUT的Queue中
que.EnQue(tensor1);
// 将tensor从VECOUT的Queue中搬出
LocalTensor<half> tensor2 = que.DeQue<half>();
```

编程范式——临时变量内存管理

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*QuePosition*上的存储空间，并使用*Get()*来将分配到的存储空间分配给新的*LocalTensor*

从*TBuf*上获取全部长度，或者获取指定长度的*LocalTensor*

```
LocalTensor<T> Get<T>();
LocalTensor<T> Get<T>(uint32_t len);
```

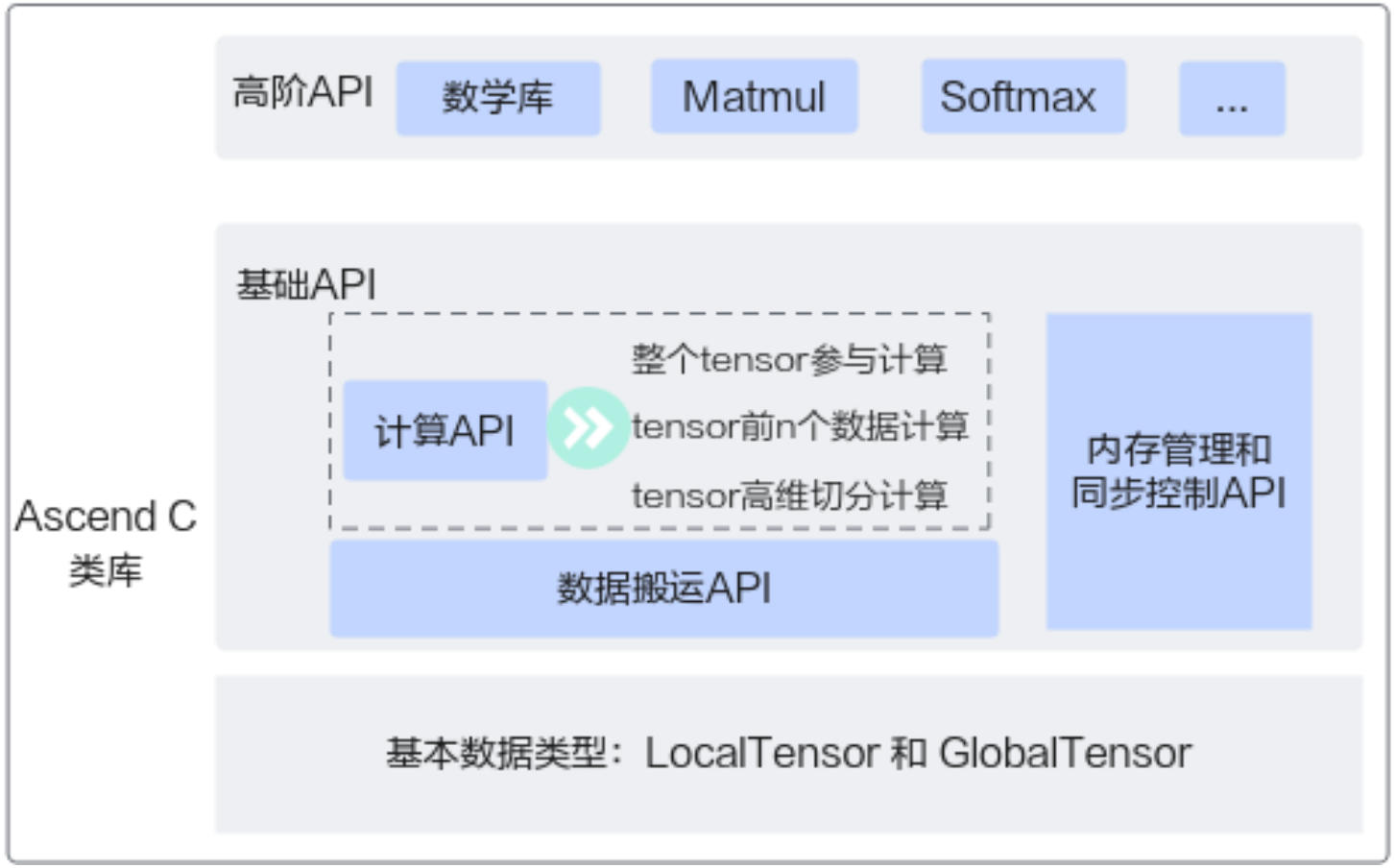
Tbuf及其Get接口的示例

```
// 为TBuf初始化分配内存，分配内存长度为1024字节
TPipe pipe;
TBuf<TPosition::VECALC> calcBuf; // 模板参数为QuePosition中的VECOUT类型
uint32_t byteLen = 1024;
pipe.InitBuffer(calcBuf, byteLen);
// 从calcBuf获取Tensor,Tensor为pipe分配的所有内存大小，为1024字节
LocalTensor<int32_t> tempTensor1 = calcBuf.Get<int32_t>();
// 从calcBuf获取Tensor,Tensor为128个int32_t类型元素的内存大小，为512字节
LocalTensor<int32_t> tempTensor1 = calcBuf.Get<int32_t>(128);
```

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

Ascend C API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，开发者根据自己的需求选择合适的API。Ascend C API的操作数都是Tensor类型：GlobalTensor和LocalTensor；类库API分为高阶API和基础API。



Ascend C编程类库API示意图

基础API

实现基础功能的API，包括计算类、数据搬运、内存管理和任务同步等。使用基础API自由度更高，可以通过API组合实现自己的算子逻辑。基础API是对计算能力的表达。

计算类API：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

数据搬运API：基于Local Memory数据进行计算，数据需要先从Global Memory搬运至Local Memory，再使用计算接口完成计算，最后从Local Memory搬出至Global Memory。比如DataCopy接口

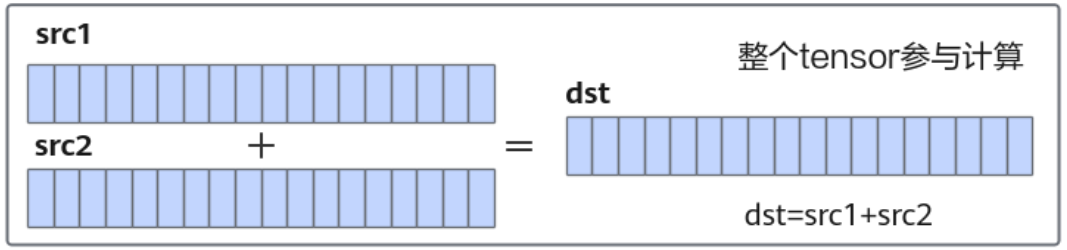
内存管理API：用于分配管理内存，比如AllocTensor、FreeTensor接口

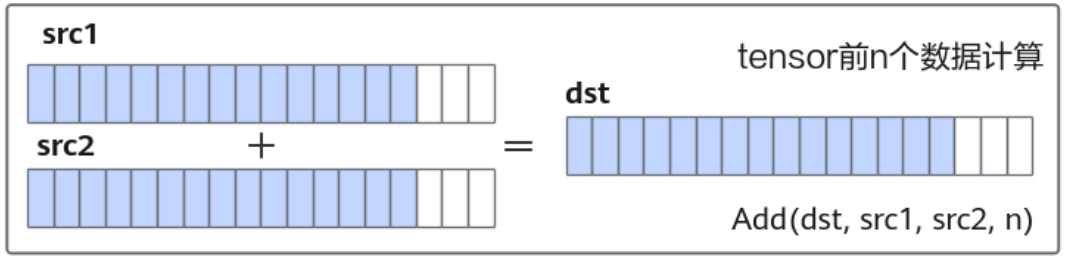
任务同步API：完成任务间的通信和同步，比如EnQue、DeQue接口。不同的指令异步并行执行，为了保证不同指令队列间的指令按照正确的逻辑关系执行，需要向不同的组件发送同步指令

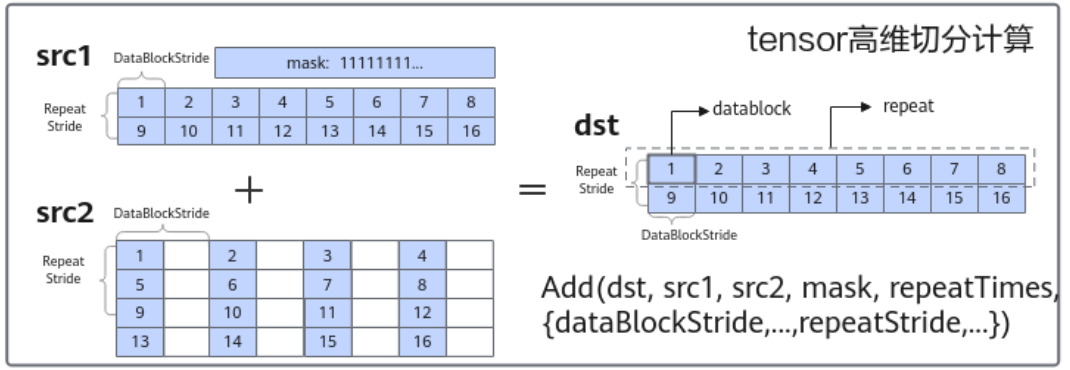
Ascend C API用于计算的基本数据类型都是Tensor：GlobalTensor和LocalTensor

计算API几种计算方式的特点

命名	说明
整个Tensor参与计算	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	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Repeat times（迭代的次数）、 Block stride（单次迭代内不同block间地址步长）、Repeat stride（相邻迭代间相同block的地址步长）、Mask（用于控制参与运算的计算单元）的操作。







HUAWEI

Tensor高维切分计算通用参数

- Repeat times（迭代的次数）
- Repeat stride（相邻迭代间相同block的地址步长）
- Block stride（单次迭代内不同block间地址步长）
- Mask（用于控制参与运算的计算单元）

以上Repeat times、Repeat stride、Block stride、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block间地址步长Block stride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。

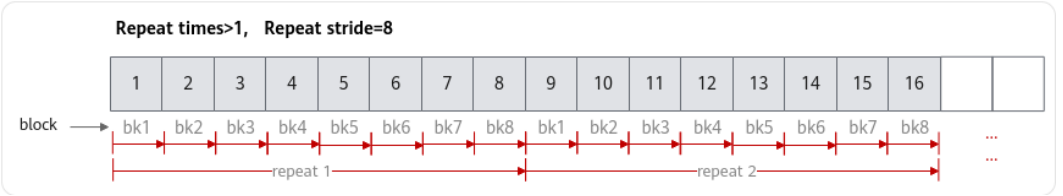
基础API通用说明--重复迭代次数-Repeat times

矢量计算单元，每次读取连续的8个block（每个block 32 Bytes，共256 Bytes）数据进行计算，为完成对输入数据的处理，必须通过多次迭代（repeat）才能完成所有数据的读取与计算。Repeat times表示迭代的次数

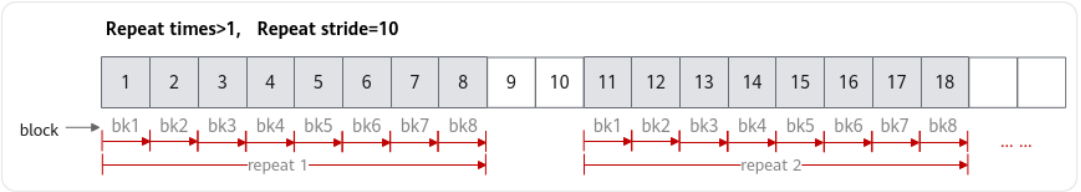
基础API通用说明--相邻迭代间相同block的地址步长-Repeat stride

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

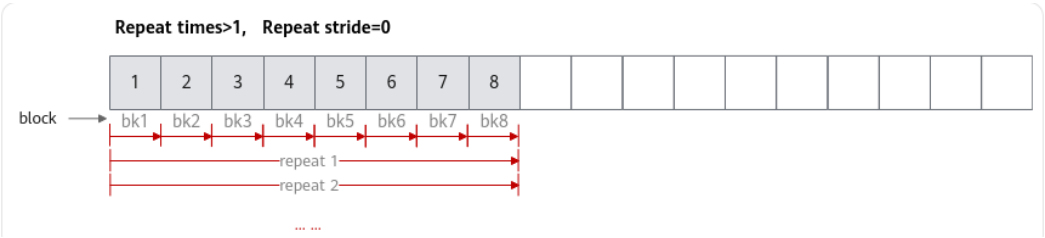
连续计算场景：假设定义一个Tensor供目的操作数和源操作数同时使用（即地址重叠），Repeat stride取值为8。此时，矢量计算单元第一次迭代读取连续8个block，第二轮迭代读取下一个连续的8个block，通过多次迭代即可完成所有输入数据的计算。



非连续计算场景：Repeat stride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block的间隔。



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

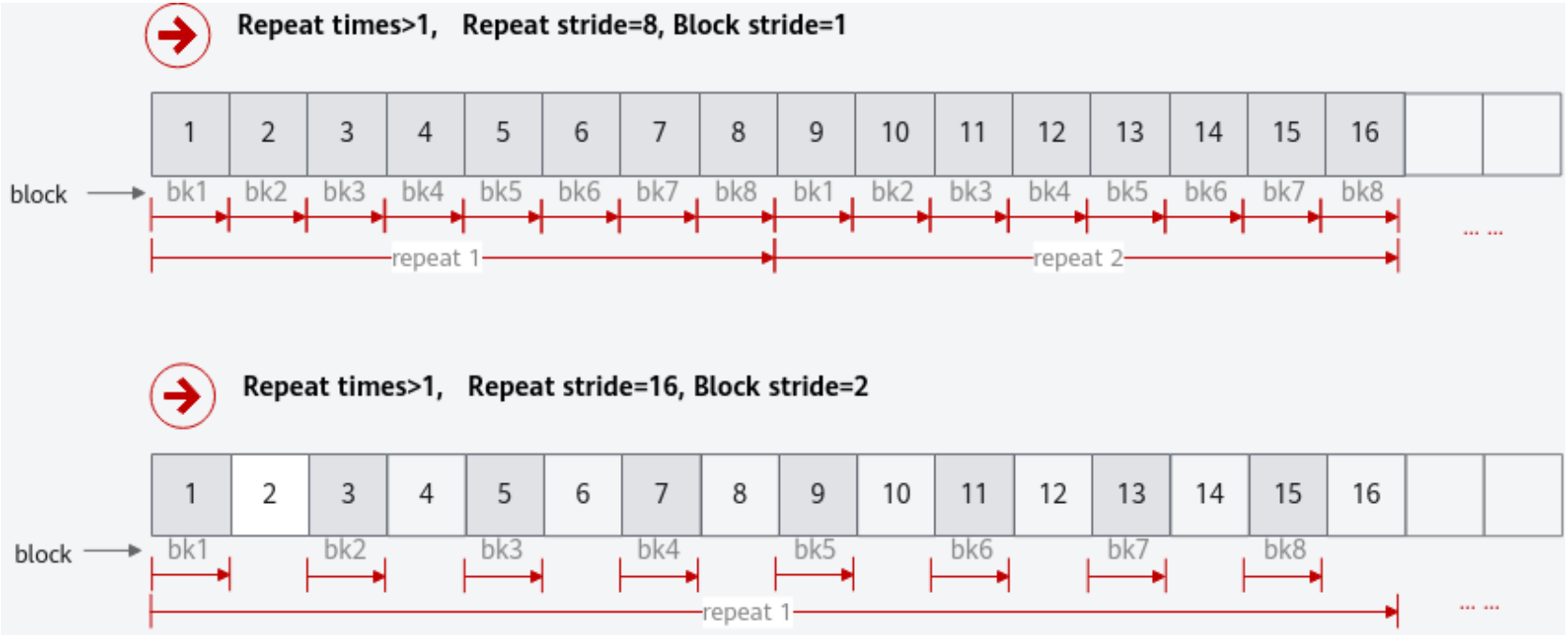


HUAWEI

基础API通用说明--同一迭代内不同block的地址步长-Block stride

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block的地址步长Block stride来实现。

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



基础API通用说明--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比特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比特位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]。

- 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dst/src为32位时，mask[1]为0，mask[0]∈（0, 2^{64}-1]。

思考?

- Repeat times（迭代的次数） 2+1 (260//128+1)
- Repeat stride（相邻迭代间相同block的地址步长） 8
- Block stride（单次迭代内不同block间地址步长） 1
- Mask（用于控制参与运算的计算单元） 128 4

一个tensor中有260个half数据类型的数，做取绝对值操作，上面四个值怎么设置?

数据搬运API

原型定义

普通数据搬运2级接口，适用于连续数据搬运：

template <typename T>

void DataCopy(const LocalTensor<T>& dstLocal, const GlobalTensor<T>& srcGlobal, const uint32_t **calCount**);

普通数据搬运0级接口，适用于连续和不连续数据搬运：

template <typename T>

void DataCopy(const LocalTensor<T>& dstLocal, const GlobalTensor<T>& srcGlobal, const DataCopyParams& **intriParams**);

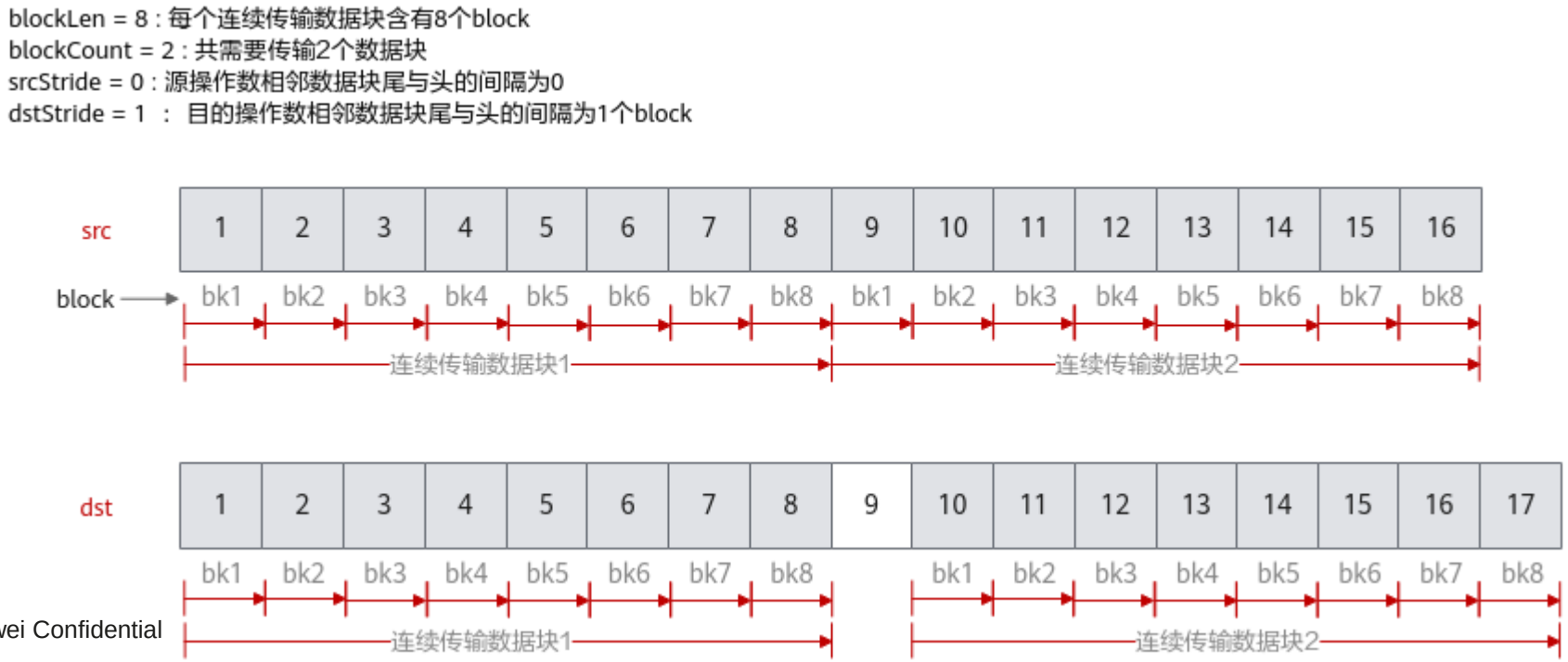
数据搬运API参数

blockCount：指定该指令包含的连续传输数据块个数。

blockLen：指定该指令每个连续传输数据块长度，单位为datablock(32Bytes)。

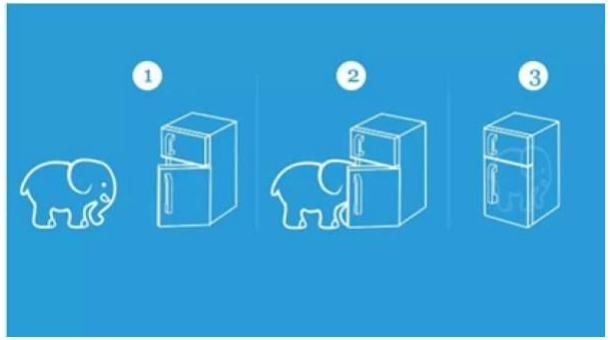
srcStride：源操作数，相邻连续数据块的间隔（前面一个数据块的尾与后面数据块的头的间隔），单位为datablock(32Bytes)。

dstStride：目的操作数，相邻连续数据块间的间隔（前面一个数据块的尾与后面数据块的头的间隔），单位为datablock(32Bytes)。



高阶API

封装常用算法逻辑的API，比如Matmul、Softmax等，可减少重复开发，提高开发者开发效率。使用高阶API可以快速的实现相对复杂的算法逻辑，高阶API是对于某种特定算法的表达。



So easy!

实现Matmul矩阵乘运算的具体步骤如下：

1.创建Matmul对象。

2.初始化

3.设置左矩阵A、右矩阵B、 Bias。

4.完成矩阵乘操作。

5.结束矩阵乘操作。

```
// 1、创建Matmul对象
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;
mm.Init(&tiling, &tpipe); // 初始化

// 2、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

// 3、完成矩阵乘操作。
while (mm.Iterate()) {
    mm.GetTensorC(gm_c);
}
//mm.IterateAll(gm_c);

// 4、结束矩阵乘操作
mm.End();
```

两种算子开发流程

Ascend C算子**快速开发**调试方式：

- 完成算子核函数的开发
- 基于Kernel直调的方式进行算子调用运行

Ascend C算子**标准开发**调试方式：

- 需完成Host侧和Device侧的开发
- 需完成应用程序的开发
- 基于单算子API（ACLNN）/ 单算子模型（ACLOP）/ PyTorch Adapter等方式进行算子调用运行

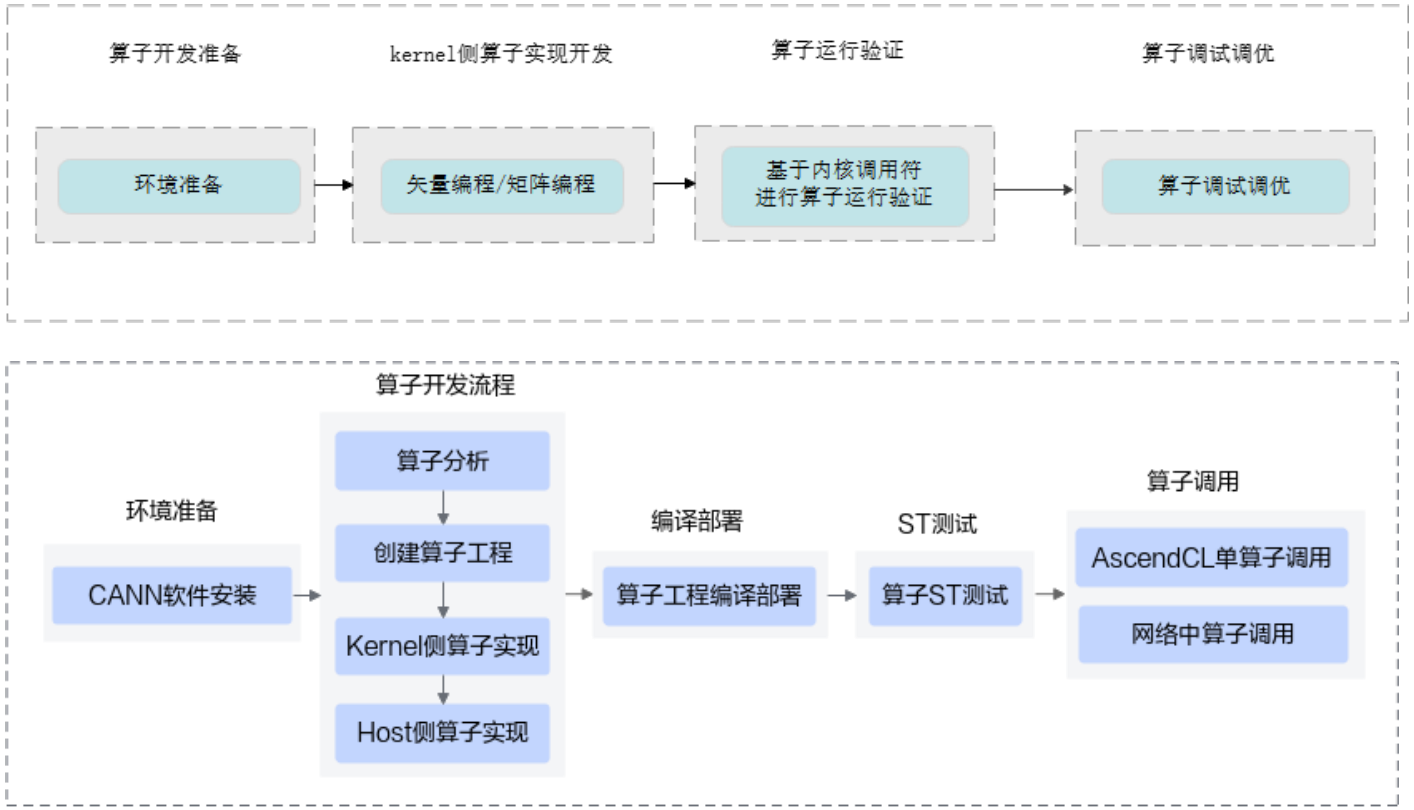


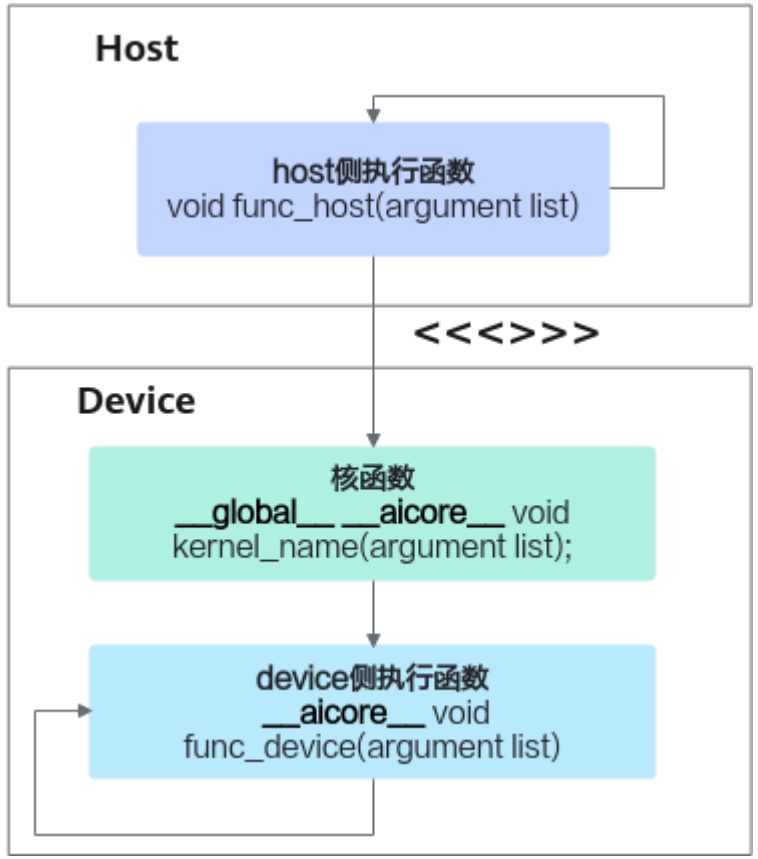
图6 自定义算子开发流程

	快速开发模式	标准开发模式
代码文件	少	多
开发时间	短	长
使用场景	单算子调用，快速验证算法逻辑	单算子网络/整网部署使用
推荐开发顺序	先	后

Ascend C算子快速开发

编程中使用到的函数可以分为三类：

- 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）。



Ascend C算子快速开发——开发流程

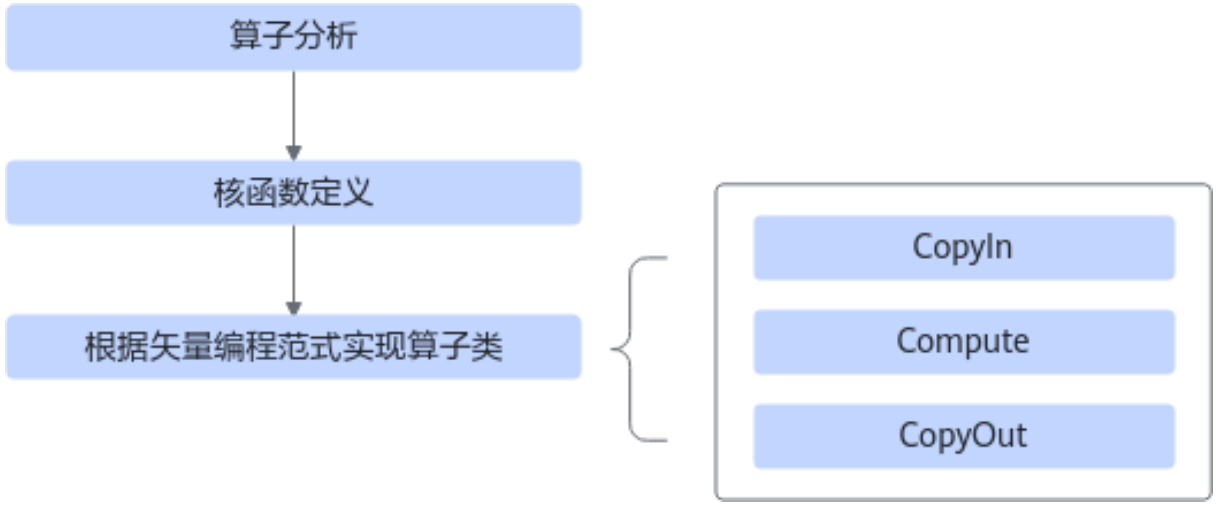
算子分析：分析算子的数学表达式、输入、输出以及计算逻辑的实现，明确需要调用的Ascend C接口。

核函数定义：定义Ascend C算子入口函数。

根据编程范式实现算子类：完成核函数的内部实现。

编写算子的应用程序：完成调用核函数main.cpp的代码

其他脚本：数据生成脚本，数据比对脚本



矢量算子开发流程

以**ElemWise**(*Add*)算子为例，数学公式：$\vec{z} = \vec{x} + \vec{y}$，为简单起见，设定输入张量*x*, *y*和输出张量*z*为固定shape(8, 2048)，数据类型dtype为*half*类型，数据排布类型format为*ND*，核函数名称为*add_custom*

Ascend C算子快速开发——算子分析

算子类型(OpType)	Add			
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	add_custom			

- **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：输入数据需要先搬入到片上存储，然后使用计算接口完成两个加法运算，得到最终结果，再搬出到外部存储

- **明确输入和输出**

Add算子有两个输入：$\vec{x}$ 与 $\vec{y}$，输出为$\vec{z}$。输入数据类型为*half*，输出数据类型与输入数据类型相同。输入支持固定shape(8，2048)，输出shape与输入shape相同。输入数据排布类型为*ND*

- **确定核函数名称和参数**

自定义核函数名，如*add_custom*。根据输入输出，确定核函数有3个入参*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

- **确定算子实现所需接口**

涉及内外部存储间的数据搬运，使用数据搬移接口：*DataCopy*实现

涉及矢量计算的加法操作，使用矢量双目指令：*Add*实现

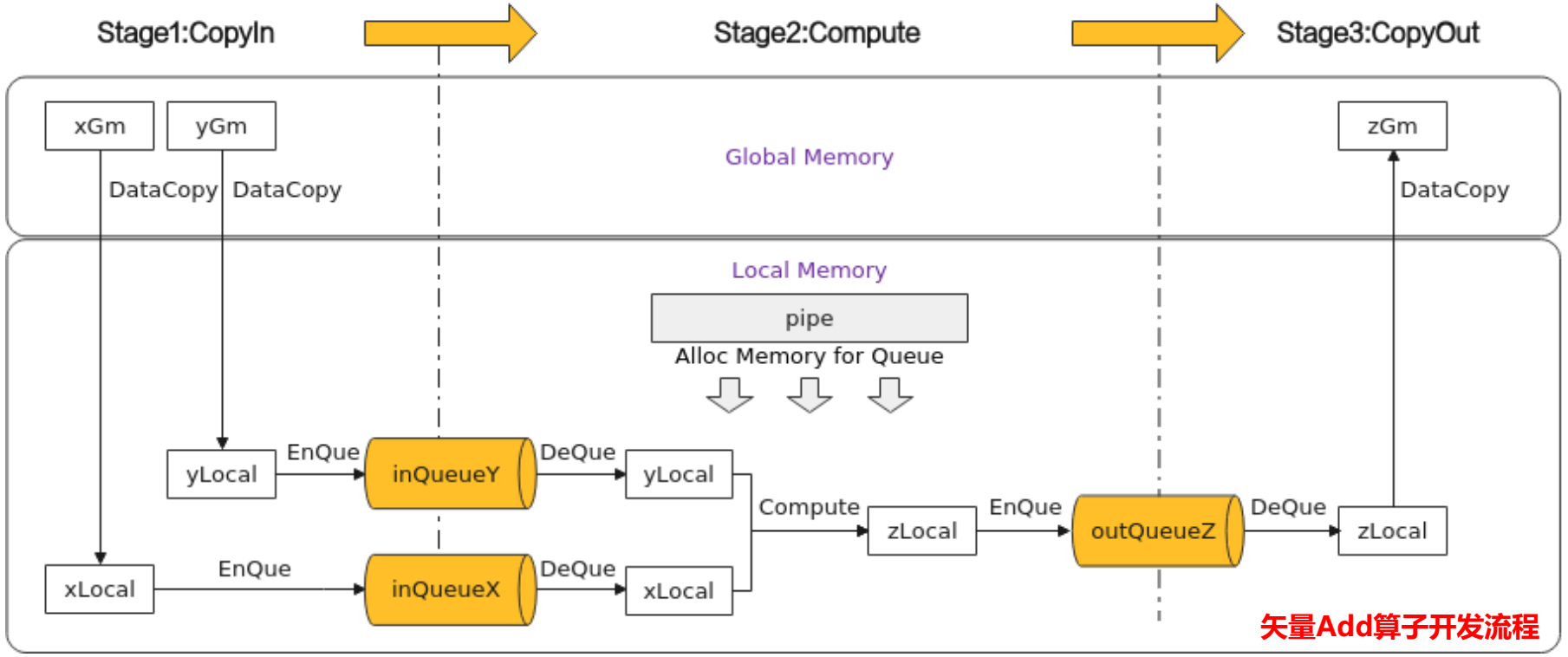
使用到*LocalTensor*，使用*Queue*队列管理，会使用到*EnQue*、*DeQue*等接口。

Ascend C算子快速开发——算子类实现

*CopyIn*任务：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：对*xLocal*, *yLocal*执行加法操作，计算结果存储在*zLocal*中
*CopyOut*任务：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

*pipe*内存管理对象对任务间交互使用到的内存、临时变量使用到的内存统一进行管理



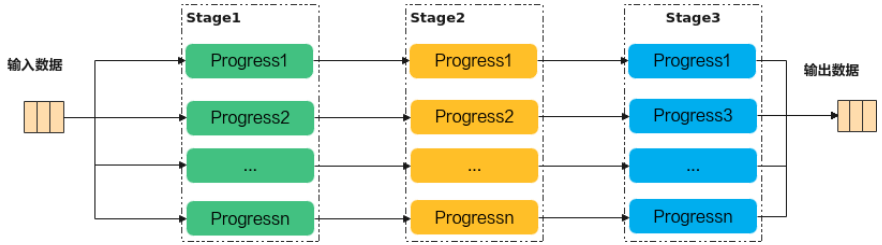
Ascend C算子快速开发——算子类实现

算子类类名：*KernelAdd*

初始化函数*Init()*和核心处理函数*Process()*

三个流水任务：*CopyIn()*，*Compute()*，*CopyOut()*

*Progress*的含义：



*TQue*模板的*BUFFER_NUM*的含义：

该Queue的深度，double buffer优化技巧

```
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__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t* z) {}
    // 核心处理函数，实现算子逻辑，调用私有成员函数CopyIn、Compute、CopyOut完成算子逻辑
    __aicore__ inline void Process() {}

private:
    // 搬入函数，完成CopyIn阶段的处理，被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Process函数调用
    __aicore__ inline void CopyOut(int32_t progress) {}

private:
    // Pipe内存管理对象
    TPipe pipe;
    // 输入数据Queue队列管理对象，QuePosition为VECIN
    TQue<QuePosition::VECIN, BUFFER_NUM> inQueueX, inQueueY;
    // 输出数据Queue队列管理对象，QuePosition为VECOUT
    TQue<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
    GlobalTensor<half> xGm, yGm, zGm;
};
```

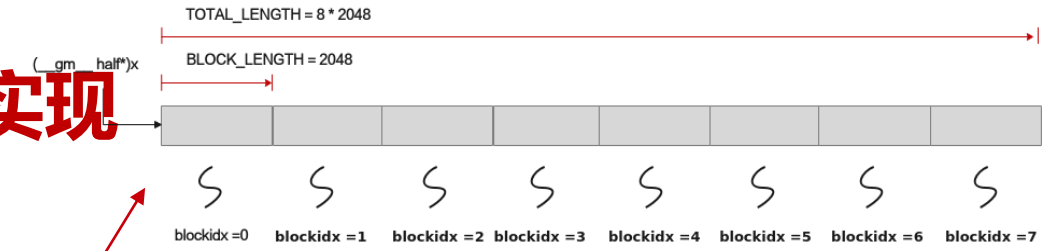
Ascend C算子快速开发——Init()实现

使用**多核并行计算**，需要将数据切片，获取到每个核实际需要处理的在Global Memory上的内存偏移地址

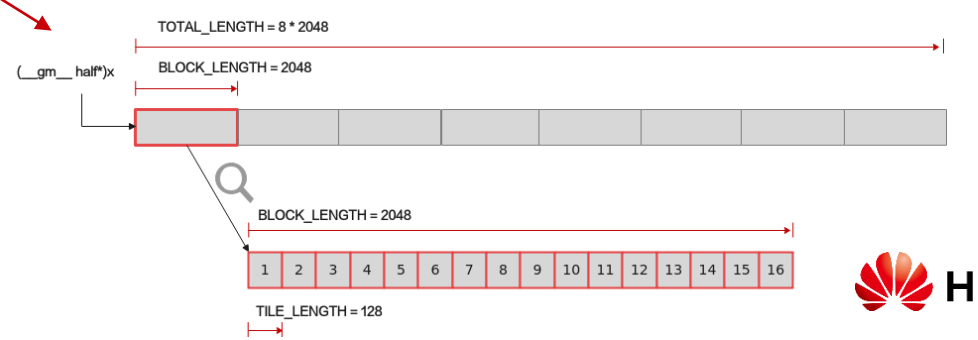
数据整体长度*TOTAL_LENGTH*为8* 2048，平均分配到8个核上运行，每个核上处理的数据大小*BLOCK_LENGTH*为2048。*block_idx*为核的逻辑ID，*(__gm__ half*)x + block_idx * BLOCK_LENGTH*即索引为*block_idx*的核的输入数据在Global Memory上的内存偏移地址

对于**单核处理数据**，可以进行数据切块（Tiling），将数据切分成8块。切分后的每个数据块再次切分成*BUFFER_NUM=2*块，可开启**double buffer**，实现流水线之间的并行

单核需要处理的2048个数被切分成16块，每块*TILE_LENGTH=128*个数据。*Pipe*为*inQueueX*分配了*BUFFER_NUM*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




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


__aicore__ inline void Init(__gm__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t*
z)
{
    // get start index for current core, core parallel
    xGm.SetGlobalBuffer((__gm__ half*)x + block_idx * BLOCK_LENGTH, BLOCK_LENGTH);
    yGm.SetGlobalBuffer((__gm__ half*)y + block_idx * BLOCK_LENGTH, BLOCK_LENGTH);
    zGm.SetGlobalBuffer((__gm__ half*)z + block_idx * BLOCK_LENGTH, 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));
}
```



HUAWEI

Ascend C算子快速开发——Process()实现

```
__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

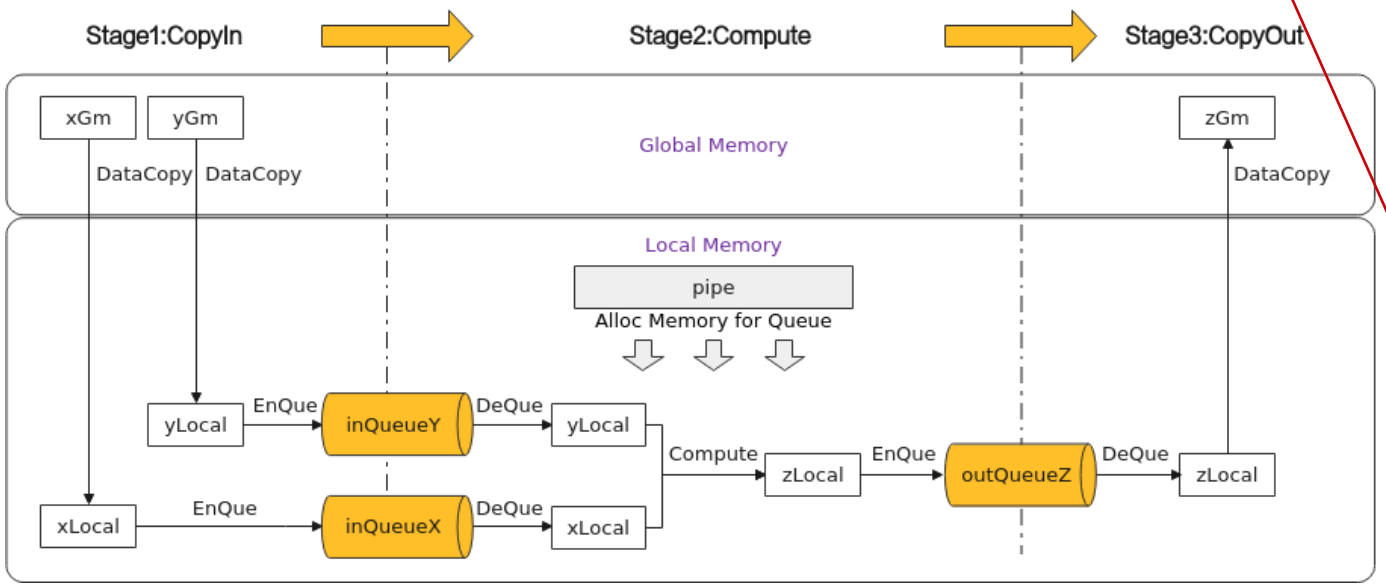
```
__aicore__ inline void CopyIn(int32_t progress)
{
    // alloc tensor from queue memory
    LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
    LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
    // copy progress_th tile from global tensor to local tensor
    DataCopy(xLocal, xGm[progress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

②

```
__aicore__ inline void Compute(int32_t progress)
{
    // deque input tensors from VECIN queue
    LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    // call Add instr for computation
    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

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



HUAWEI

矢量编程 —— double buffer机制

double buffer通过将**数据搬运**与**矢量计算**并行执行以隐藏数据搬运时间并降低矢量指令的等待时间，最终提高矢量计算单元的利用效率

1个Tensor同一时间只能进行搬入、计算和搬出三个流水任务中的一个，其他两个流水任务涉及的硬件单元则处于Idle状态

如果将待处理的数据一分为二，比如Tensor1、Tensor2

- 当矢量计算单元对Tensor1进行Compute时，Tensor2可以执行CopyIn的任务
- 当矢量计算单元对Tensor2进行Compute时，Tensor1可以执行CopyOut的任务
- 当矢量计算单元对Tensor2进行CopyOut时，Tensor1可以执行CopyIn的任务

由此，数据的进出搬运和矢量计算之间实现并行，硬件单元闲置问题得以有效缓解

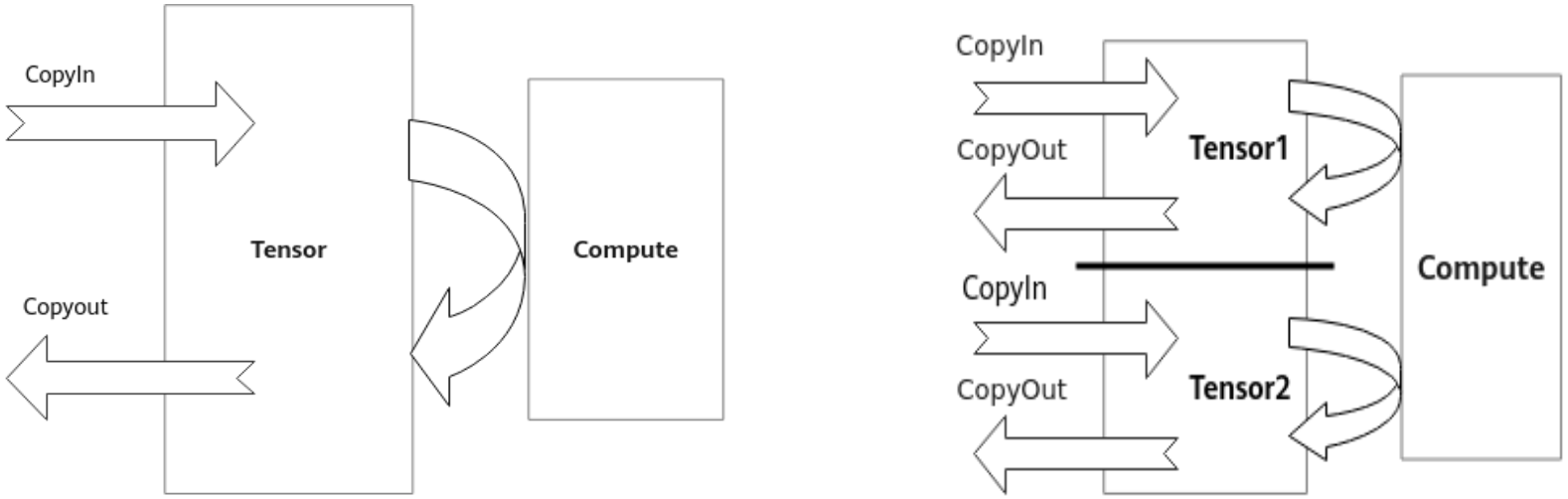


图9 Double Buffer下的数据搬运与Vector计算过程

HUAWEI

向量加法z=x+y代码样例：数据通路

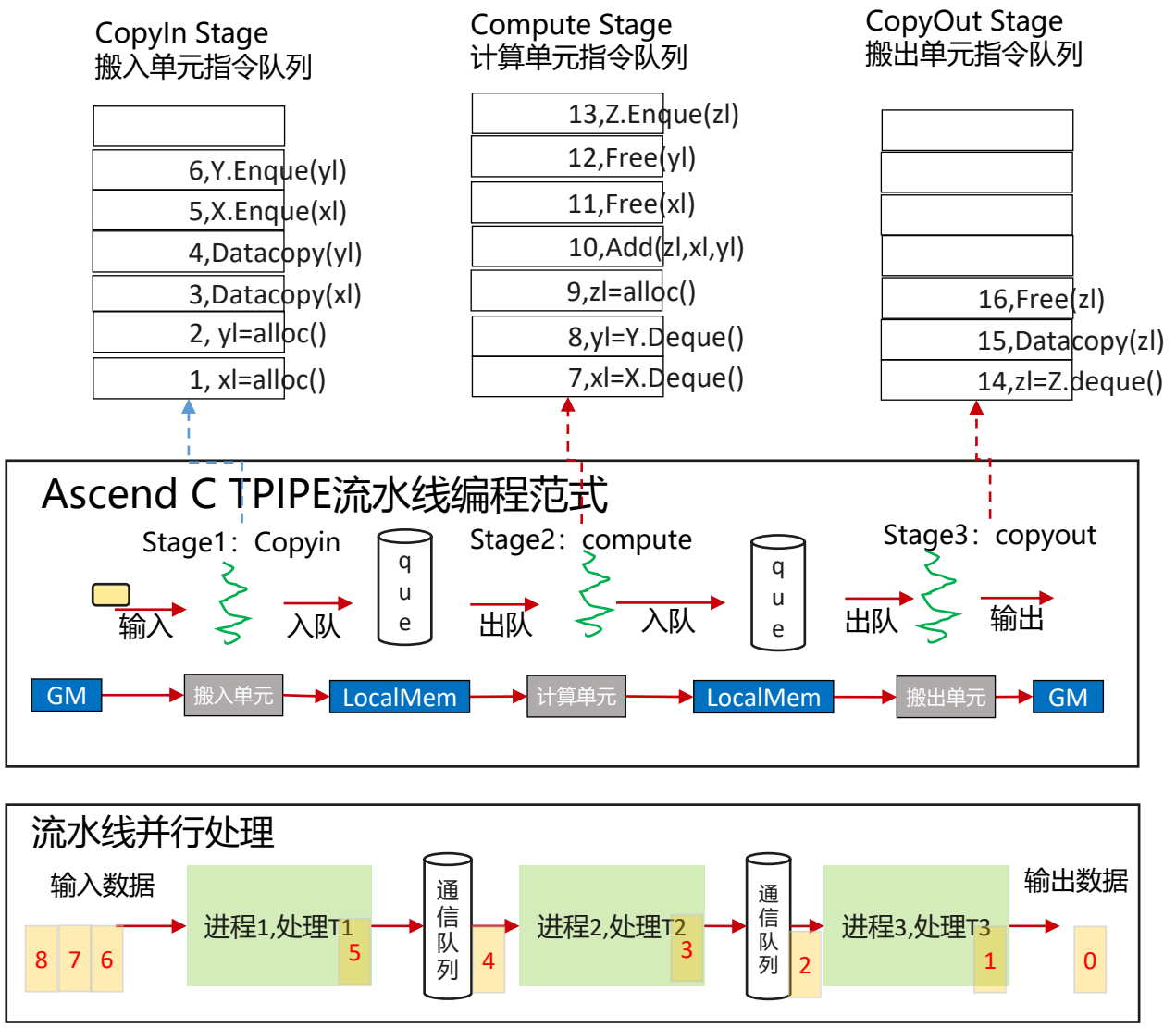
```
class KernelAdd {
public:
    __aicore__ void Init(__gm__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t* z)
    {
        ...
    }
    __aicore__ void Process()
    {
        constexpr int32_t loopCount = tileNum;
        for (int32_t i = 0; i < loopCount; i++) {
            CopyIn(i);
            Compute(i);
            CopyOut(i);
        }
    }

private:
    __aicore__ void CopyIn(int32_t progress)
    {
        LocalTensor<half> xLocal = inQueueX.AllocTensor<half>(); // 获得tensor所需内存资源
        LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
        DataCopy(xLocal, xGm[progress * tileLength], tileLength); // 复制当前第progress分片到local
        DataCopy(yLocal, yGm[progress * tileLength], tileLength);
        inQueueX.EnQue(xLocal);
        inQueueY.EnQue(yLocal);
    }
    __aicore__ void Compute(int32_t progress)
    {
        LocalTensor<half> xLocal = inQueueX.DeQue<half>();
        LocalTensor<half> yLocal = inQueueY.DeQue<half>();
        LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
        Add(zLocal, xLocal, yLocal, tileLength);
        outQueueZ.EnQue<half>(zLocal);
        inQueueX.FreeTensor(xLocal); // 释放tensor
        inQueueY.FreeTensor(yLocal);
    }
    __aicore__ void CopyOut(int32_t progress)
    {
        LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
        DataCopy(zGm[progress * tileLength], zLocal, tileLength);
        outQueueZ.FreeTensor(zLocal);
    }
private:
    TPipe pipe;
    TQue<QuePosition::VECIN, bufferNum> inQueueX, inQueueY; // 创建输入队列资源
    TQue<QuePosition::VECOUT, bufferNum> outQueueZ;         // 创建输出队列资源
    GlobalTensor<half> xGm, yGm, zGm;
} « end KernelAdd » ;
extern "C" __global__ __aicore__ void add_kernel(__gm__ uint8_t* x, __gm__ uint8_t* y, __gm__ uint8_t* z)
{
    KernelAdd op;
    op.Init(x, y, z);
    op.Process();
}
```

1，算子调度程序，循环对每个分片数据进行
搬入，计算，搬出操作

2，完成copyin Stage的处理

3，完成compute Stage的处理

4，完成copyout Stage的处理



HUAWEI

AddCustom

Ascend C矢量算子样例代码

1. 组织CPU模式和NPU模式下编译的cmake脚本

2. 为了演示计算结果与测试功能，提供了：
   - 真值数据生成脚本：*gen_data.py*
   - 计算结果校验脚本：*verify_result.py*

3. 为了方便对多个源文件进行编译，提供了：
   - *CMakeLists.txt*

4. 核函数源文件 *add_custom.cpp*

5. 为了使得算子运行调试更加完整，提供了：
   - 读写数据文件辅助函数：*data_utils.h*

6. 为了结构更加清晰，将主机侧和核函数逻辑拆分开：
   - 主机侧源文件：*main.cpp*

7. - 一键化执行脚本：*run.sh*

```
1 ├── cmake
  │   ├── cpu_lib.cmake      // CPU可执行程序编译脚本
  │   └── npu_lib.cmake      // NPU可执行程序编译脚本
2 ├── scripts
  │   ├── gen_data.py        // 测试数据生成脚本
  │   └── verify_result.py   // 结果校验脚本
3 ├── CMakeLists.txt         // 可执行程序编译脚本
4 ├── add_custom.cpp         算子实现文件
5 ├── data_utils.h           // 工具类
6 ├── main.cpp               // 调用程序源码
7 ├── run.sh                 // 编译运行算子的脚本
```

AddCustom —— main.cpp

```
// uint16_t represent half
size_t inputByteSize = 8 * 2048 * sizeof(uint16_t);
size_t outputByteSize = 8 * 2048 * sizeof(uint16_t);
uint32_t blockDim = 8;
```

CPU模式下，主机侧逻辑如下

```
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);

// use this macro for cpu debug
ICPU_RUN_KF(add_custom, blockDim, x, y, z);

WriteFile("./output/output_z.bin", z, outputByteSize);

AscendC::GmFree((void *)x);
AscendC::GmFree((void *)y);
AscendC::GmFree((void *)z);
```

NPU模式下，主机侧逻辑如下所示

```
aclInit(nullptr);
aclrtContext context;
int32_t deviceId = 0;
aclrtSetDevice(deviceId);
aclrtCreateContext(&context, deviceId);
aclrtStream stream = nullptr;
aclrtCreateStream(&stream);

uint8_t *xHost, *yHost, *zHost;
uint8_t *xDevice, *yDevice, *zDevice;
aclrtMallocHost((void**)(&xHost), inputByteSize);
aclrtMallocHost((void**)(&yHost), inputByteSize);
aclrtMallocHost((void**)(&zHost), outputByteSize);
aclrtMalloc((void**)&xDevice, inputByteSize, ACL_MEM_MALLOC_HUGE_FIRST);
aclrtMalloc((void**)&yDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST);
aclrtMalloc((void**)&zDevice, outputByteSize, ACL_MEM_MALLOC_HUGE_FIRST);

ReadFile("./input/input_x.bin", inputByteSize, xHost, inputByteSize);
ReadFile("./input/input_y.bin", inputByteSize, yHost, inputByteSize);

aclrtMemcpy(xDevice, inputByteSize, xHost, inputByteSize,ACL_MEMCPY_HOST_TO_DEVICE);
aclrtMemcpy(yDevice, inputByteSize, yHost, inputByteSize,ACL_MEMCPY_HOST_TO_DEVICE);

add_custom_do(blockDim, stream, xDevice, yDevice, zDevice);
aclrtSynchronizeStream(stream);

aclrtMemcpy(zHost, outputByteSize, zDevice, outputByteSize,ACL_MEMCPY_DEVICE_TO_HOST);
WriteFile("./output/output_z.bin", zHost, outputByteSize);

aclrtFree(xDevice);
aclrtFree(yDevice);
aclrtFree(zDevice);
aclrtFreeHost(xHost);
aclrtFreeHost(yHost);
aclrtFreeHost(zHost);

aclrtDestroyStream(stream);
aclrtDestroyContext(context);
aclrtResetDevice(deviceId);
aclFinalize();
```

数据生成脚本 —— gen_data.py

使用NumPy编写Python脚本，以实现输入数据和真值数据的生成逻辑

```
#!/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()
```

简易工程执行

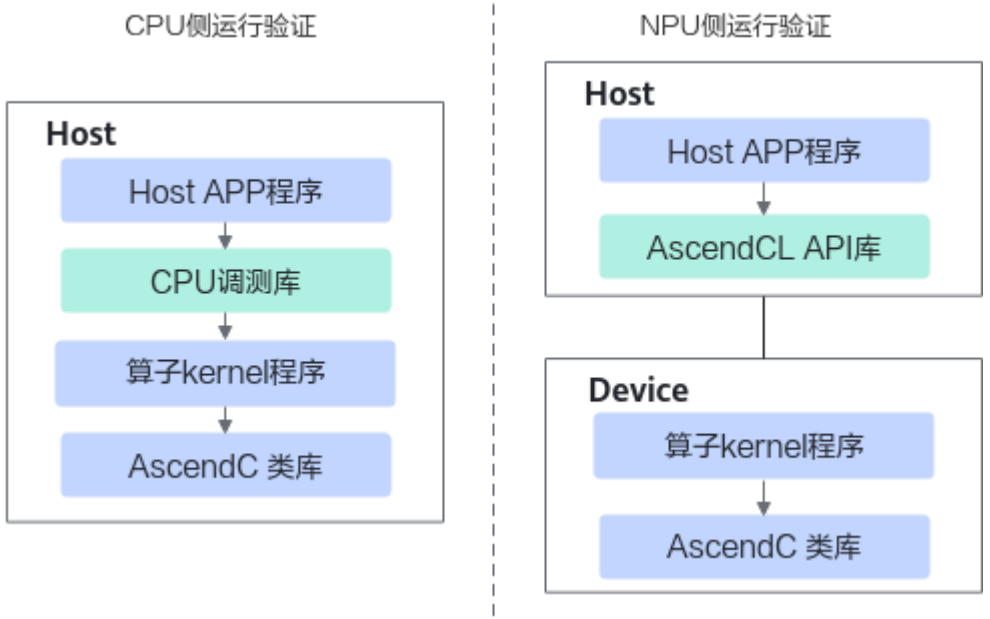
将原先的编译和执行命令封装在的run.sh中，可以直接执行run.sh脚本文件，以实现不同模式下Ascend C算子的运行调试

CPU模式： `bash run.sh -v Ascend910B4 -r cpu`

```
INFO: execute op on cpu succeed!
md5sum:
d28ba77baccfc73a686b52204d64f8c2  output/golden.bin
d28ba77baccfc73a686b52204d64f8c2  output/output_z.bin
```

NPU模式： `bash run.sh -v Ascend910B4 -r npu`

```
INFO: compile op on npu succeed!
INFO: execute op on npu succeed!
md5sum:
26845c5259e605db104912003 92f9552  output/golden.bin
26845c5259e605db104912003 92f9552  output/output_z.bin
```



HUAWEI

•演示实验及小测试

演示：基于Add核函数调用，实现Sub算子功能。

算子类型（OpType）	Sub			
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	sub_custom			

计算逻辑：z=x-y

小测试：基于Add核函数调用，实现Sinh算子功能

算子类型（OpType）	Sinh			
算子输入	name	shape	data type	format
	x	(8, 2048)	half	ND
算子输出	z	(8, 2048)	half	ND
核函数名	sinh_custom			

计算逻辑：z=(exp(x) – exp(-x))/2

Thank you.

昇腾开发者社区

http://hiascend.com

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI