CANN ops-cv仓介绍及相关优秀实践分享

https://gitcode.com/cann

CANN

目录

ops-cv仓介绍

ops-cv是CANN（Compute Architecture for Neural Networks）算子库中提供图像处理、目标检测等能力的高阶算子库，涵盖常见的图像处理操作，包括image类、objdetect类

- **image类**：图像处理类算子，例如Resize、 Upsample、GridSample等;

- **objdetect类**：目标检测类算子，例如Iou、RoiAlign、RoiAlignRotated等;

https://gitcode.com/cann

CANN

ops-cv仓典型算子功能介绍

分类	算子名	功能介绍
image	GridSample	网格采样，空间变换、图像扭曲、可变形卷积
	ResizeLinear	线性插值缩放
	ResizeUpsampleTrilinear	三线性插值上采样
	UpsampleBicubic2d	双三次插值上采样（2D）
	UpsampleBilinear2d	双线性插值上采样（2D）
	UpsampleLinear1d	线性插值上采样（1D）
	UpsampleNearest	最近邻上采样
	UpsampleNearest3d	最近邻上采样（3D）
	UpsampleNearestExact3d	精确最近邻上采样（3D）
objdetect	IouV2	计算两个矩阵的重叠面积占两个矩阵总面积的比例
	RoiAlignRotated	旋转的感兴趣区域对齐
	StackGroupPoints	将分组后的点云数据重新堆叠和组织，常用于点云采样、特征聚合

ops-cv仓目录结构介绍

```
├── build.sh                  # 项目工程编译脚本
├── cmake                     # 项目工程编译目录
├── CMakeLists.txt
├── common                      # 项目公共头文件和公共源码
├── docs                       # 项目文档介绍
├── examples                     # 使用通用算子开发和调用示例
├── objdetect                 # objdetect类算子
├── image                      # image类算子
│   ├── grid_sample           # GridSample算子所有交付件，如Tiling、Kernel等
│   │   ├── CMakeLists.txt      # 算子编译配置文件
│   │   ├── docs               # 算子说明文档
│   │   ├── examples           # 算子使用示例
│   │   ├── op_graph           # 算子构图相关目录
│   │   ├── op_host            # 算子信息库、Tiling、InferShape相关实现目录
│   │   │   └── op_api         # 算子aclnn接口实现目录
│   │   ├── op_kernel          # 算子kernel目录
│   │   └── README.md          # 算子说明文档
│   ├── ...
│   └── CMakeLists.txt          # 算子编译配置文件
├── tests                       # 测试工程目录
├── README.md
├── requirements.txt          # 本项目需要的第三方依赖包
└── scripts                   # 脚本目录，包含自定义算子、kernel构建相关配置文件
```

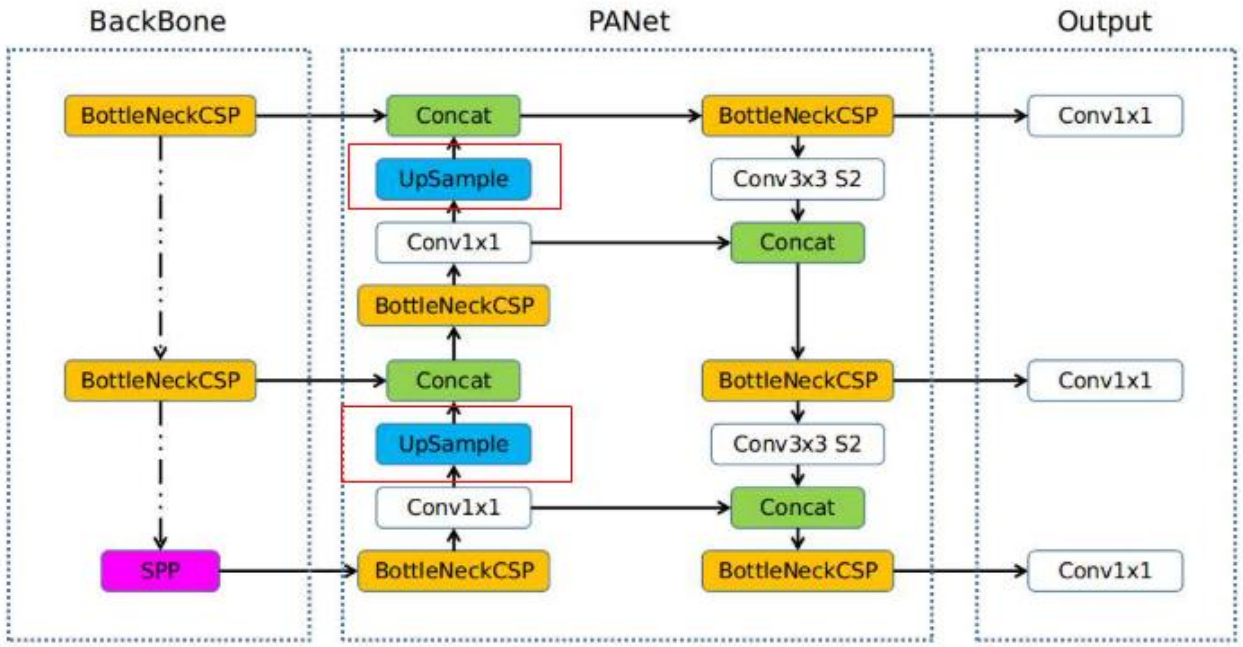
- **Docs**：包含aclnn接口介绍等md文档；
- **Examples**：算子调用示例；
- **op_graph**：算子图模式场景交付件，例如InferDatatype等；
- **op_host**：算子基础host侧交付件，例如Tiling、aclnn接口等；
- **op_kernel**：算子device侧交付件，包含算子核心实现。

https://gitcode.com/cann

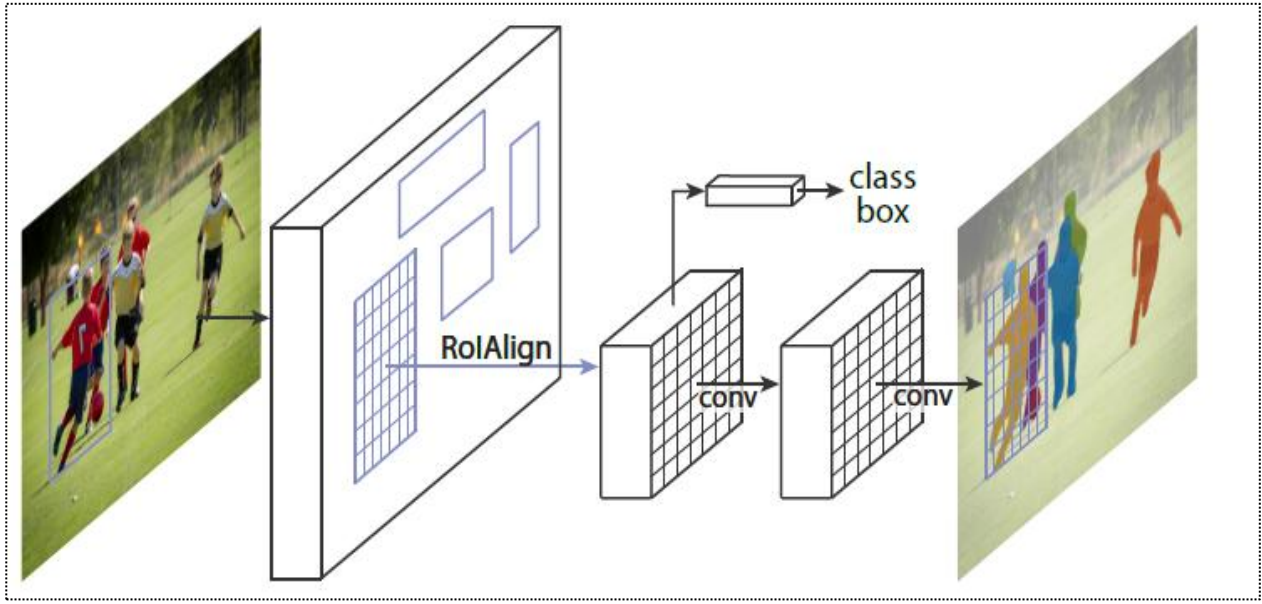
CANN

ops-cv仓算子应用典型场景: 智能驾驶、 视频监控、 医学影像分割

YOLOv5



Mask R-CNN



U-net

.....

https://gitcode.com/cann

CANN

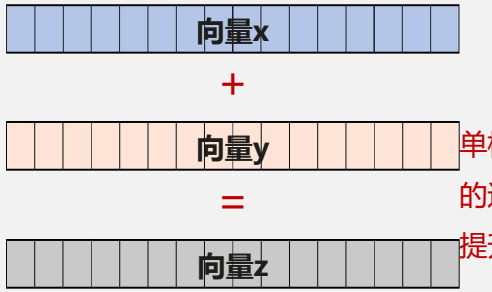
采用NPU进行计算加速的小知识

标量计算单元

标量计算

```
int x=0, y=0;
int z=x + y;
```

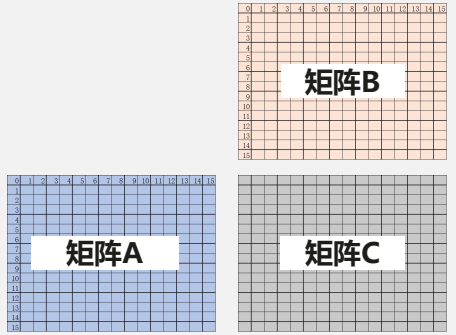
Vector计算单元



向量计算

单核1个cycle可以处理128个fp16数据的运算计算效率的相对于标量有很大的提升

Cube计算单元



矩阵计算

单核1cycle可以处理两个矩阵(多条向量组合)的点乘运算，针对多条向量的点乘运算计算效率比向量计算单元有很大提升

用Scalar 计算矩阵乘

```
for i in range(16):
  for j in range(16):
    acc = 0.0
    for k in range(16):
        acc += x[i, k] * y[k, j]
    z[i, j] = acc
```

用Vector 计算矩阵乘

```
for i in range(16):
  for j in range(16):
    tmp = x[i,:] * y[:, j]
    z[i, j] = sum(tmp)
```

用Cube 计算矩阵乘

```
z = x · y
```


CANN

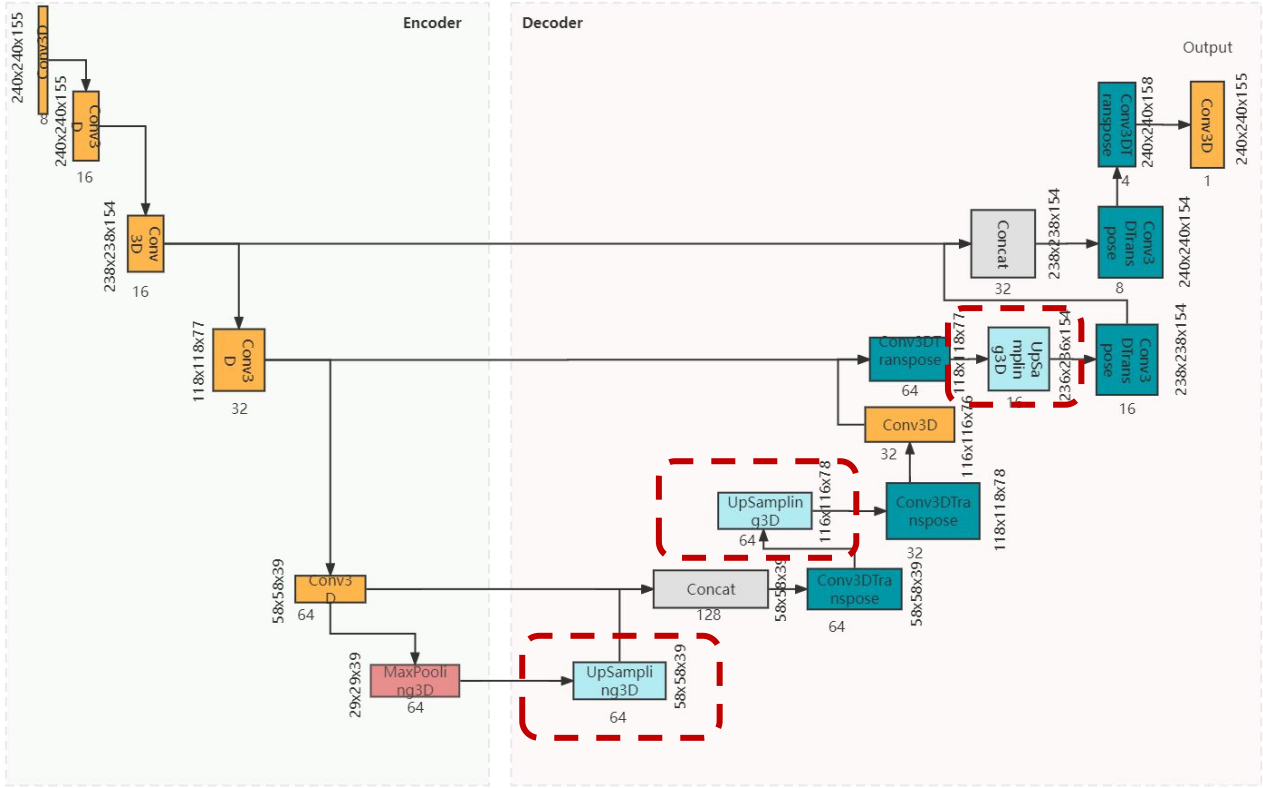
目录

插值技术：让数字世界“无缝衔接”的隐形推手

- 在深度学习的视觉世界中，插值计算在PyTorch、TensorFlow等AI框架中是模型的核心操作之一，用于缩放特征图，将低分辨率特征扩展为高清表达，或反向压缩高维特征以简化计算，支撑着现代AI视觉系统的主要场景

算法类型	核心特点	应用场景（含典型模型 / 任务）
最近邻插值（Nearest）	1. 速度最快，无浮点计算； 2. 易产生 “锯齿感”； 3. 部署成本低。	1. 目标检测 FPN（快速上采样高层特征）； 2. YOLO 轻量版（实时性优先）； 3. 3D 点云粗采样（PointNet 预处理）。
最近邻精确插值（Nearest Exact）	1. 改进版最近邻，避免浮点偏移； 2. 保持像素位置准确性； 3. 速度接近传统最近邻。	1. 轻量化分割模型（MobileNet-UNet，需坐标对齐）； 2. 监控视频实时上采样； 3. 1D 传感器信号调整。
双线性插值（Bilinear）	1. 基于 4 邻域加权，空间连续性好； 2. 计算量中等； 3. 轻微模糊细节。	1. 语义分割（DeepLabV3+、U-Net，医学影像选 True，普通场景选 False）； 2. GAN 生成器（GigaGAN，避免棋盘效应）。
三线性插值（Trilinear）	1. 双线性的 3D 扩展，基于 8 邻域加权； 2. 无体素 “断层”； 3. 适配 3D 硬件加速。	1. 医学影像 3D 分割（3D U-Net，需精准病灶定位选 True）； 2. 3D 目标检测（VoxelNet 体素特征上采样）。
双三次插值（Bicubic）	1. 基于 16 邻域三次多项式加权； 2. 细节保留优于双线性； 3. 计算量较大。	1. 超分辨率（ESPCN 变体、EDSR 消融实验，8× 放大选 True）； 2. Stable Diffusion 扩展（Ultimate SD Upscale，纹理细化）。

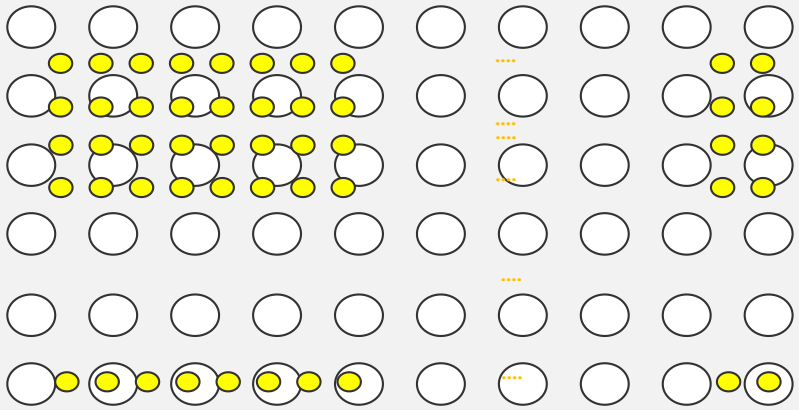
3D U-Net：三线性插值，恢复 3D 特征图的分辨率



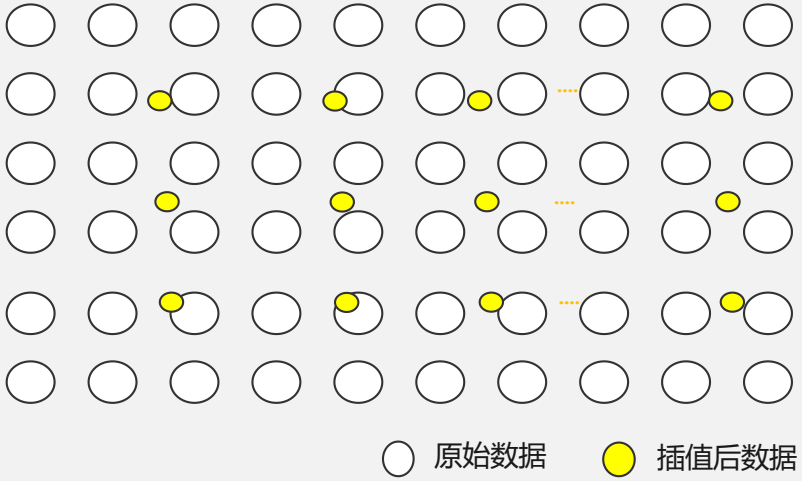
CANN

插值算子：常用AI框架有关空间变换的核心操作

上采样（放大），输出Tensor的点变多



下采样（缩小），输出Tensor的点变少



- PyTorch 的采样算子主要通过层接口（nn.Upsample） 和函数式接口（F.interpolate） 实现，两者功能一致，仅使用方式不同

torch.nn.Upsample(size, scale_factor, mode, align_corners, recompute_scale_factor)

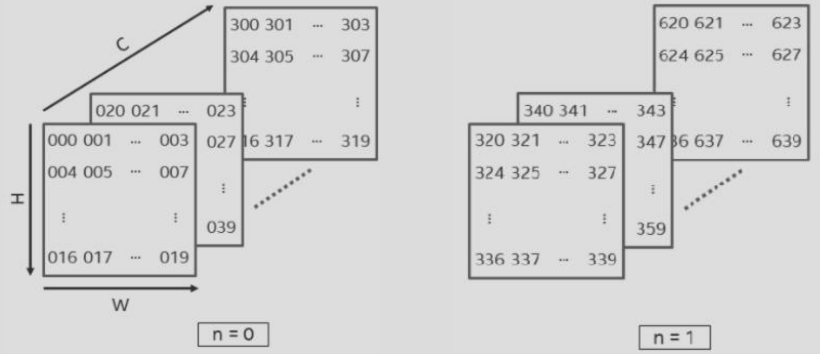
- size：目标输出尺寸
- scale_factor：缩放因子（如2.0表示放大 2 倍）；
- mode：插值模式（nearest/bilinear/bicubic/trilinear等）
- align_corners：是否对齐边角像素

CANN

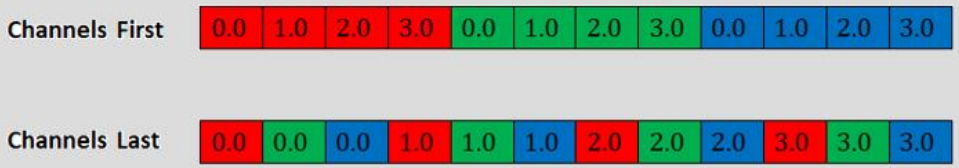
插值算子张量的典型结构

格式	维度	结构说明
NCHW	(Batch, Channel, Height, Width)	1. 通道优先：同一通道的像素连续存储（如 RGB 图像先存完所有 R 通道像素，再存 G、B）； 2. PyTorch 默认格式，逻辑维度与物理存储一致
NCL	(Batch, Channel, Length)	1. 1D 时序数据：通道维度独立，适合处理单维信号（如音频、传感器数据）； 2. 逻辑上类似 NCHW 的简化版，物理存储为连续内存块。
NCDHW	(Batch, Channel, Depth, Height, Width)	1. 3D 体数据：通道维度优先，适合处理体积数据（如 CT/MRI 影像、3D 视频）； 2. 物理存储按 D→H→W 顺序展开，通道连续存储。
Channel Last（例如：NHWC ）	(Batch, Height, Width, Channel)	1. 硬件友好型存储：通道作为最后维度，提升 CPU 缓存利用率和向量运算效率； 2. PyTorch 中通过torch.channels_last启用，需手动转换输入格式。

NCHW格式（例：N=2，C=8，H=5，W=4）



Channel在前 vs Channel在后

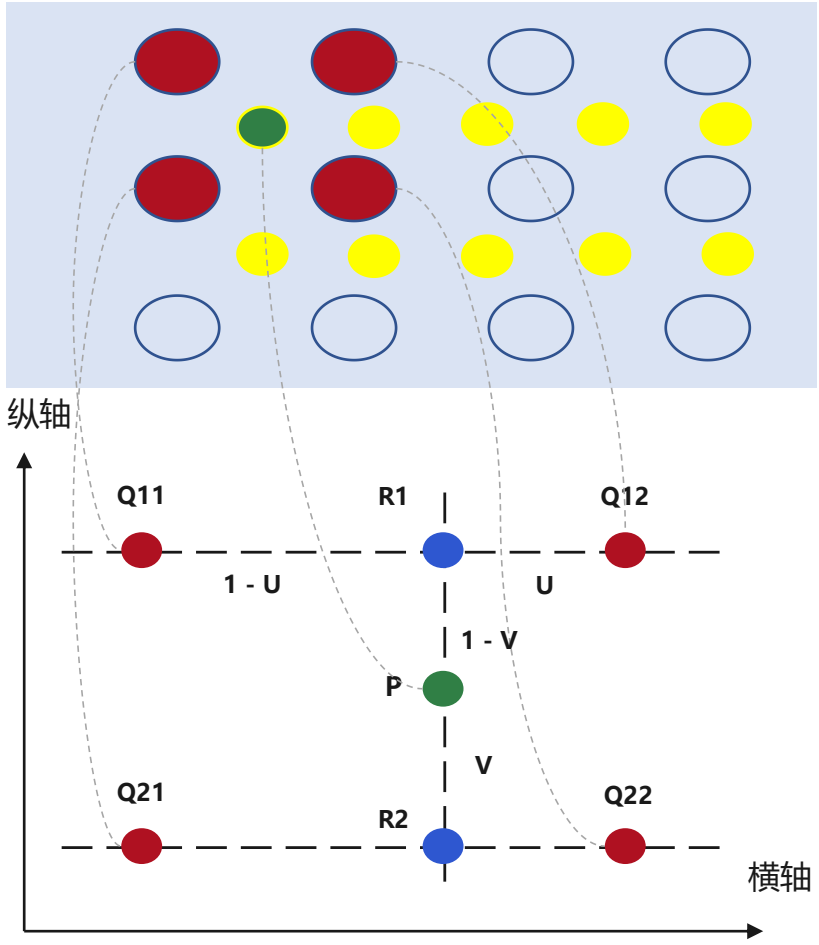


- Channel在后实际上是将相对位置相同但属于不同channel的数据连续存放

CANN

插值算子的计算原理

插值的计算方法，以Bilinear上采样为例



本质：依据与周围4个点的距离加权求和

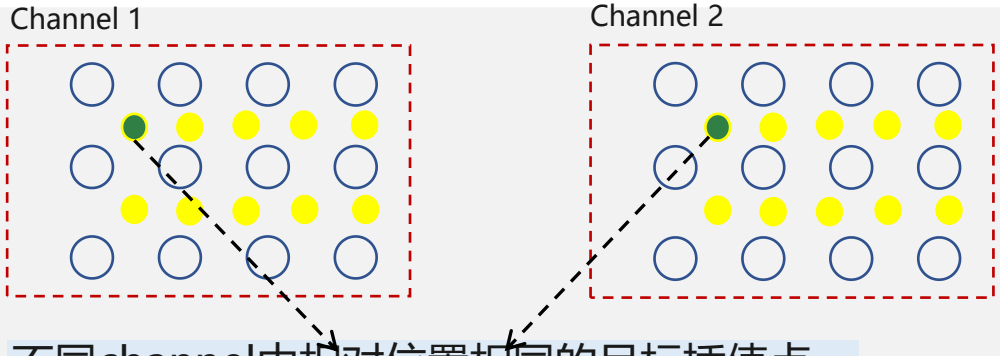
先计算水平方向

$$R1 = (1-u)\times Q11+u\times Q12$$
$$R2 = (1-u)\times Q21+u\times Q22$$

再计算垂直方向

$$P = (1-v)\times R1 + v\times R2$$
$$= (1-u)(1-v)\times Q11 + u(1-v)\times Q12 + (1-u)v\times Q21 + uv\times Q22$$

权重1 **权重2** **权重3** **权重4**



不同channel中相对位置相同的目标插值点，权重相同

CANN

插值算子的计算方法

四重循环逐点计算

```
# 伪代码,传统实现: 四重循环,逐点计算
for n in range(N):                          # 批次循环
  for c in range(C):                        # 通道循环 瓶颈
    for h_out in range(H_out):
      for w_out in range(W_out):
        # 计算权重和邻域坐标
        w00, w01, w10, w11 = compute_weights(h_out, w_out)
        y0, x0, y1, x1 = get_neighbor_indices(h_out, w_out)
        # 获取四个邻域点值
        v00 = input[n, c, y0, x0]
        v01 = input[n, c, y0, x1]
        v10 = input[n, c, y1, x0]
        v11 = input[n, c, y1, x1]
        #加权求和
        output(n,c, h_out, w_out) = w00*v00 + W01*v01 +
w10*v10 + w11*v11
```

向量化计算

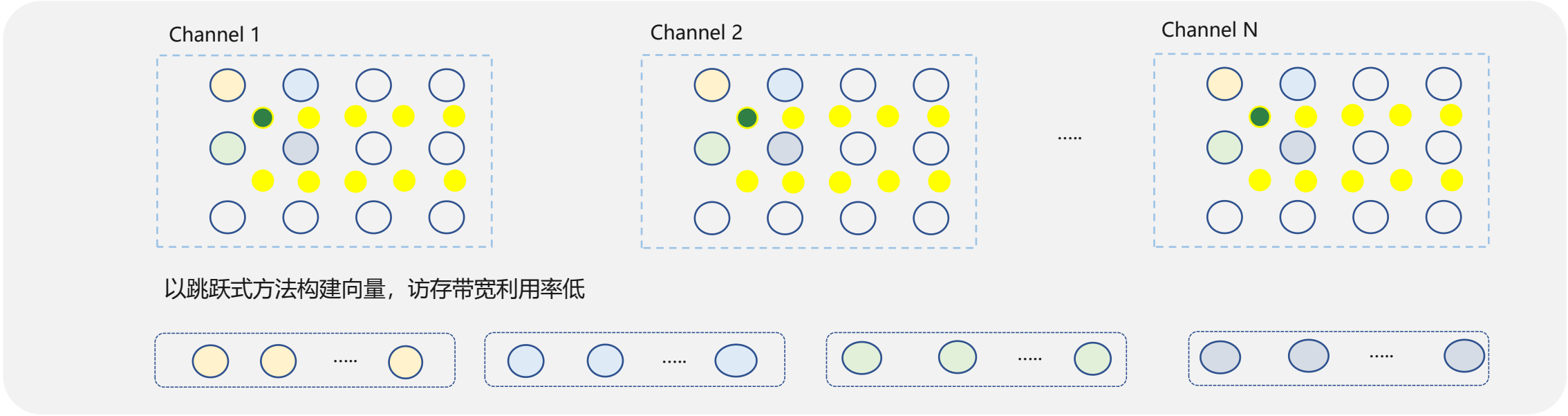
```
# 伪代码,向量化实现:空间位置外循环,批次通道内聚
for h_out in range(H_out):
  for w_out in range(W_out):
    # 1.计算权重和邻域坐标(所有NC共享)
    w00, w01, w10, w11 = compute_weights(h_out, w_out)
    y0, x0, y1, x1 = get_neighbor_indices(h_out, w_out)
    # 2.向量化加载所有通道和批次的邻域数据,[N,C]维度被展平为向量。
    v00_vec = input[:, :, y0, x0] # 形状[N × C]
    v01_vec = input[:, :, y0, x1] # 形状[N × C]
    v10_vec = input[:, :, y1, x0] # 形状[N × C]
    v11_vec = input[:, :, y1, x1] # 形状[N × C]
    # 3.向量化加权计算(单指令处理所有NC)
    output_vec = (w00 * v00_vec + w01 * v01_vec + w10 *
v10_vec + w11 * v11_vec)
    # 4.向量化存储结果
    output[:, :, h_out, w_out] = output_vec
```

通过向量化计算重组，对于每个输出位置，一次性加载所有批次和通道的邻域数据，即形成长度为[N × C]向量，利用现代处理器的SIMD指令并行完成加权求和，这种优化带来额外收益。向量化计算消除传统实现中大量的循环开销，计算效率至少提升4倍以上，特别在通道数大的场景（如ResNet的2048通道）效果更为显著。

CANN

向量化计算方法存在访存瓶颈

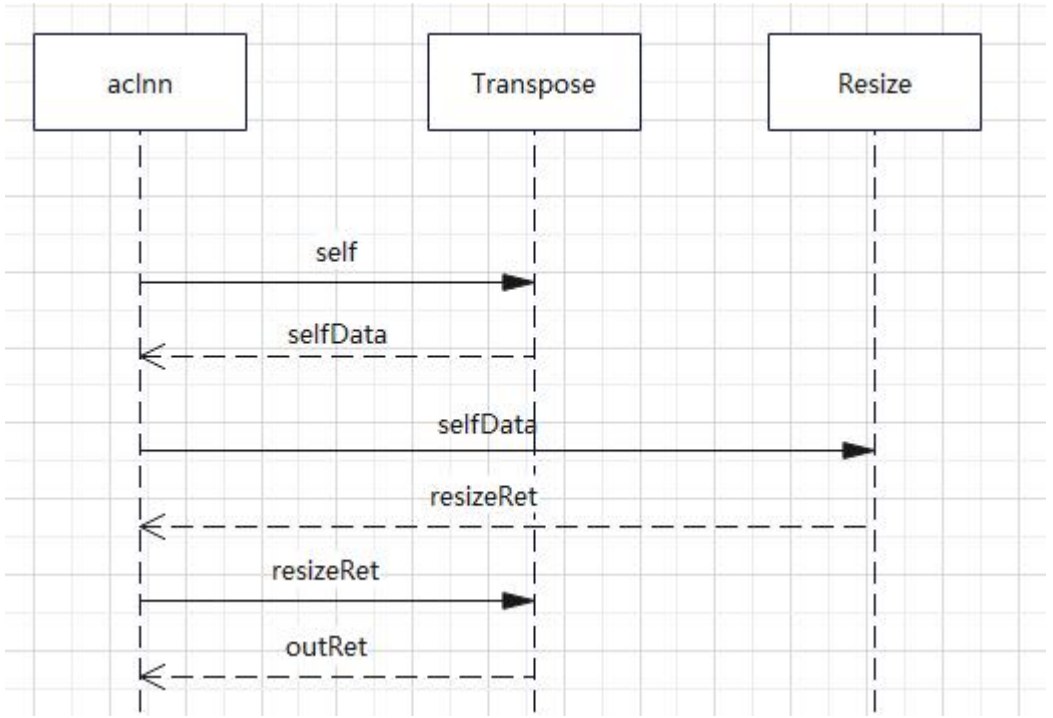
- 插值算法向量化计算后能明显提升性能，但也存在访存问题
  - 当构建跨通道/批次的向量时（如input[:, :, y0, x0]），需要从全局内存中收集非连续存储的数据点，每个通道的数据间隔的字节数为H * W * sizeof(data type)。这种“跳跃式”访问模式导致硬件加速卡的访存带宽利用率急剧下降



CANN

预先转置更改数据排布，缓解部分访存瓶颈

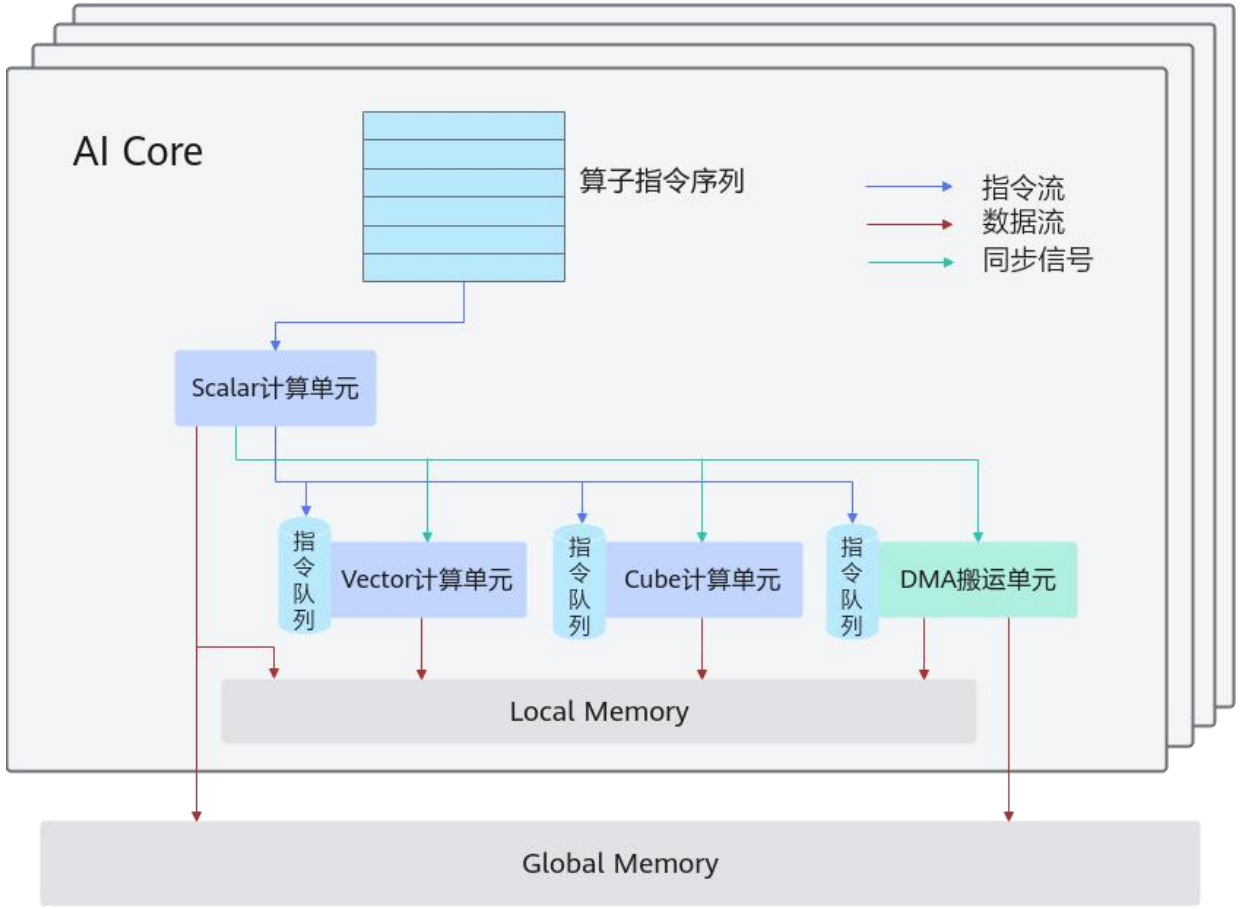

```
// 伪代码，转置调用算子
auto selfData = l0op::TransDataSpecial(self,
Format::FORMAT_HWCN, 0, uniqueExecutor.get());
CHECK_RET(selfData != nullptr, ACLNN_ERR_INNER_NULLPTR);

// 执行L0算子
auto resizeRet = l0op::ResizeBilinearV2(selfData, sizes,
false, outData, uniqueExecutor.get());
CHECK_RET(resizeRet != nullptr, ACLNN_ERR_INNER_NULLPTR);

// 输出转NCHW
auto outRet = l0op::TransData(resizeRet, self-
>GetStorageFormat(), 0, uniqueExecutor.get());
CHECK_RET(outRet != nullptr, ACLNN_ERR_INNER_NULLPTR);
```



Aclnn调用算子前后添加转置算子，将算子转置为HWNC格式，算子使用HWNC格式进行算子计算。避免“跳跃式”取点，提升访存利用率，提升性能。
但是同时多了两次转置运算和MTE2、MTE3搬入搬出操作，总体带宽还是被浪费。

CANN

NPU算力特点回顾



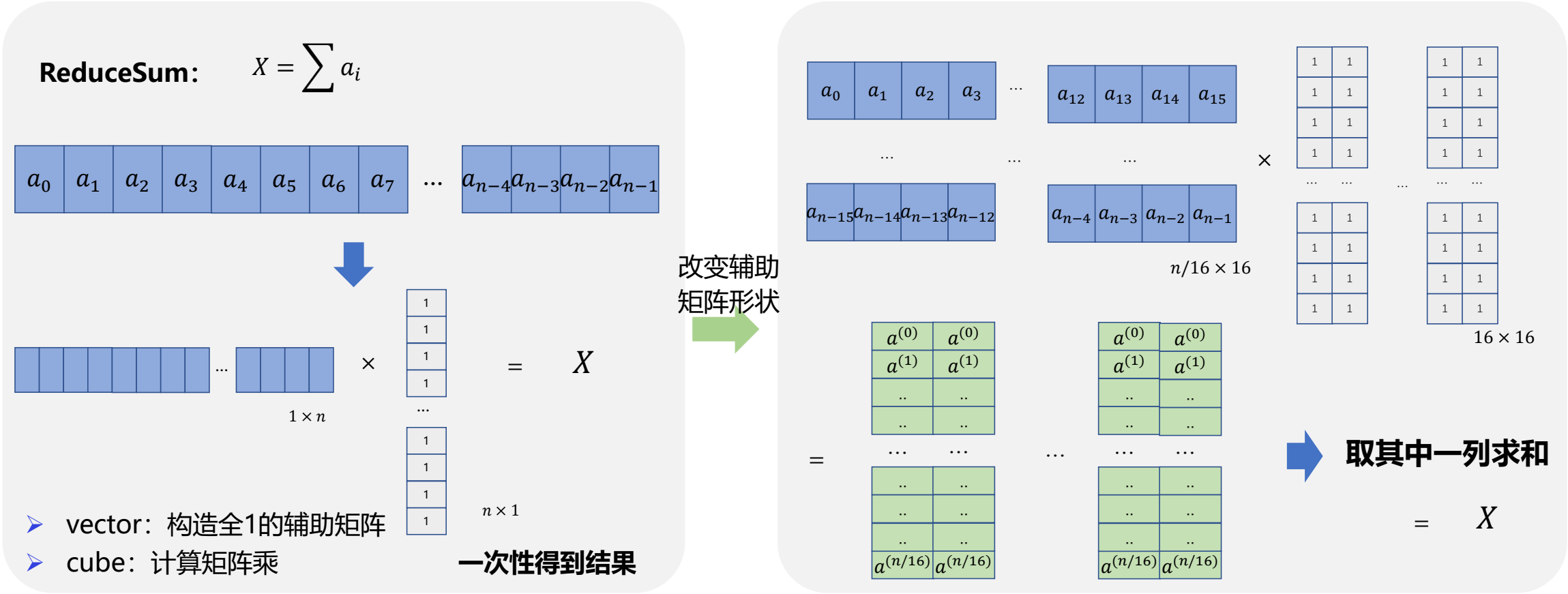
- **Scalar Unit**：负责标量运算；负责程序的流程控制；负责CUBE/Vector等指令的分发；功能上可以看做一个小CPU
- **Vector Unit**：向量运算单元，覆盖各种基本的计算类型，一个节拍完成16*16Byte运算
- **Cube Unit**： 算力巨大，负责矩阵乘计算，一节拍可完成fp16 16*16与16*16的矩阵乘；

  理论上Cube算力是Vector的16倍

CANN

用Cube代替Vector执行计算

- 例如：ReduceSum可看做是权重为1的**加权求和，符合矩阵乘范式**，可使能Cube算力



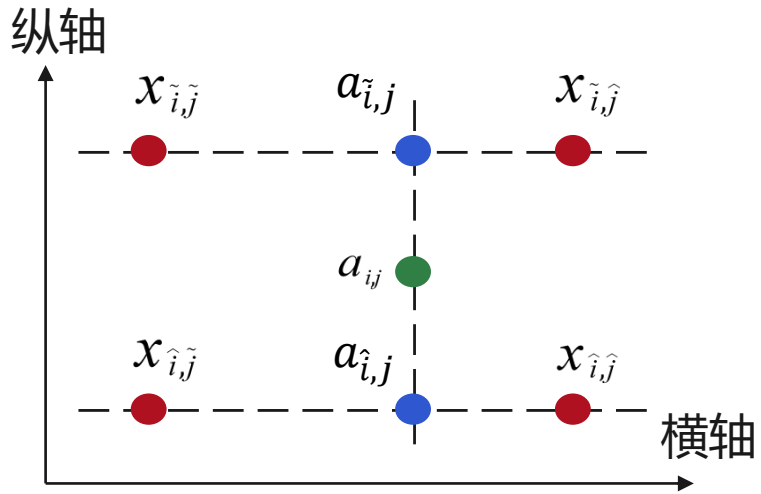
- vector：构造全1的辅助矩阵
- cube：计算矩阵乘

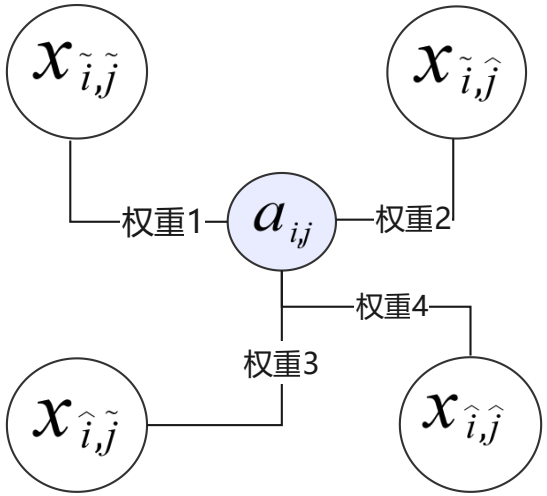
一次性得到结果

- 全1辅助矩阵在算子执行开始阶段一次性构建；vector需处理数据格式排布变换

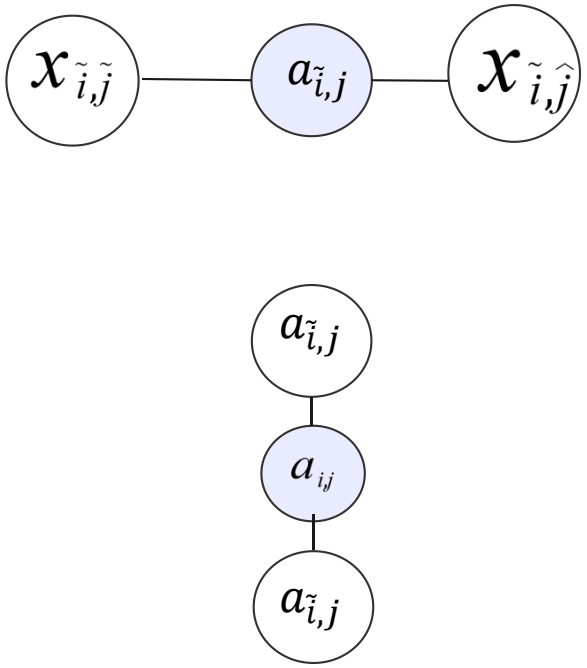
CANN

插值算子具备矩阵化计算的潜质（1）





等价

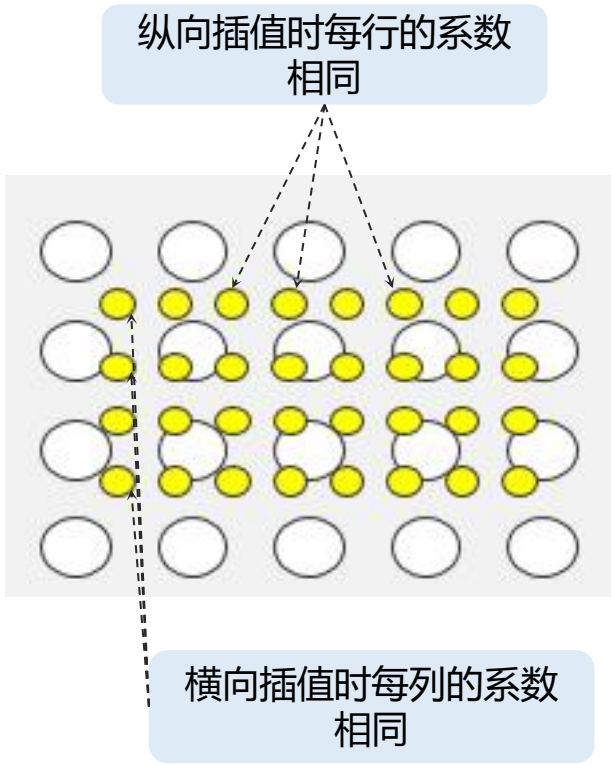


先横向插值

再纵向插值

- 分解为横向插值和纵向插值两个步骤，每个步骤都是加权求和

插值算子矩阵化计算改造原理（2）

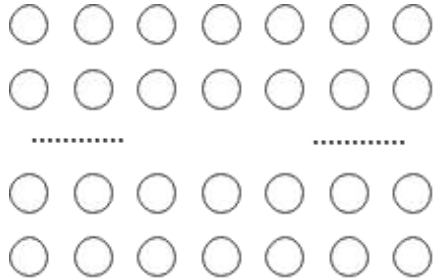


纵向插值

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & \cdots & 0 \\ 0 & w_{1,1} & w_{1,2} & \cdots & 0 \\ 0 & w_{2,1} & w_{2,2} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & w_{k,k} \\ 0 & 0 & 0 & \cdots & w_{k+1,k} \end{bmatrix}$$

权重矩阵示意

原始数据



横向插值

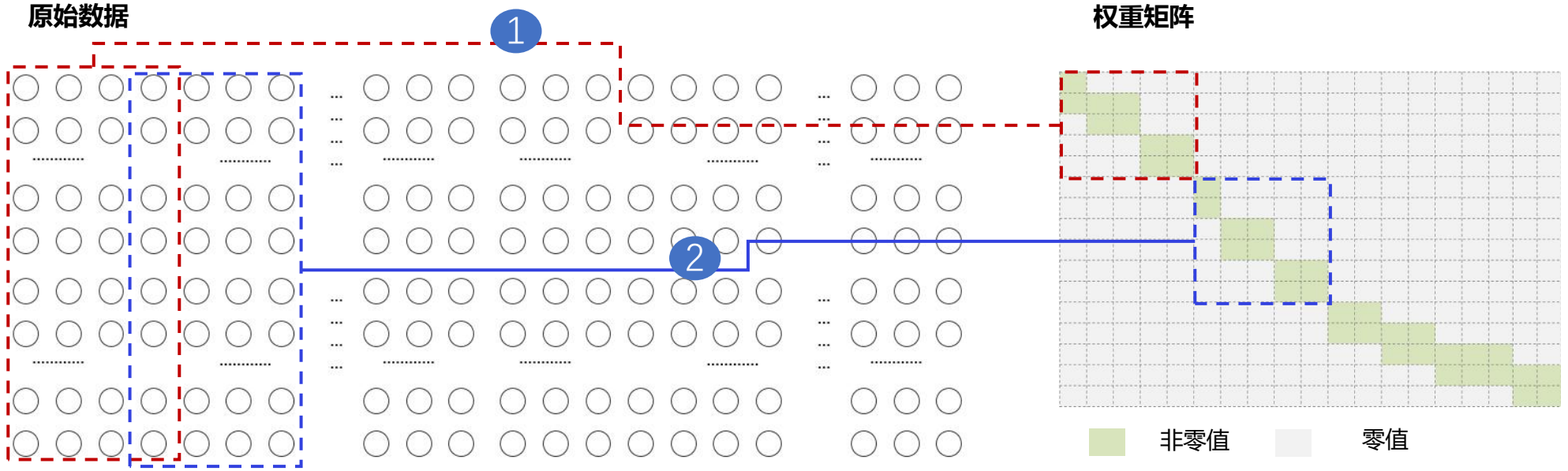
$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & 0 \\ w_{1,0} & w_{1,1} & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & w_{k,k} \\ 0 & 0 & \cdots & w_{k+1,k} \end{bmatrix}$$

- 上采样（放大）场景：权重矩阵较稀疏
- 下采样（缩小）场景：参与加权求和的原始数据更多；相应的，权重矩阵更加稠密
- Bicubic等插值方式参与加权求和的点更多，稀疏程度低

CANN

插值算子矩阵化计算改造原理（3）

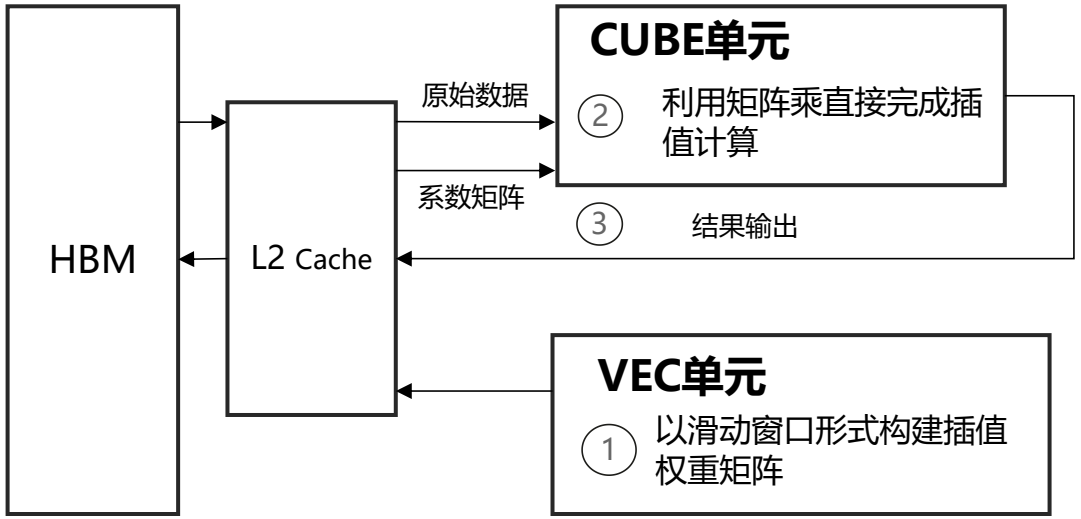
- 插值计算的矩阵乘算法改造后，由于权重矩阵过于稀疏，矩阵乘必须考虑数据特点



- 假定原始数据$(H, W) = (100,120)$，横向放大5倍到$(100, 600)$；权重矩阵$shape = (120,600)$
- 权重矩阵大量数据为0，过于稀疏；全0值部分没有必要计算
- 沿着对角线方向分段局部计算权重矩阵，在与原始数据执行矩阵乘，避免算力浪费

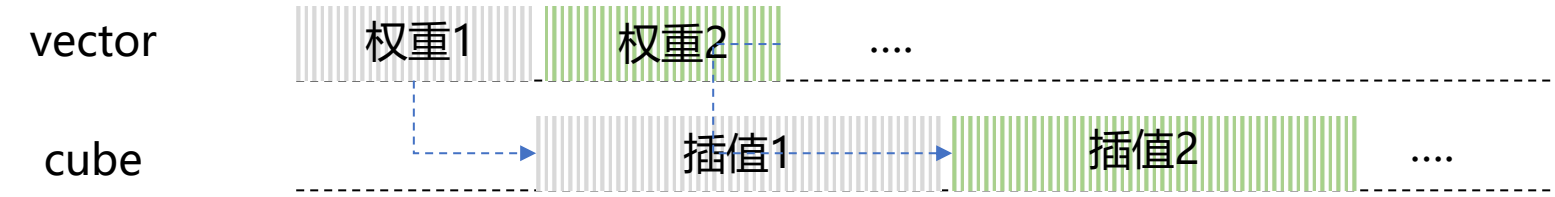
CANN

插值算子矩阵化计算改造原理（4）



- 向量单元：负责系数矩阵构建
- 矩阵单元：负责插值计算
- 矩阵运算和向量运算并行

期望流水

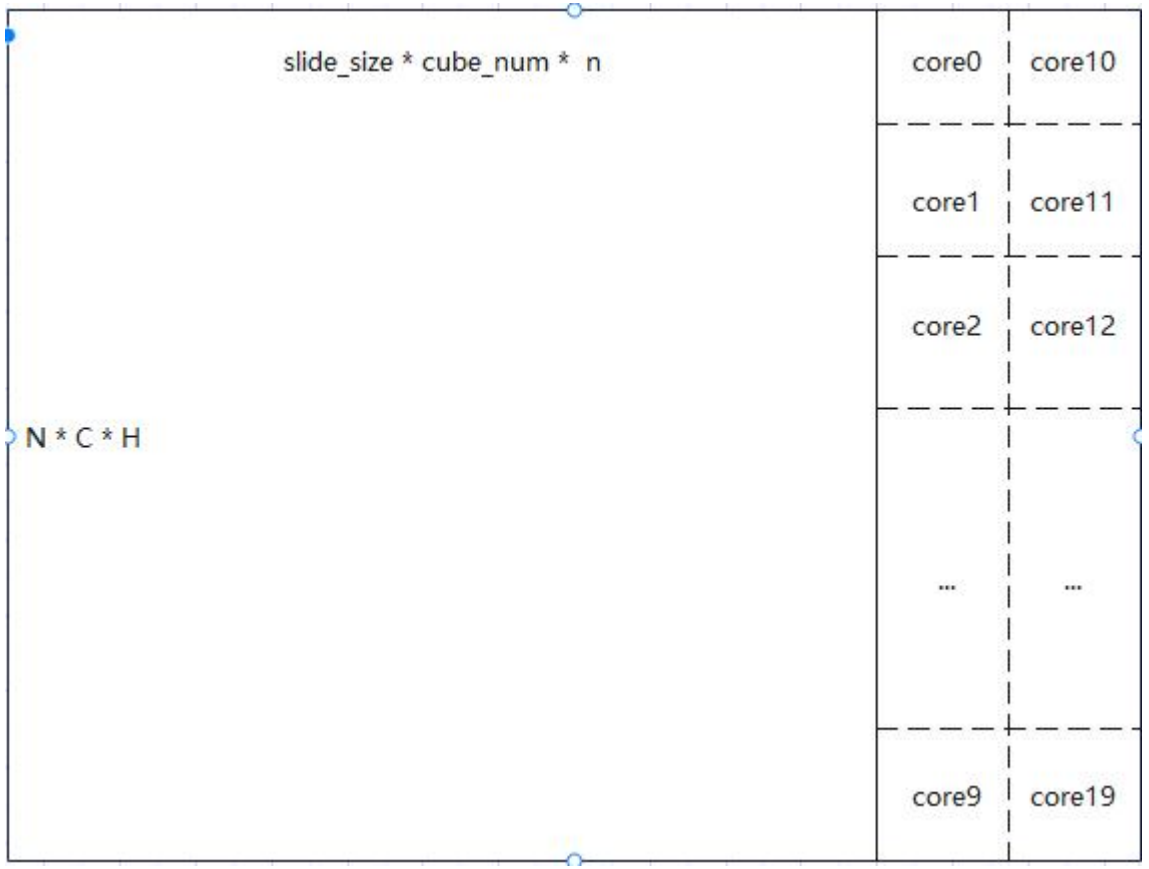


CANN

插值算子矩阵化实现（1）--Tiling伪码


```
// 伪代码，Tiling逻辑
// 获取平台信息、输入tensor、属性信息
auto platformInfo = GetPlatformInfo();
auto tensorsInfo = GetTensor();
auto attrsInfo = GetAttrs();
// 计算Scale
float scale = CalcScale(tensorsInfo, attrsInfo);
// 计算滑窗大小
int64_t slide_size = CalcSlideSize(platformInfo,
tensorsInfo, attrsInfo);
// 计算workspace空间
getWorkSpace(attrsInfo, scale);
// 其他tilingData计算
......
// 填充tilingData
FillTilingData();
```

分核策略（横向扩展为例：）



CANN

插值算子矩阵化实现（2）--Kernel伪码

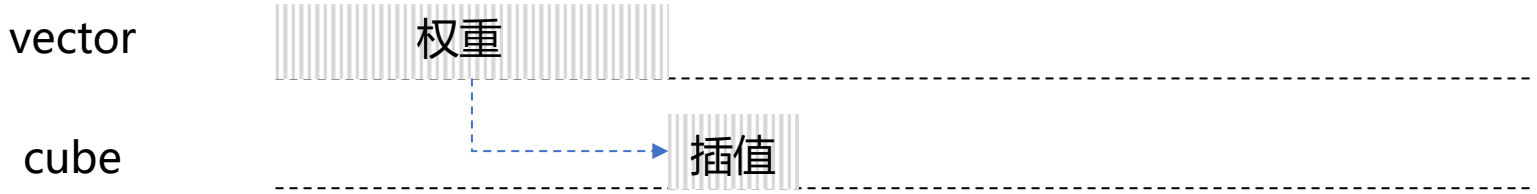

```
// 伪代码，kernel逻辑
void Init() {
    ......  // 初始化Tensor
    getSlideRange(tilingData); // 获取每个核计算的分块位置
}

void Process() {
    WDirectionExpansion(); // 横向扩展
    SyncAll();
    HDirectionExpansion(); // 纵向扩展
}

void WDirectionExpansion() {
    ......
    for (int64_t i = slideStartW; index < slideEnd_w; index
+= slide_size) {
        // 计算系数矩阵
        CalculateRadioTensorW();
        // 拷贝系数矩阵到workspace
        CopyRadioTensorToGM();
        // 矩阵计算
        CalculateMatmulW();
    }

    // 尾块处理
    ......
}
```

```
void CalculateRadioTensorW() {
    radioTensor[x][slideSize];
    Duplicate(radioTensor, 0);
    for (int64_t i = 0; i < size; i++) {
        if align_corners
            index = i * scale
        else
            index = max(0,scale*(i+0.5)-0.5)
        index0 = floor(index)
        index1 = min(index0 + 1, input_size - 1)
        lamda0 = index - index0
        lamda1 = 1 - lamda0
        lamda[index0][i] = lamda1;
        lamda[index1][i] = lamda0;
    }
}

void CalculateMatmulW() {
    ......
    // 重新设置矩阵形状
    matmulW.SetSingleShape(singCoreM, singleCoreN,
singleCoreK);
    // 调用Matmul高阶接口计算
    Matmul(inTensor, radioTensor, intermediateTensorGm);
    ......
}
```

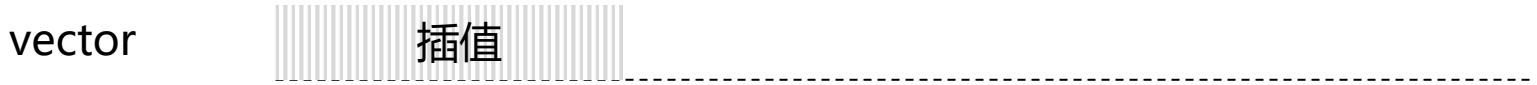
问题1：极小shape场景性能

- Cube单元虽然算力大，但是不是所有场景都有正向收益。向量单元应用得当也能获得很好的性能。
- 极小shape场景下，矩阵化运算会存在cube算力利用率低场景，构造权重矩阵耗时显得过长，同等时间下向量化运算已经完成，此时矩阵化运算是负收益。

矩阵化运算流水：



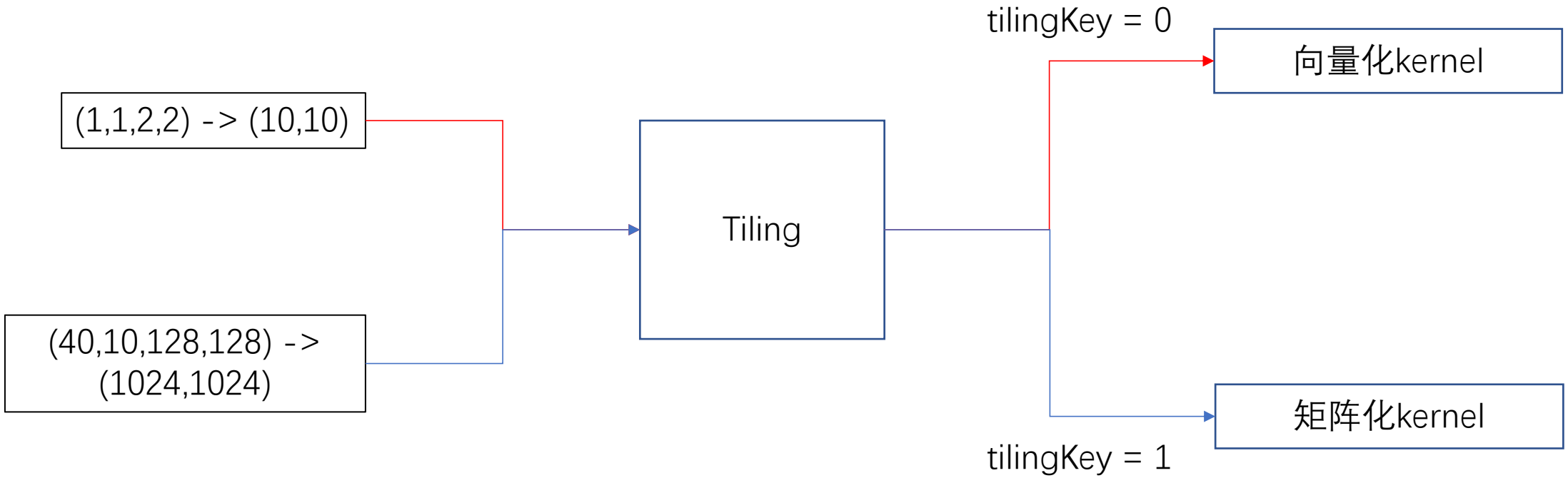
向量化化运算流水：



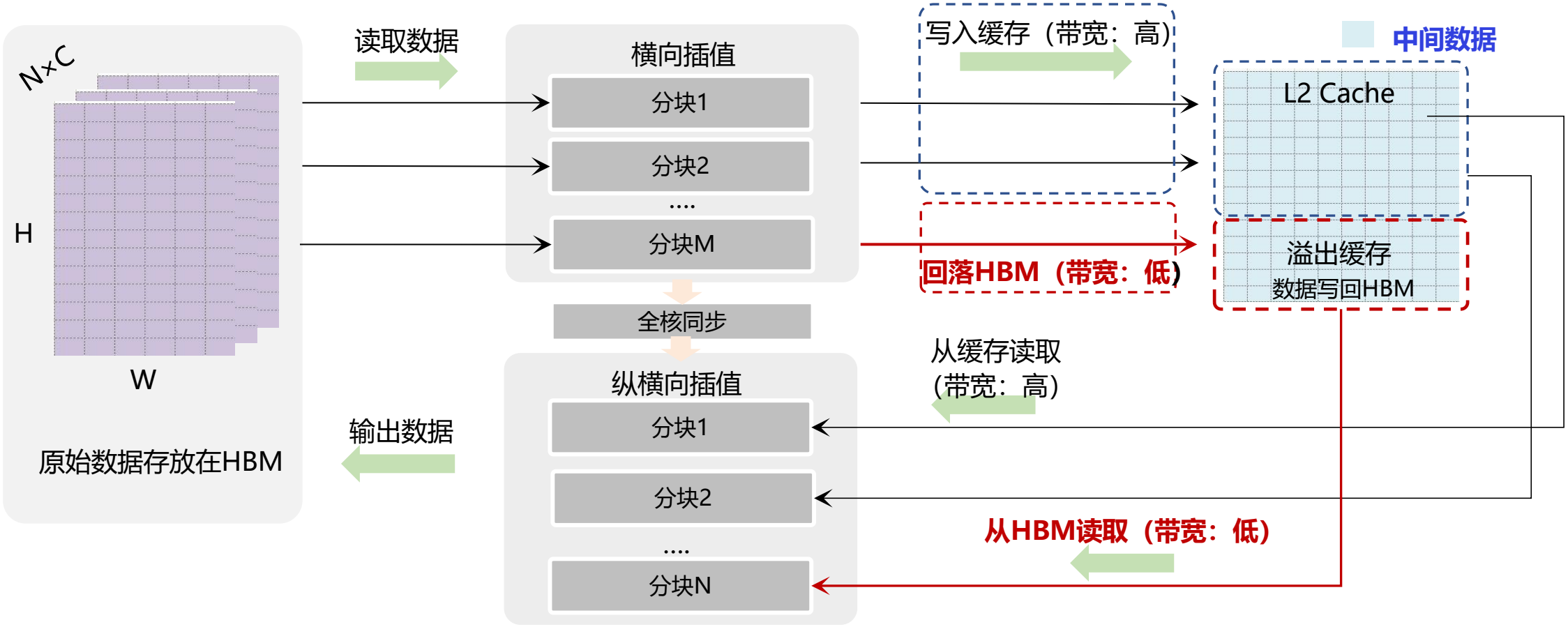
CANN

问题1：极小shape场景性能

- 设置多个kernel模板，tiling中根据输入输出shape设置不同tilingKey，分别走向量化运算和矩阵化运算



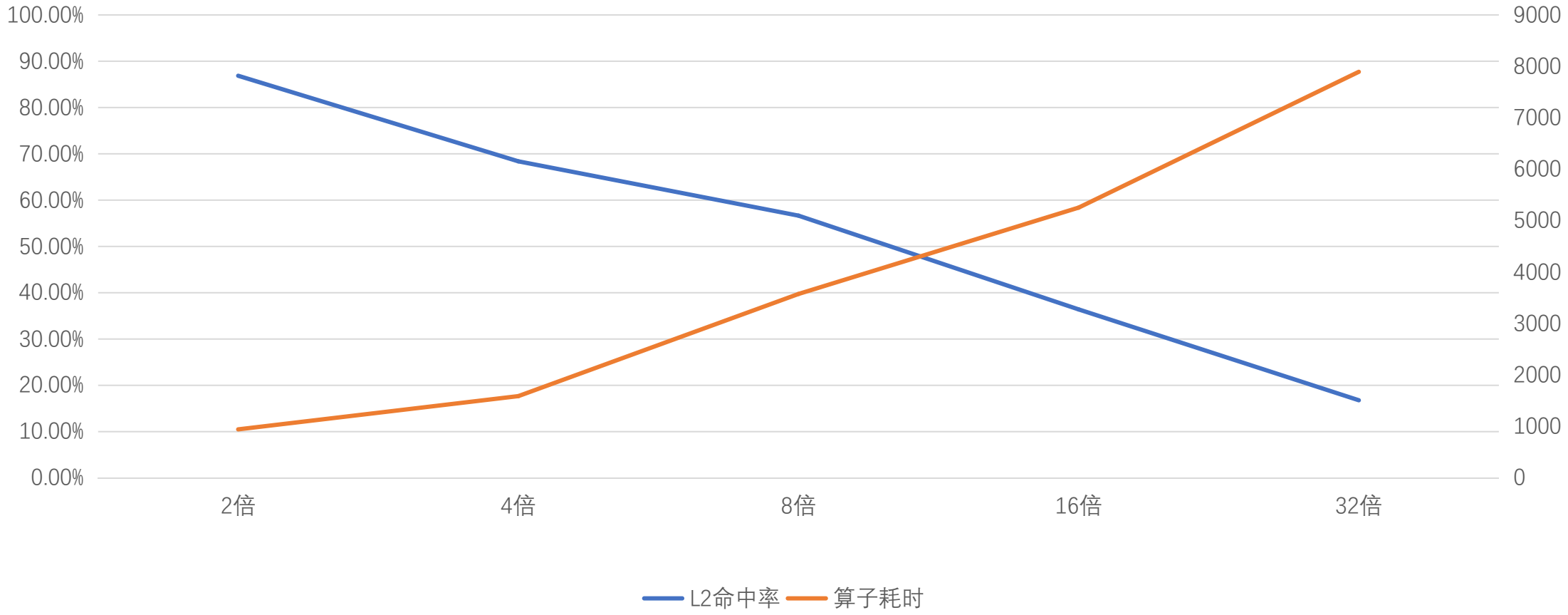
问题2：L2 Cache“击穿”，性能下降（1）



- 先横向再纵向：数据量较小时，中间数据可全写入L2 Cache，充分利用高带宽；数据量变大后，L2 Cache击穿，触发数据回落HBM，可用带宽变小，影响整体性能

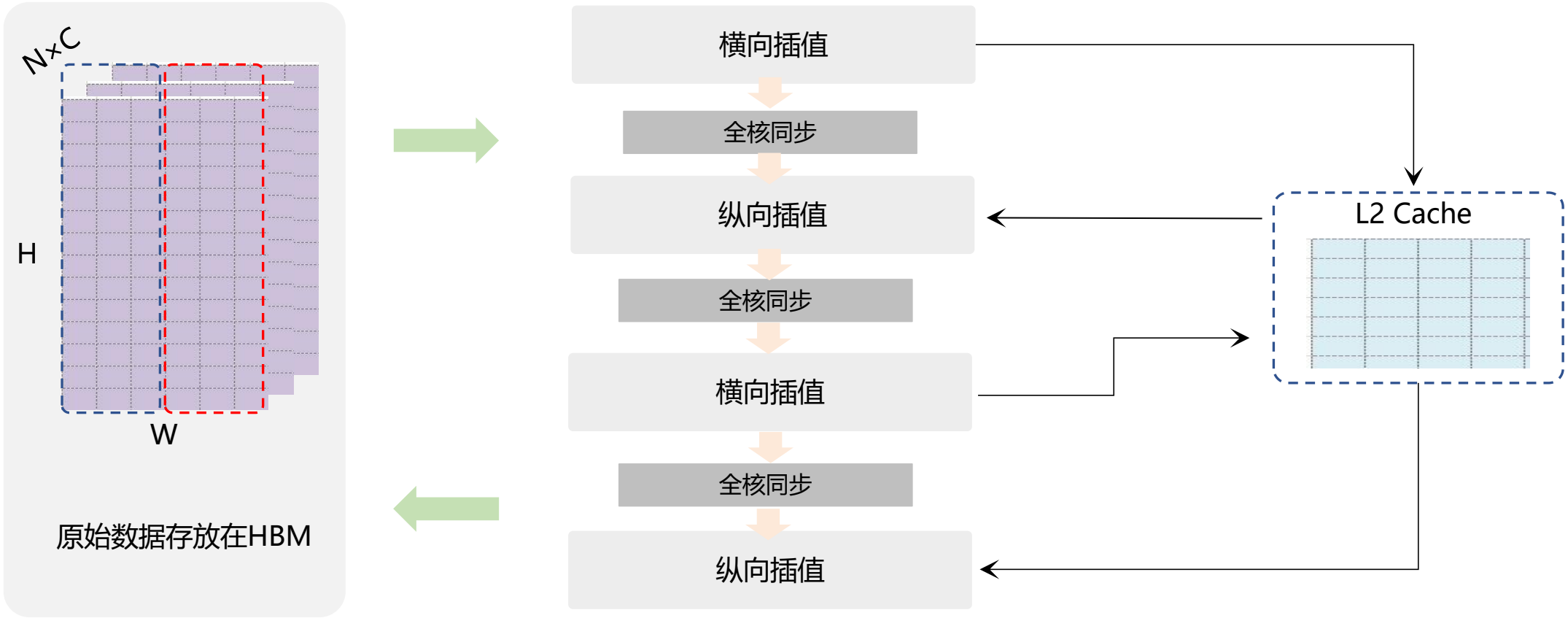
CANN

L2 Cache“击穿”，性能下降（2）



测试使用shape为（40, 1, 500, 500） -> （40, 1, 1000, 1000）-> (40, 1, 1000, 2000）-> ...

L2 Cache优化思路



- 结合L2 Cache容量 + 高带宽特点，Tiling精细规划数据分块策略，充分利用高带宽，提升性能

CANN

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