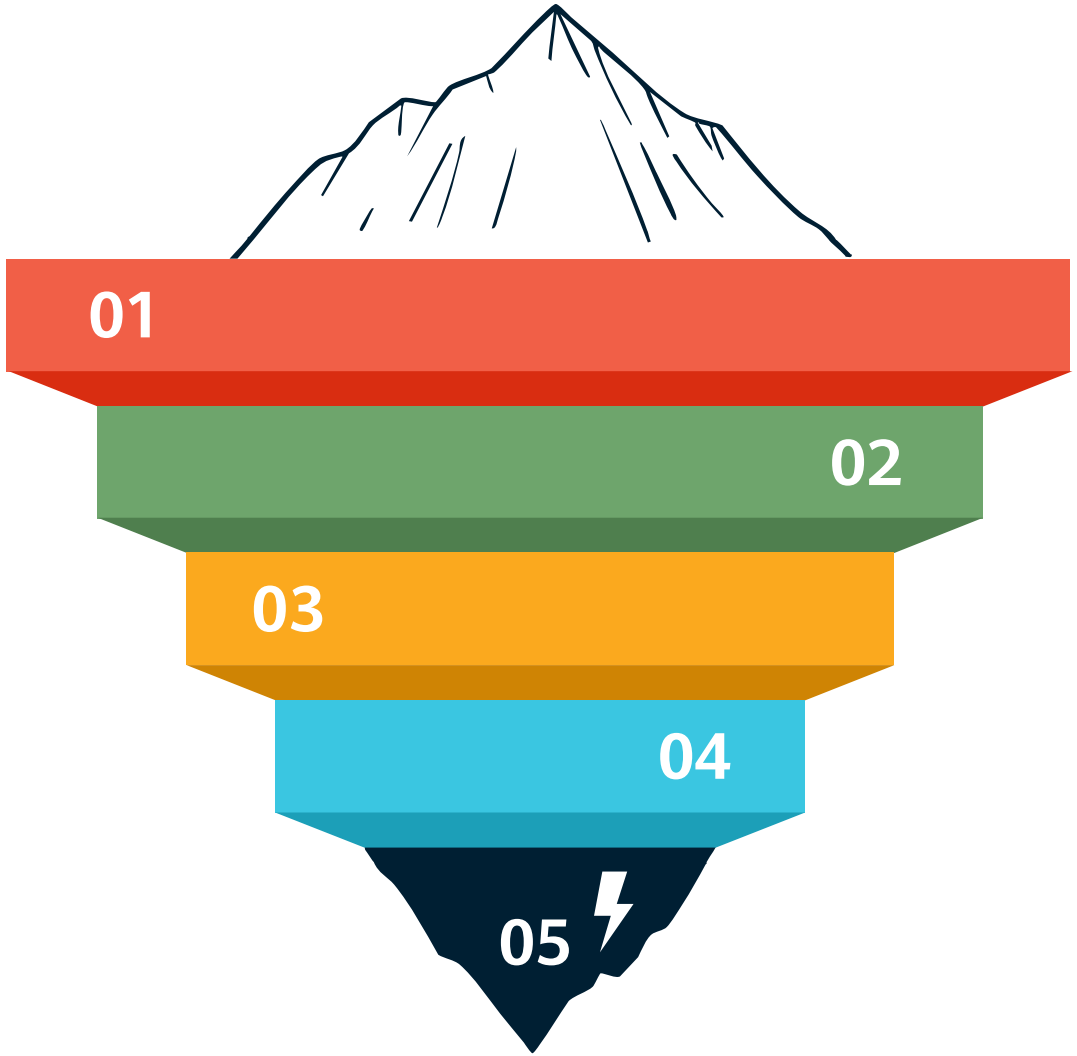
ops-samples 开源介绍

作者：张子杰、林鹏翔

时间：2026/02/05

CANN

算子开发的冰山模型



从功能跑通进阶到极致性能，才真正踏入了算子开发的**深水区**
越向深处，挑战越大

01 功能跑通	算法逻辑正确 / 编译通过
02 资源利用	多核并行 / Tiling设计 / 负载均衡 / 流水并行
03 访存优化	合并访存 / UB缓存 / 对齐访问 / 数据预取 / L2 Cache
04 指令调优	寄存器规划 / 指令精简 / 循环展开 / VF融合
05 系统优化	AICPU计算Tiling / Kernel启动开销优化

ops-samples: 1-100分的进阶指导

全景入门教程

入门教程

samples (构建中)

为CANN初学者提供0->1的系统化学习路径、快速入门以及参考实现。

组件功能示例

集中样例仓

ops-samples

提供算子领域高性能实训演进样例与体系化调优知识库，让开发者系统性掌握从逻辑实现到精细化调优的演进路径。

组件examples目录

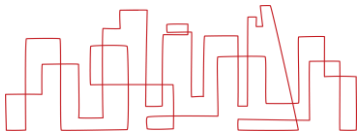
ops-math/examples

ops-nn/examples

ops-cv/examples

ops-transformer/examples

让开发者快速上手的简易调用样例



行业落地案例库

Recipes系列

cann-recipes-infer

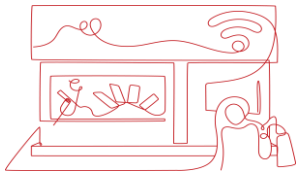
cann-recipes-train

cann-recipes-embodied-intelligence

cann-recipes-spatial-intelligence

cann-recipes-harmony-infer

模型落地最佳实现，提供典型模型的优化样例



https://gitcode.com/cann

CANN

ops-samples简介

背景和目标

背景

1. 算子开发门槛不高，高性能算子开发门槛高，开发者性能调优有难度。
2. 大模型关键算子复杂，优化手段多。
3. 关键算子有深度定制需求，追求卓越性能。

目标

1. 提供算子领域高性能实训演进样例与体系化调优知识库。
2. 让开发者系统性掌握从逻辑实现到精细化调优的演进路径。
3. 提供关键算子的优化细节。

https://gitcode.com/cann

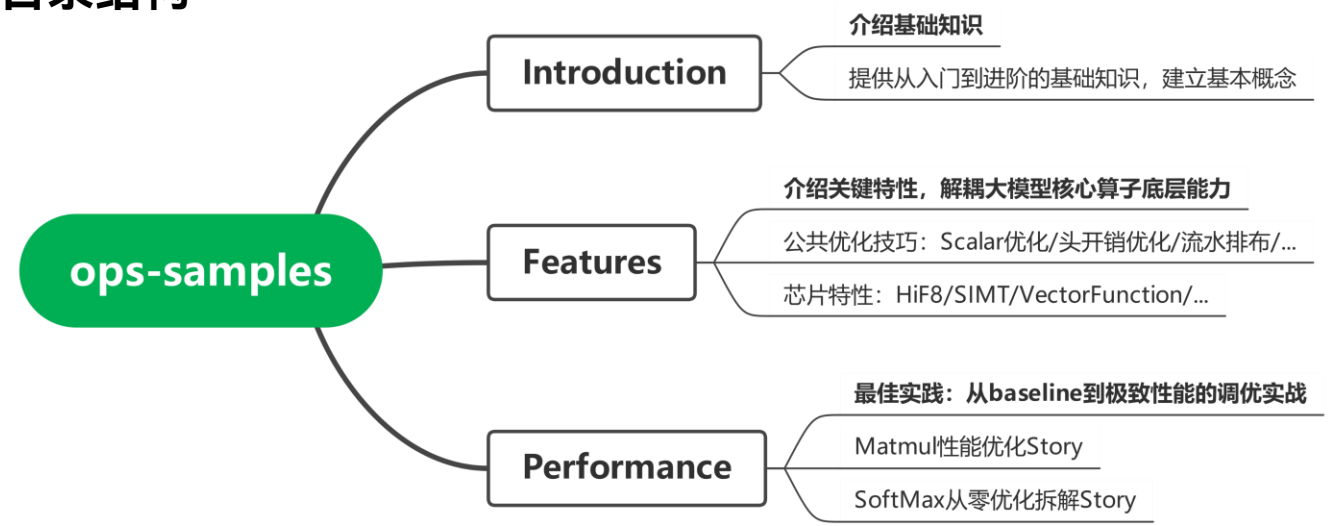
代码仓全景

三大板块

1. Introduction: 介绍基础知识，建立基本概念，补全从入门到精通的知识空缺。
2. Features: 介绍关键特性，解耦大模型核心算子底层能力。包括公共优化技巧和关键芯片特性。
3. Performance: 最佳实践，从baseline到极致性能的调优实践。包含中间态代码，支持独立调试学习。

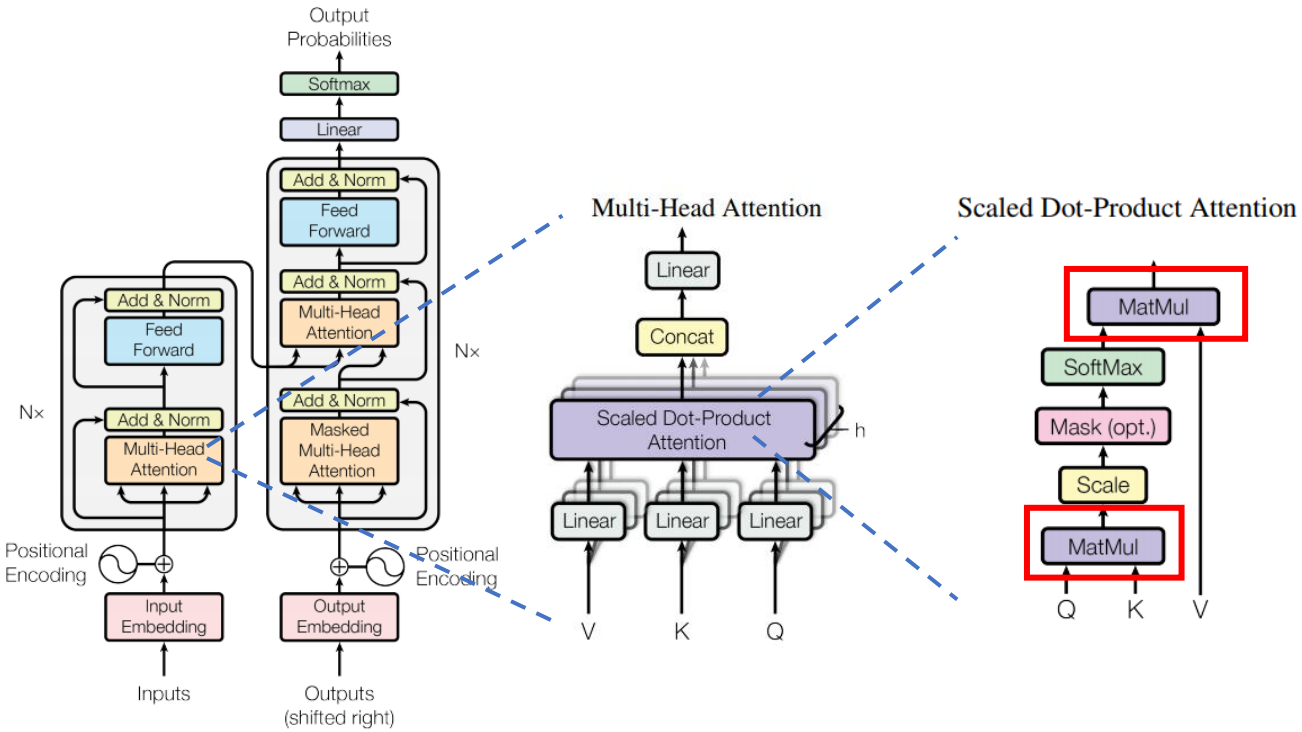
针对冰山模型的深水区挑战，ops-samples 将隐性的调优经验转化为显性的代码资产，构建了阶梯式的成长路径。

目录结构



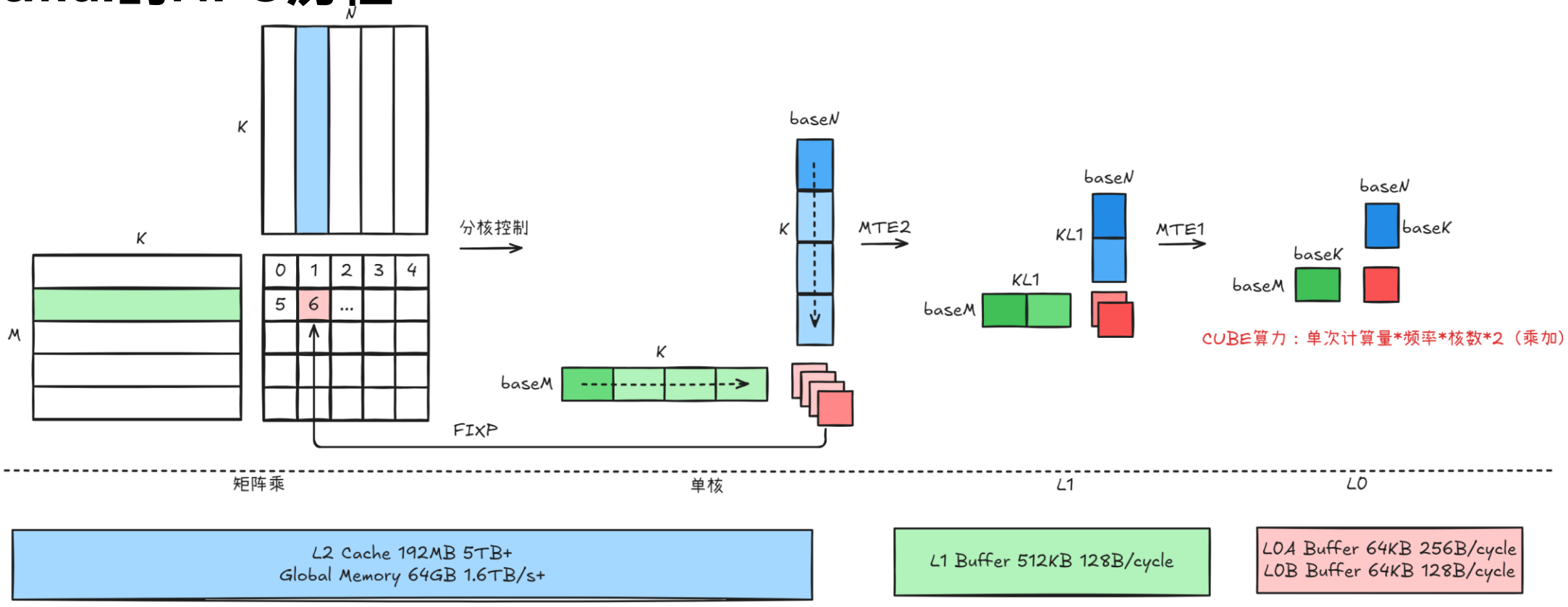
为什么聚焦矩阵乘优化？

当前主流大模型基本都建立在Transformer架构和扩散模型两个基石之上，无论前者的Self-Attention和Feed Forward，还是后者的UNet网络，其本质核心都是矩阵乘运算，**矩阵乘的占比高达90%+**。优化矩阵乘的性能，对于整网的性能提升收益比最高。



网络类型	核心矩阵乘法占比	主要瓶颈来源
全连接网络（FCN）	95%+	几乎全部为线性层，无卷积/注意力等操作。
Transformer/LLM	80%~95%	注意力机制和FFN中的矩阵乘法。
CNN	60%~85%	卷积操作（本质为局部矩阵乘）。
轻量网络（EfficientNet-Lite）	40%~60%	拆分结构多，低维操作，非线性操作占比高。

Matmul的NPU历程



矩阵乘中核心任务

- 数据在内存中的逐层搬运
- 数据在多核中的并行计算

算子性能瓶颈

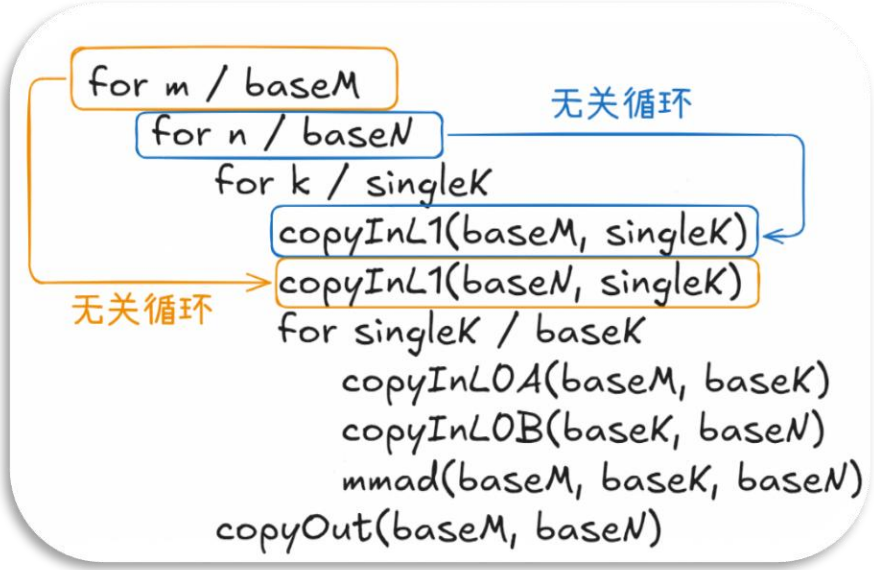
- **搬运瓶颈**
- **计算瓶颈**
- **流水阻塞**

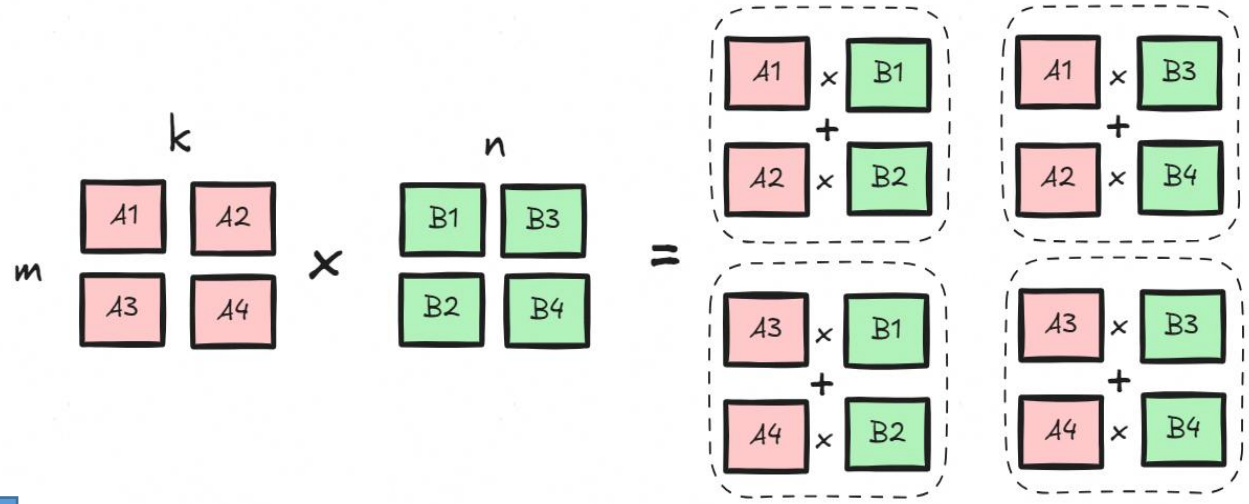
三类优化措施

- 搬运优化：包括减少搬运量，提升搬运效率
- 计算优化：提高多核计算负载均衡率
- 指令并行度优化：提高搬运/计算指令的并行能力

减少搬运量 - 决策基本块

由于BufferSize的限制，完成一次矩阵乘需要将输入切成若干基本块循环载入，从而引入数据的重复搬运量。





主要受限于L0A/L0B/L0C Size大小；
在baseM=baseN时可以取到最大；
A2/A3指导设置(128, 256) or (256, 128)

$$\text{GM2L1搬运量} = \left(\frac{n}{baseN}\right) \times m \times k + \left(\frac{m}{baseM}\right) \times n \times k = \left(\frac{1}{baseM} + \frac{1}{baseN}\right) \times m \times k \times n$$

优化策略：性能被搬运阻塞时，可以通过提高$\frac{1}{baseM} + \frac{1}{baseN}$来减少整体搬运量，从而优化搬运耗时。

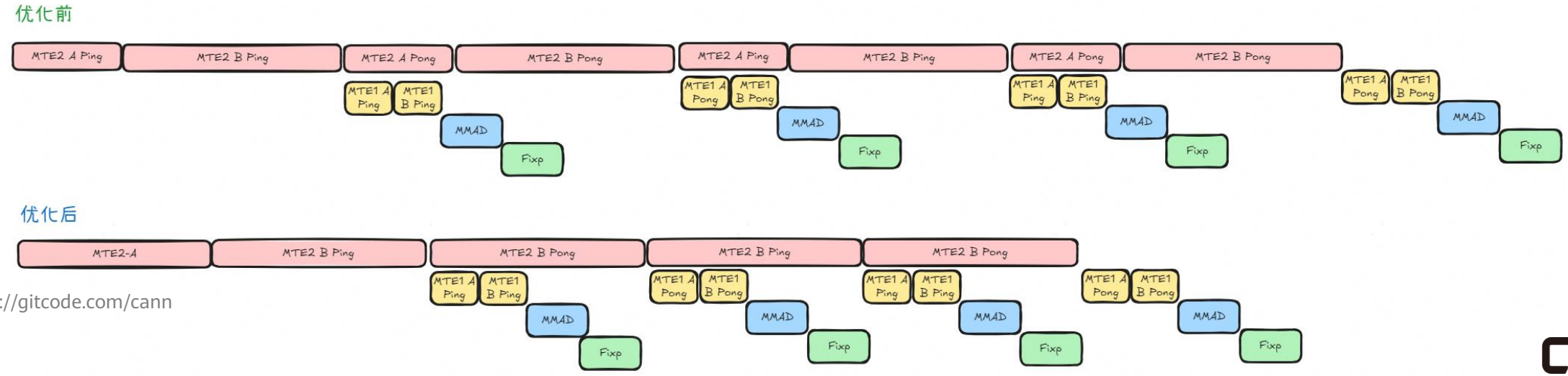
减少搬运量 - L1全载

当一侧输入可以全载到L1 Buffer时，可以让数据全载驻留到L1中，减少GM2L1的循环搬运量，让这部分搬运由带宽更高的L0 Buffer承担。

```
copyInL1(m, k)  驻留L1
    for n / baseN
        for k / singleK
            copyInL1(baseN, singleK)
            for m / baseM
                for singleK / baseK
                    copyInL0A(baseM, baseK)
                    copyInL0B(baseK, baseN)
                    mmad(baseM, baseK, baseN)
        copyOut(baseM, baseN)
```

应用场景：多应用于推理Decode场景，BatchSize较小时满足左矩阵全载L1 Buffer。此时激活侧多核各载一次，权重侧多核只载一次。

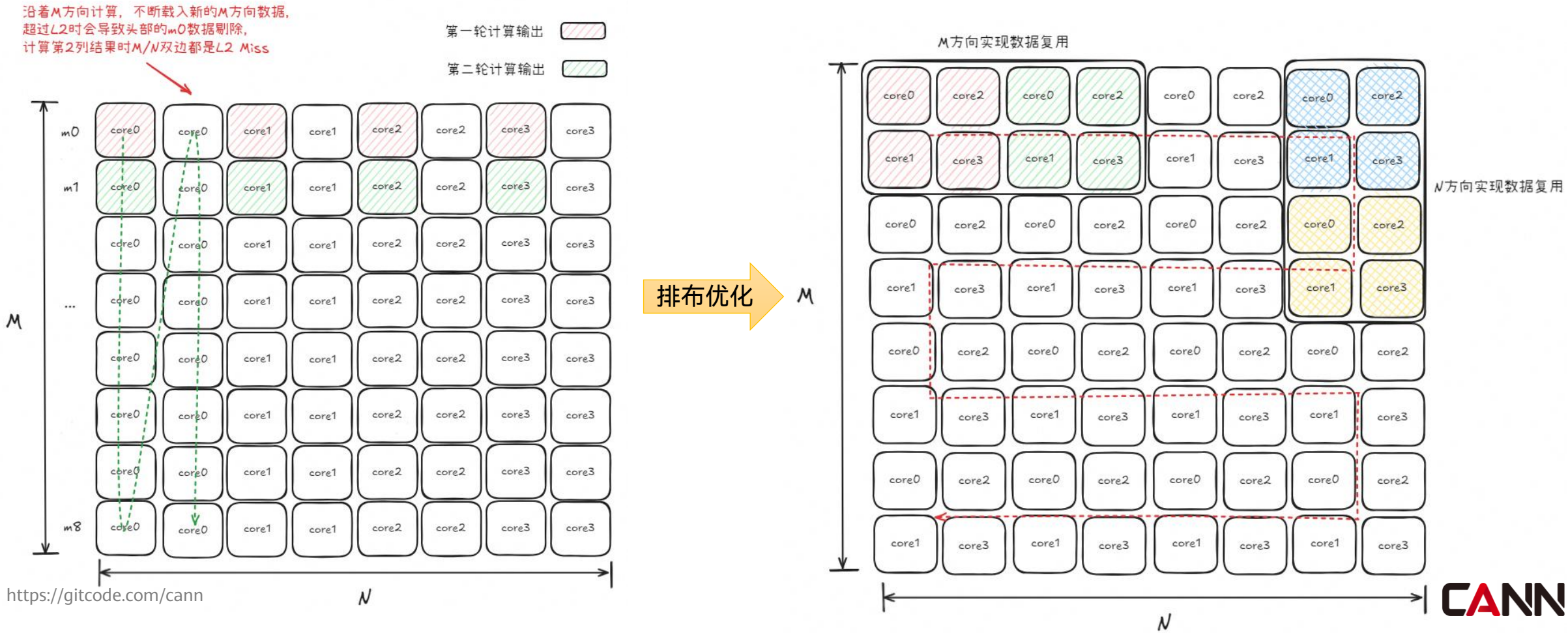
全核GM2L1搬运量$= m \times k \times \text{core}Num + n \times k$



提升搬运效率 - 增加L2命中率

根据内存规格，L2带宽速率是Global Memory的4~5倍。因此访问同样数据量的情况下，L2的占比越高，整体带宽就越高，更容易满足MMAD计算速率要求。然而，由于L2 Cache的空间有限（192MB），当输入矩阵超过L2 Size时，新搬入的数据会替换旧数据，导致重新读取旧数据时需要再从Global Memory获取，降低整体带宽效率。

可以通过排布优化，来降低数据替换的概率，提高L2命中率。

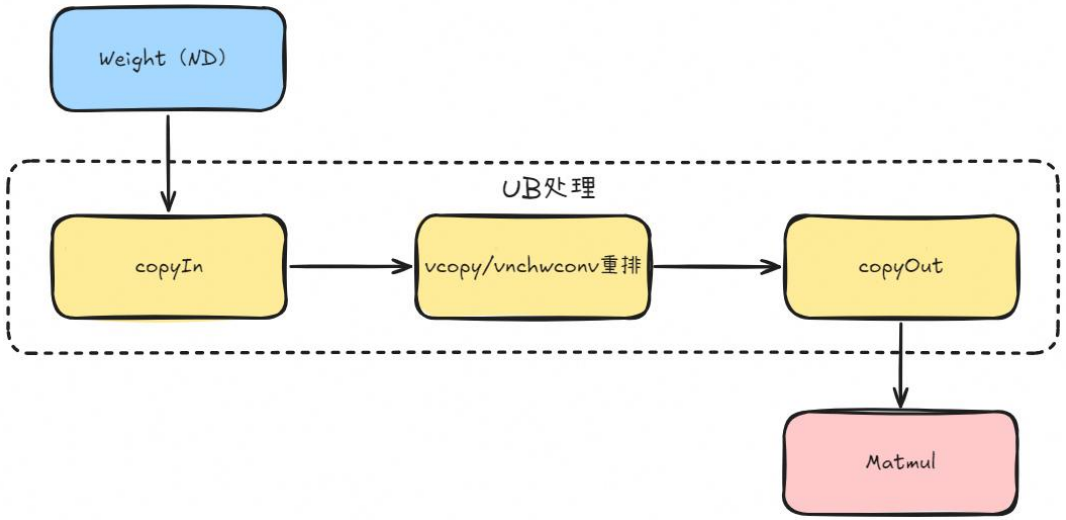


提升搬运效率 - 亲和排布

矩阵默认输入为ND排布，搬运到L1 Buffer需要通过指令实现ND转NZ；当前随路的ND2NZ指令存在小包搬运/拆包等行为，带宽利用率不及亲和排布NZ。因此可以通过优化转NZ方式来提升搬运效率。

训练场景

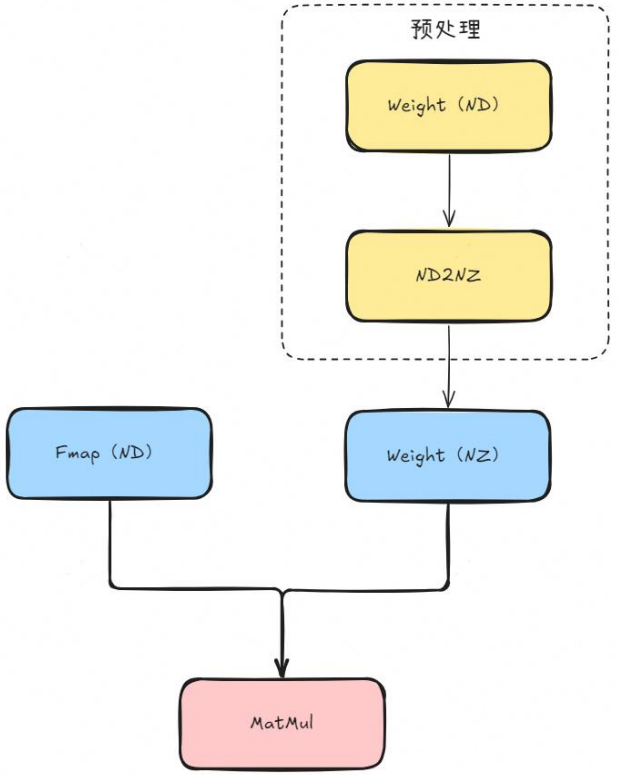
特点：输入无法预处理，依赖算子内部手动重排格式



效果：通过大包搬运UB提高搬运效率，进而通过切分实现UB前处理和Matmul流水并行，提高流水并行度。

推理场景

特点：权重支持预处理，可提前优化成目标排布

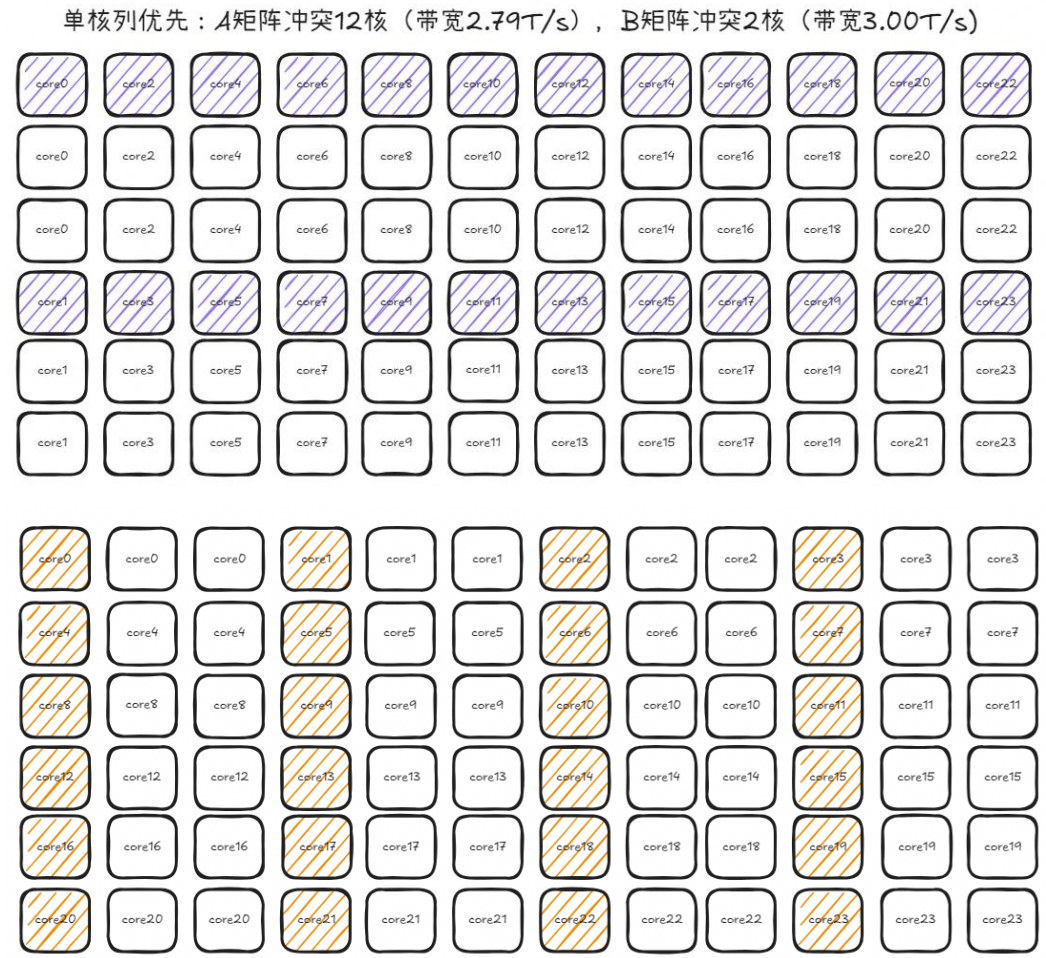


效果：Prefill场景提升10%，Decode场景提升13%

提升搬运效率 - 同地址冲突

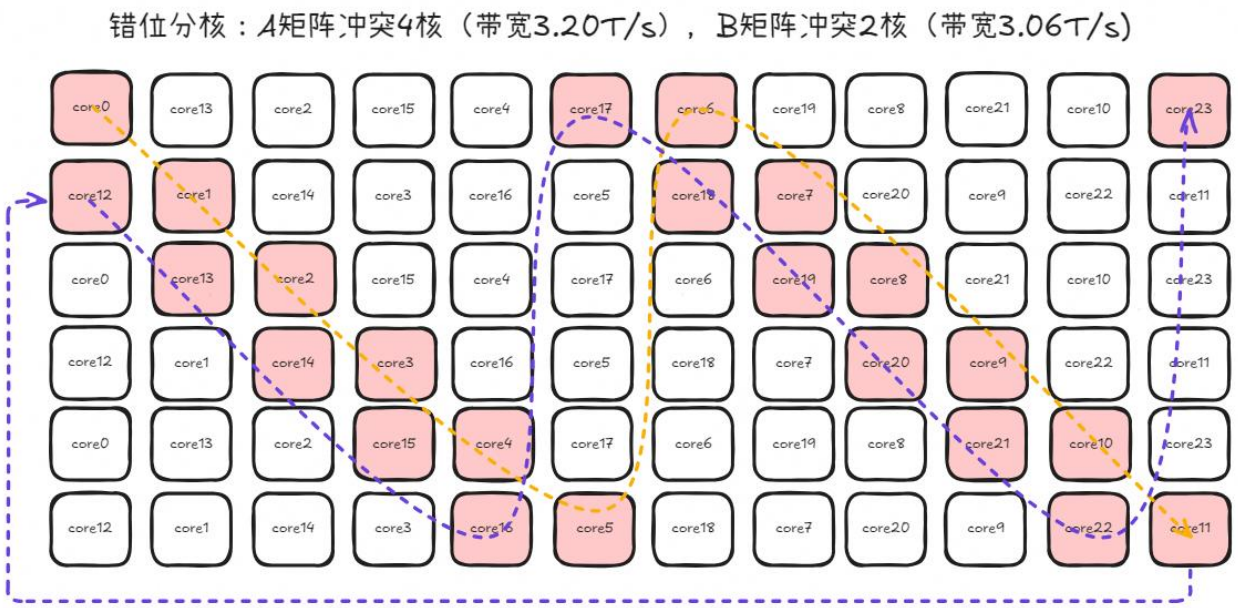
每个channel包含多个bank，但同一bank无法同时处理多个请求(需先关闭当前行才能访问新行）。

冲突场景：若多个请求访问同一channel的同一bank，需排队处理，导致bank conflict，影响搬运效率。



传统行列优化，同地址冲突多，带宽效率低

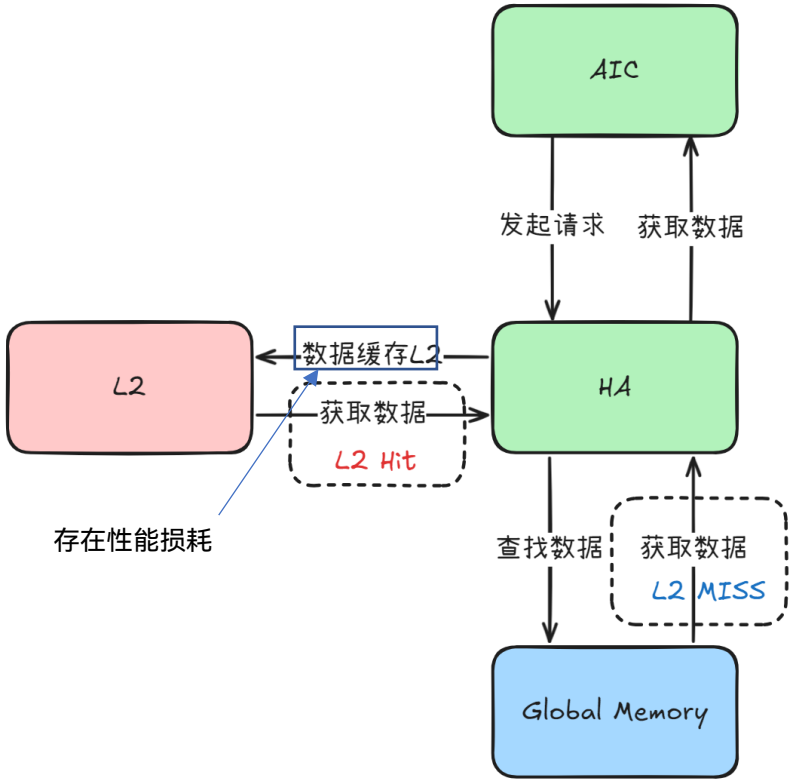
M/N方向自定义控制错位分核，减少同地址冲突



PS：该方案会降低L2命中率，因此需要权衡地址冲突和L2利用率的影响

提升搬运效率 - 双页表

默认场景下，首次从Global Memory中获取的数据都会同步缓存到L2中，方便重复访问的时候能够使用L2的高带宽。
但是对于只访问一次的数据，同步L2不会带来长远的正收益，因此可以将初次同步的损耗屏蔽掉，获取极致性能。



使能约束：数据仅从Global Memory搬运一次；
使能方法：给数据地址加上特定的偏移，从而让框架识别，屏蔽该地址数据的L2同步缓存。

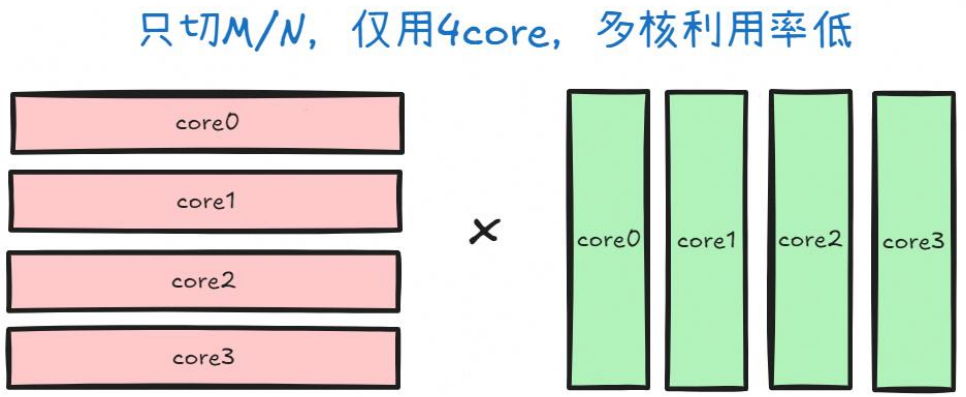
```
template <class A_T, class B_T, class C_T, class BiasT>
__aicore__ inline void SetL2CacheEnable(const L2cacheUseInfo& l2EnableInfo,
    GlobalTensor<A_T> &aGlobal, GlobalTensor<B_T> &bGlobal,
    GlobalTensor<C_T> &cGlobal, GlobalTensor<BiasT> &biasGlobal) {

    if ((l2EnableInfo.l2CacheFlag & ALL_L2_CACHE_ENABLE) == 0) {
        if (l2EnableInfo.l2CacheFlag & C_L2_DISABLE) {
            cGlobal.SetL2CacheHint(CacheMode::CACHE_MODE_DISABLE);
        }
    }
}
```

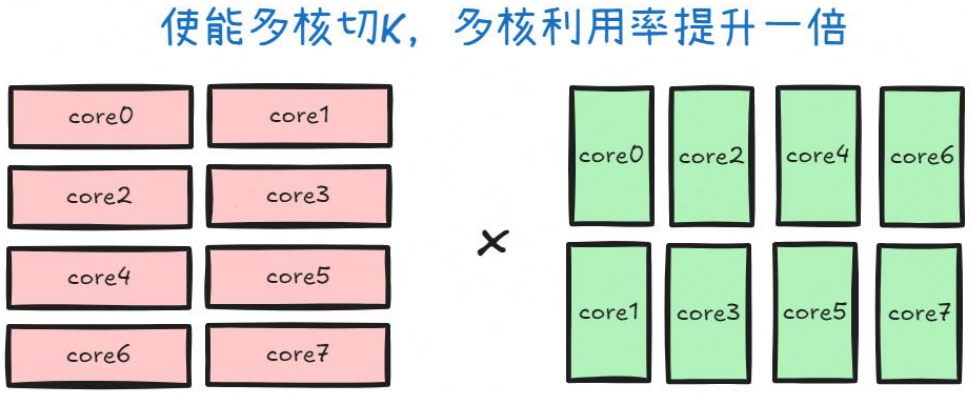
设置方法

计算负载均衡 - 多核切K

在算力固定的情况下，一个用例的计算时间是固定的。但是如果计算负载不均衡，会导致算子耗时和理论耗时存在较大偏差，进而导致计算利用率降低。

典型场景有当M/N较小时，按照最优基本块无法分满多核，从而出现多核负载不均衡情况。此时通过多核切K可有效改善这类场景。






```
//切K代码示例
tileNum = Div(m, baseM) * Div(n, baseN) * Div(k, baseK);
if ASCEND_IS_AIC {
    for (int64_t tileIdx = curBlockIdx; tileIdx < tileNum; tileIdx += usedCoreNum) {
        for (uint64_t iter0 = 0; iter0 < curKL1Iter; ++iter0) {
            CopyInA1();
            CopyInB1();
            for (uint64_t iter1 = 0; iter1 < kL0Iter; ++iter1) {
                CopyInA2();
                CopyInB2();
                Mmad();
            }
            CopyOut();
        }
    }
}
if ASCEND_IS_AIV {
    DataCopyPad<>();
    for (uint64_t i = 1; i < Div(k, baseK); ++i) { // 在ub上按序累加
        Add();
    }
    DataCopyPad<>();
}
```

以shape为((32, 4096), (4096, 32))矩阵切K为例：

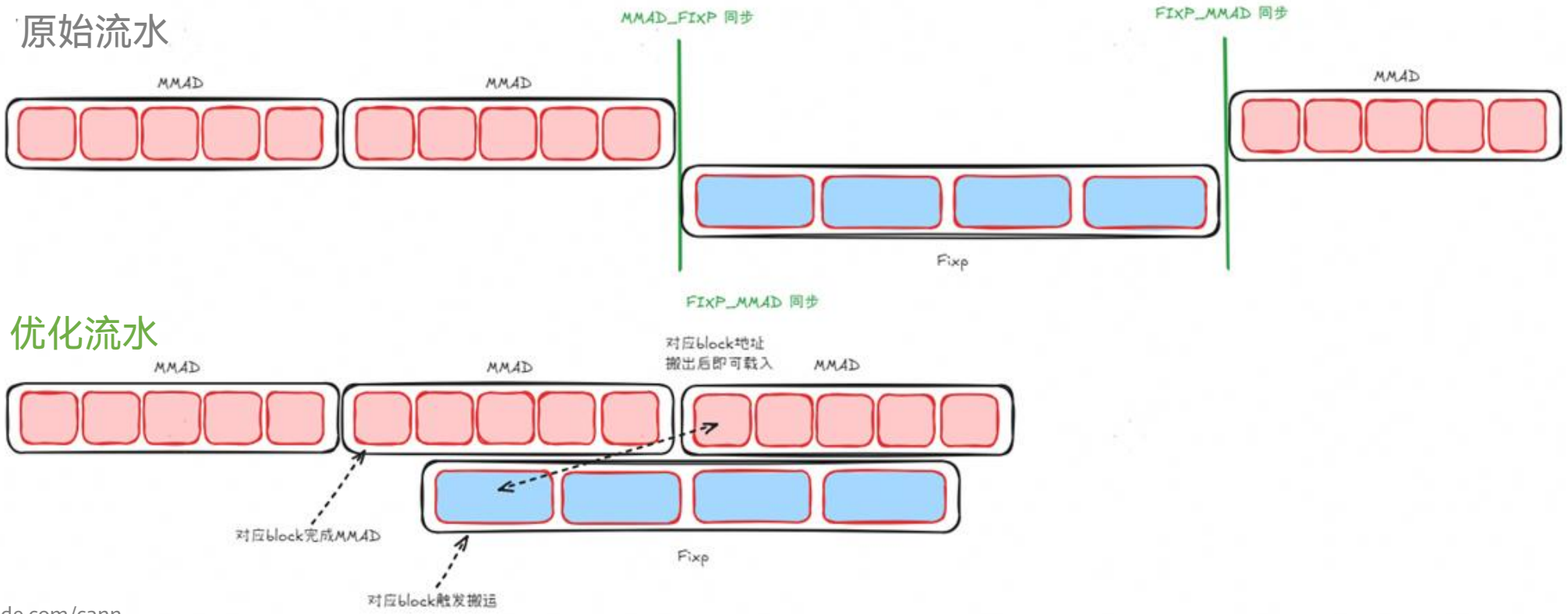
- 切K前：若使用普通代码逻辑，运行耗时需要12.17us。
- 切K后：采用切K模板代码逻辑，运行耗时需要8.44us。

CANN

Unitflag

为了提高计算访存比，降低对带宽的需求，要求选择更大的baseM/baseN，导致L0C Buffer无法开启Double，进而引起MMAD和FIXP指令因同步关系存在断流。

为了解决这类问题，当前支持通过Unitflag能力，实现MMAD指令和FIXP指令从指令级的同步转变成Block级同步。当最后一条MMAD指令计算完一个Block后，对应Block的矩阵输出结果已经完成计算，FIXP就可以从L0C搬出，优化流水如下所示：



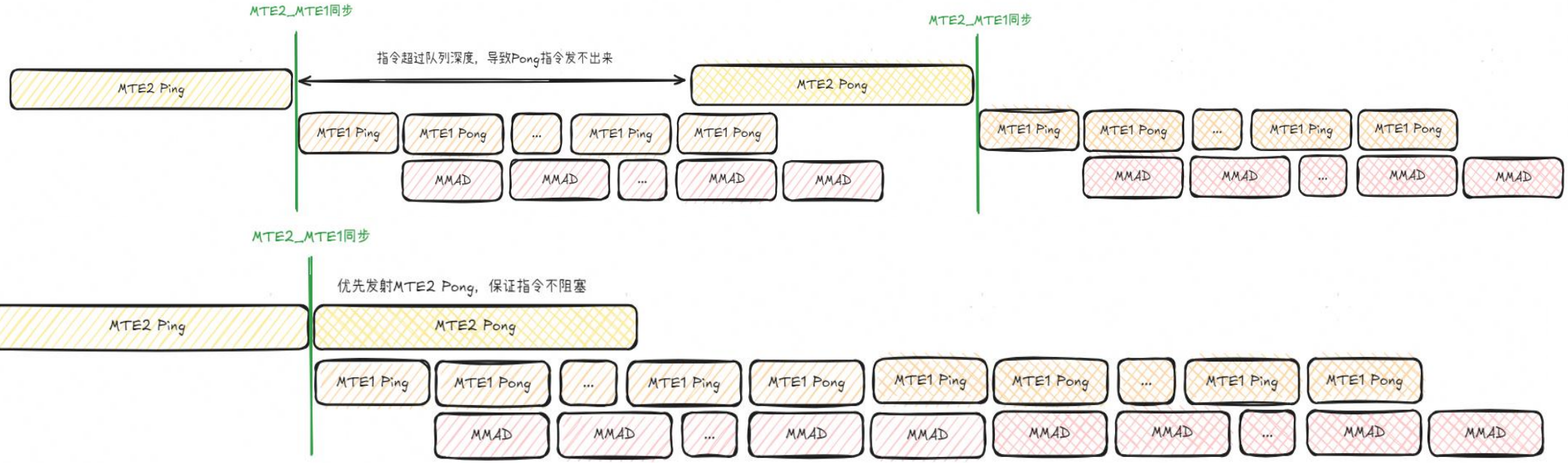
Preload

算子实际运行中，PC指针通过逐行读取指令，将指令放置在一个总队列当中。当一条指令的前置指令太多，就会导致后续的指令出现阻塞。此时即使没有同步约束，指令依然无法发射，导致流水并行度降低。

为此可以优化指令发射的顺序，让没有同步依赖的指令提前发射，从而提高流水并行度。

传统写法

```
MTE2_PING()
    MTE1_PING()
    MMAD_PING()
    MTE1_PONG()
    MMAD_PONG()
    ...
MTE2_PONG()
    ...
```



优化写法

```
MTE2_PING()
MTE2_PONG() // 优先发i
    MTE1_PING()
    MMAD_PING()
    MTE1_PONG()
    MMAD_PONG()
    ...
MTE2_PING() // 继续优先发送下一轮PING的MTE2
    MTE1_PING()
    MMAD_PING()
    MTE1_PONG()
    MMAD_PONG()
    ...
```


CANN

性能瓶颈量化分析

各流水理论耗时

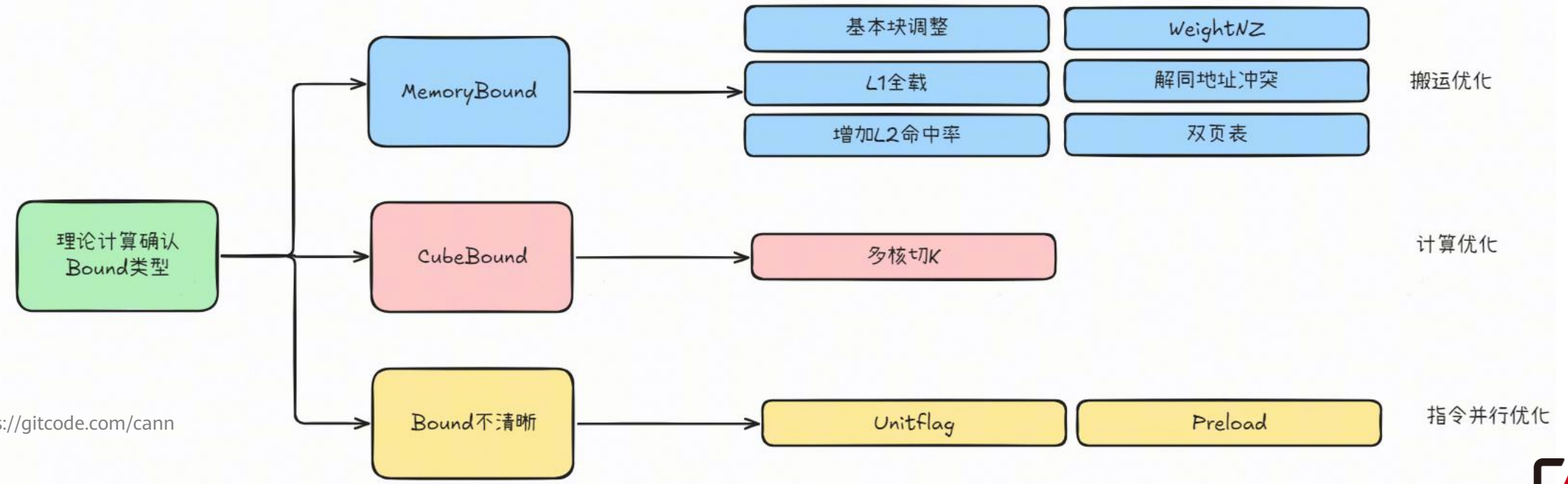
MTE2搬运耗时$=(\frac{1}{baseM}+\frac{1}{baseN})\times m\times k\times n\div \text{BandWidth}$

CUBE计算耗时$=m\times k\times n\div \text{ComputePower}$

FIXPIPE搬出耗时$=m\times n\div \text{BandWidth}$

Bound类型判断

- **MemoryBound**：MAX(MTE2, FIXPIPE) > CUBE
- **CubeBound**：CUBE >=MAX(MTE2, FIXPIPE)



Features - 访存、指令、芯片与系统级优化全景图谱

设计理念与方法论

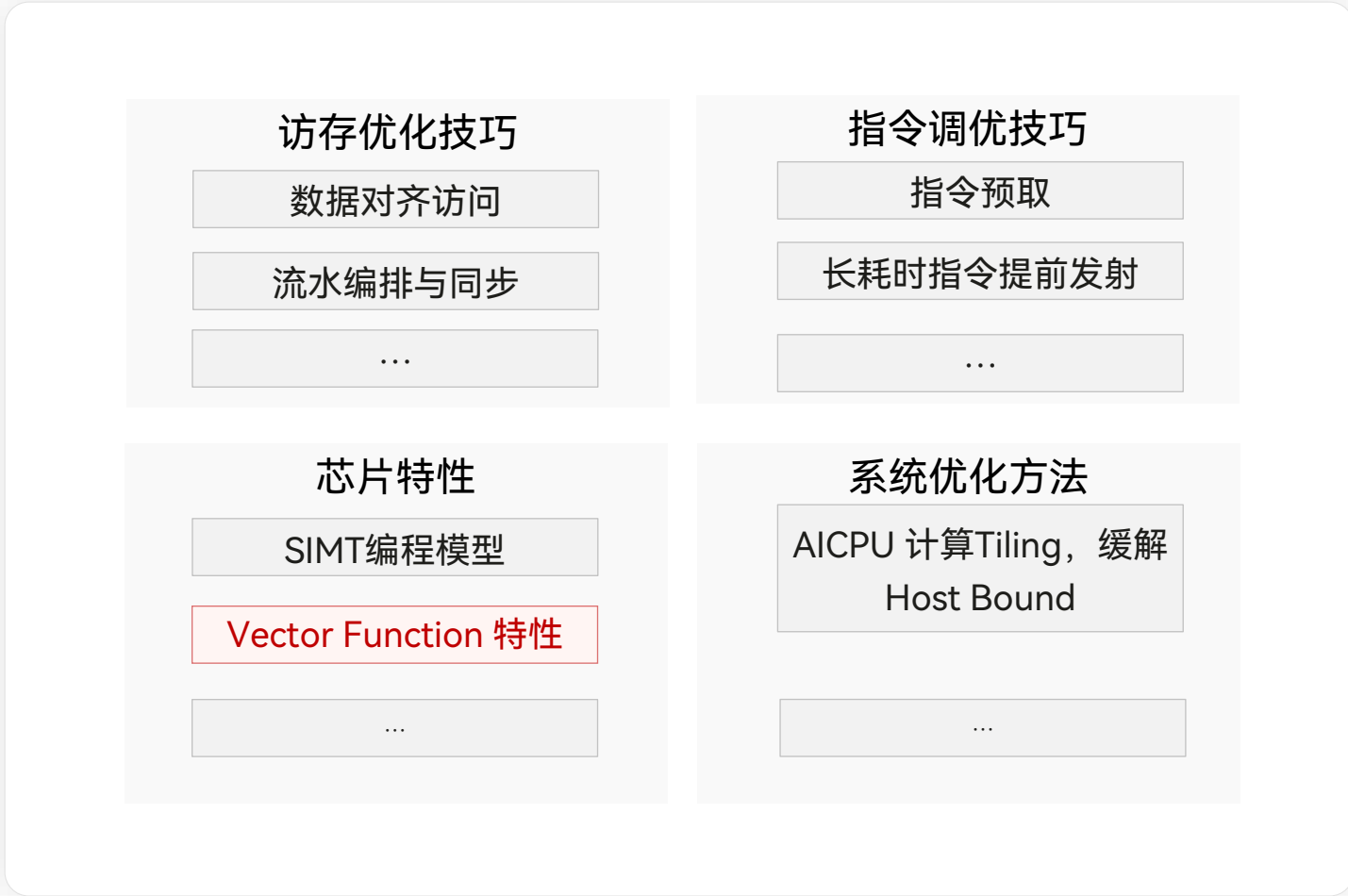
痛点

1. 算子代码高度工程化，难以剥离单个特性。
2. 底层特性涉及多点修改，不易完整掌握。
3. 芯片相关特性，阅读代码理解其底层原理有难度。

解法

1. 单一职责，聚焦核心。每个Sample仅演示1个特性技术点，裁剪无关的工程代码，确保核心逻辑清晰。
2. 模块化，即插即用。单一Sample的核心代码片段，可以嵌入到自定义算子中。
3. 极简闭环。针对芯片相关特性，提供剥离业务背景的独立可执行样例，打通“硬件原理”到”代码实现”的最后一公里。

Features规划

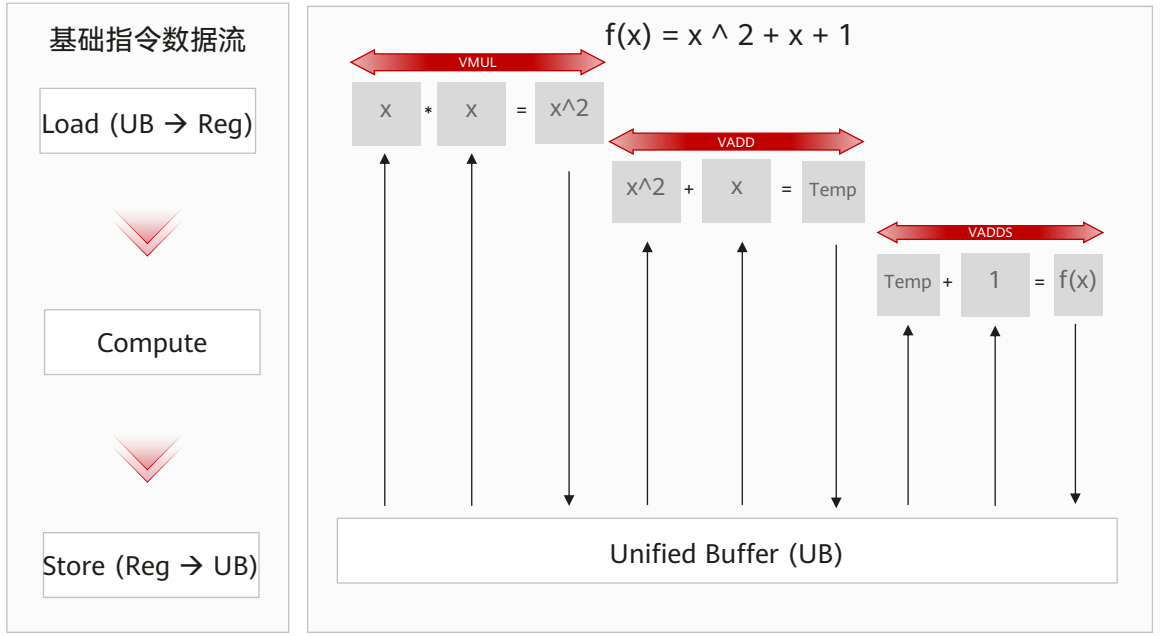


Vector Function(VF) 特性概览

Memory Based 编程模型

- 在Memory Based编程模型中，向量计算以UB为存储媒介，向量指令遵循”读-算-写” 的处理范式，多步复合计算中间结果需要频繁写回UB。
- 技术特征：显式内存管理与手动维护数据依赖，在代码中插入PipeBarrier指令，保障向量指令间的数据依赖。
- 局限性：冗余访存挤占UB带宽，中间结果反复读写，计算单元因等待数据而出现闲置。

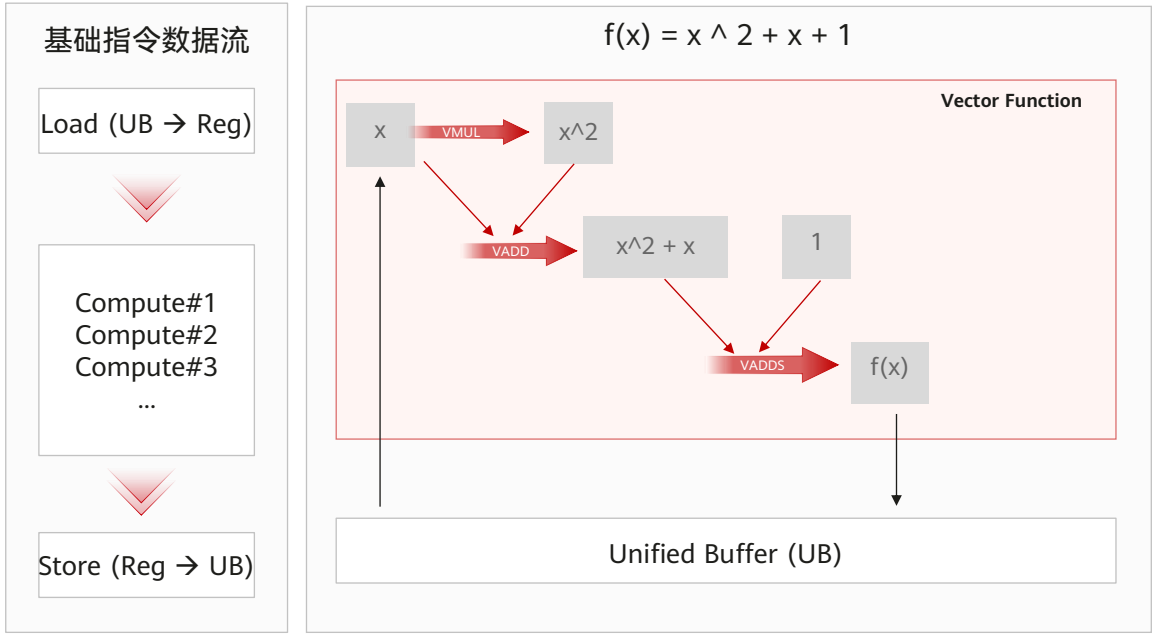
基础数据流



Register Based 编程模型

- **Vector Function**是Register Based编程模型中的重要概念。它以寄存器为核心计算载体，支持数据驻留与指令级的计算融合，从而消除冗余的UB访存开销。
- 技术特征：
  - 显式寄存器管理与计算融合，利用向量寄存器实现数据传递，中间结果在寄存器内部高效流转。
  - 硬件自动处理寄存器依赖关系，消除显式PipeBarrier.
  - 指令双发、乱序执行，最大限度的掩盖指令延迟，提高计算吞吐。
- 性能优势：突破访存瓶颈、提高计算单元利用率。

基础数据流



实训 - 使用Vector Function计算融合加速GeLU

Memory Based 编程模型

■GeLU的计算公式(使用Tanh近似)

$$GELU(x) = 0.5 * x * (1 + \mathrm{Tanh}(\sqrt{2/\pi} * (x + 0.044715 * x^3)))$$

- 不使用Vector Function的传统实现

```
__aicore__ void gelu_compute(
  const AscendC::LocalTensor<float> &xLocal,
  const AscendC::LocalTensor<float> &yLocal,
  const AscendC::LocalTensor<float> &xCube,
  const AscendC::LocalTensor<float> &tLocal,
  int64_t n)
{
    const float NEG_SQRT_EIGHT_OVER_PI = -1.595769121 * 0.044715;
    const float TANH_APPROX_FACTOR = 1 / 0.044715;
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Mul(xCube, xLocal, xLocal, n);
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Mul(xCube, xCube, xLocal, n);
    AscendC::Muls(tLocal, xLocal, TANH_APPROX_FACTOR, n);
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Add(xCube, xCube, tLocal, n);
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Muls(xCube, xCube, NEG_SQRT_EIGHT_OVER_PI, n);
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Exp(xCube, xCube, n);
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Adds(xCube, xCube, 1.0f, n);
    AscendC::PipeBarrier<PIPE_V>();
    AscendC::Div(yLocal, xLocal, xCube, n);
    AscendC::PipeBarrier<PIPE_V>();
}
```

单核处理409600 Float32数据耗时：**69.2us**

Register Based 编程模型

■使用Vector Function的计算融合能力，可以将GeLU的计算融合在一个VF中，实现高性能的Vector计算

- 使用Vector Function的优化实现

```
__aicore__ void gelu_compute(
  const AscendC::LocalTensor<float> &xLocal,
  const AscendC::LocalTensor<float> &yLocal,
  const AscendC::LocalTensor<float> &xCube,
  const AscendC::LocalTensor<float> &tLocal,
  int64_t n)
{
    const float NEG_SQRT_EIGHT_OVER_PI = -1.595769121 * 0.044715;
    const float TANH_APPROX_FACTOR = 1 / 0.044715;
    uint32_t vectorLength = AscendC::VECTOR_REG_WIDTH / sizeof(float);
    uint32_t loopNum = (n + vectorLength - 1) / vectorLength;
    __VEC_SCOPE__
    {
        __ubuf__ float *xAddr = (__ubuf__ float *)xLocal.GetPhyAddr();
        __ubuf__ float *yAddr = (__ubuf__ float *)yLocal.GetPhyAddr();
        AscendC::MicroAPI::MaskReg pMask;
        AscendC::MicroAPI::RegTensor<float> xReg, yReg, cubeReg, tReg;
        uint32_t count;
        count = static_cast<uint32_t>(n);
        for (uint16_t i = 0; i < loopNum; ++i) {
            pMask = AscendC::MicroAPI::UpdateMask<float>(count);
            AscendC::MicroAPI::DataCopy<float, AscendC::MicroAPI::LoadDist::DIST_NORM>(
                xReg, (__ubuf__ float *)xAddr + i * vectorLength);
            AscendC::MicroAPI::Mul(cubeReg, xReg, xReg, pMask);
            AscendC::MicroAPI::Mul(cubeReg, cubeReg, xReg, pMask);
            AscendC::MicroAPI::Muls(tReg, xReg, TANH_APPROX_FACTOR, pMask);
            AscendC::MicroAPI::Add(cubeReg, cubeReg, tReg, pMask);
            AscendC::MicroAPI::Muls(cubeReg, cubeReg, NEG_SQRT_EIGHT_OVER_PI, pMask);
            AscendC::MicroAPI::Exp(cubeReg, cubeReg, pMask);
            AscendC::MicroAPI::Adds(cubeReg, cubeReg, 1.0f, pMask);
            AscendC::MicroAPI::Div(yReg, xReg, cubeReg, pMask);
            AscendC::MicroAPI::DataCopy<float, AscendC::MicroAPI::StoreDist::DIST_NORM_B32>(
                (__ubuf__ float *)yAddr + i * vectorLength, yReg, pMask);
        }
    }
}
```

单核处理409600 Float32数据耗时：**25.5us**

加速比：**2.7x**

Roadmap

2026.02.05

首次开源

短期规划

核心板块构建

- 知识内容分层
- 规范文档标准
- 丰富Features案例
- 更多Performance故事

中期规划

紧跟前沿技术

- 大模型算子专题
- 新代际芯片介绍
- 调优工具链集成
- AI驱动知识库(Skills)

长期愿景

生态共建，智能演进

- 确立算子调优方法论
- 探索自动调优
- 建立社区自治体系

https://gitcode.com/cann

CANN

欢迎使用和贡献

代码仓地址： https://gitcode.com/cann/ops-samples

Thank you.

社区愿景：打造开放易用、技术领先的AI算力新生态

社区使命：使能开发者基于CANN社区自主研究创新，构筑根深叶茂、跨产业协同共享共赢的CANN生态

Vision: Building an Open, Easy-to-Use, and Technology-leading AI Computing Ecosystem

Mission: Enable developers to independently research and innovate based on the CANN community and build a win-win CANN ecosystem with deep roots and cross-industry collaboration and sharing.

上CANN社区获取干货

关注CANN公众号获取资讯

https://gitcode.com/cann

CANN