Graph-autofusion开源

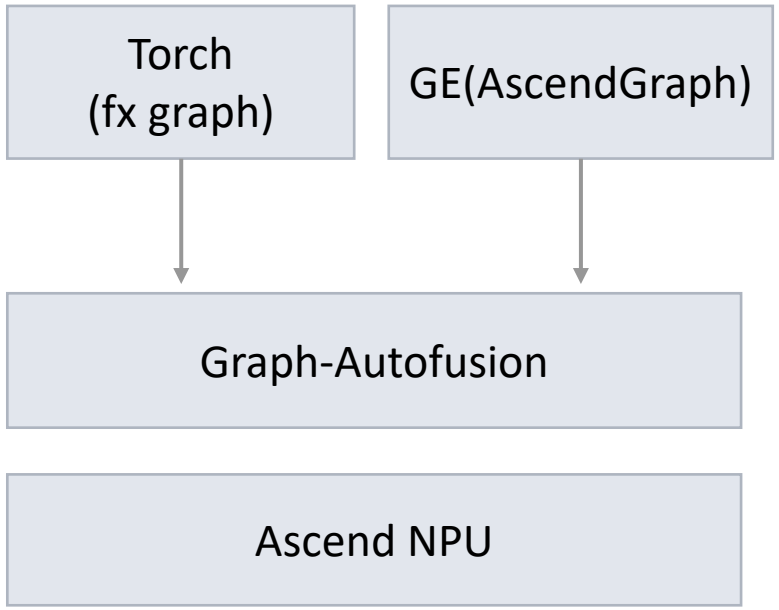
--暨SuperKernel原理介绍与优秀实践

https://gitcode.com/cann/graph-autofusion

CANN

目录

Graph-Autofusion仓的定位



Graph-Autofusion 仓背景

- 面向昇腾芯片，提供自动融合技术
- 从GE中，将SuperKernel、自动融合codegen等融合能力抽取到Graph-Autofusion 中

技术特点

- **生态对接**优先：支持 fx graph
- **轻量级**：单特性代码量数K到数十K
- **依赖少**：仅依赖 Ascend C编程框架、Runtime 等基础库
- **泛化性好**：基于JIT技术，在线codegen+编译

CANN

Graph-Autofusion仓路标

2025.10.31
- SuperKernel 特性开源

2025.11.30
- 启动 inductor backend
- 启动 SuperKernel 与 Ascend C 解耦工作，目标独立升级

2025.12.31
- SuperKernel 与 Ascend C 可独立升级，SuperKernel 演进不再依赖升级 Ascend C

2026
- SuperKernel、inductor 后端持续演进

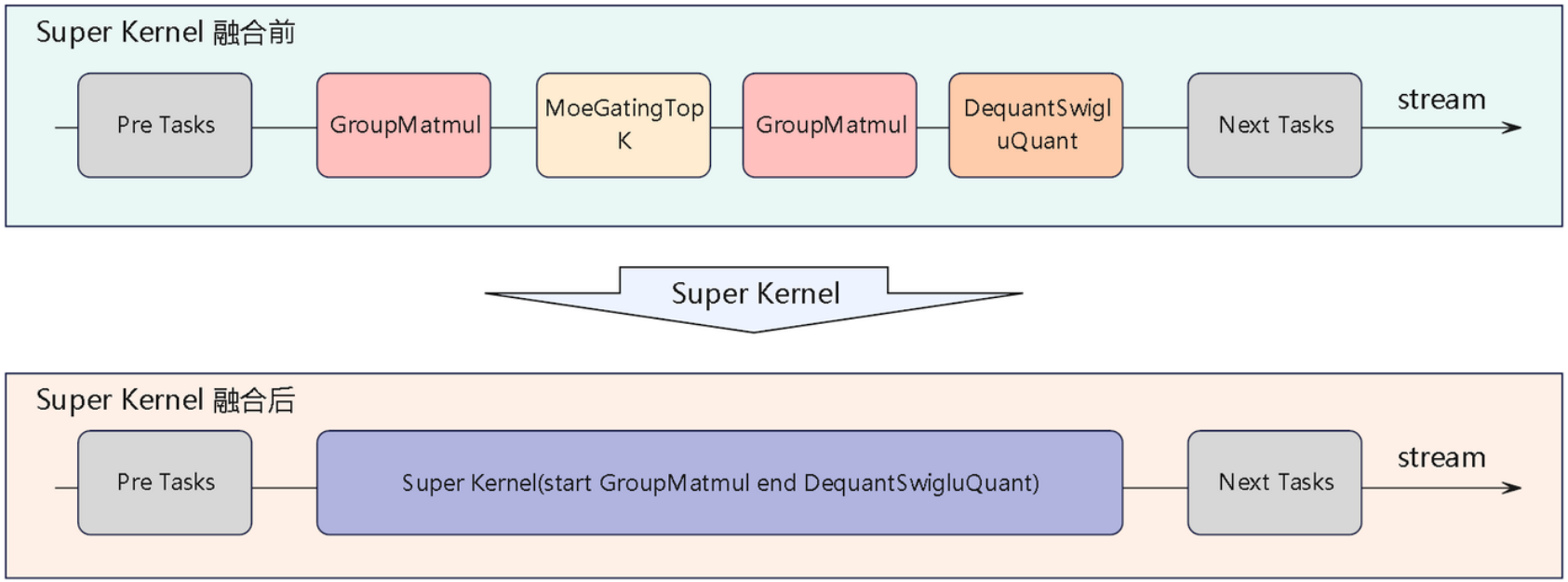
目录

Super Kernel技术原理

背景：

2025年初，随着DeepSeekV3/R1网络模型的开源，大模型推理迈入了一个全新阶段。该模型基于MoE（Mixture of Experts）混合专家结构，实现了参数的稀疏激活。如何在低计算量的前提下，进一步提升模型性能成为关键。

为此，我们引入了一种名为SuperKernel的执行优化方法。该方法将整个网络模型重新编译为一个大算子，减少硬件调度开销。



CANN

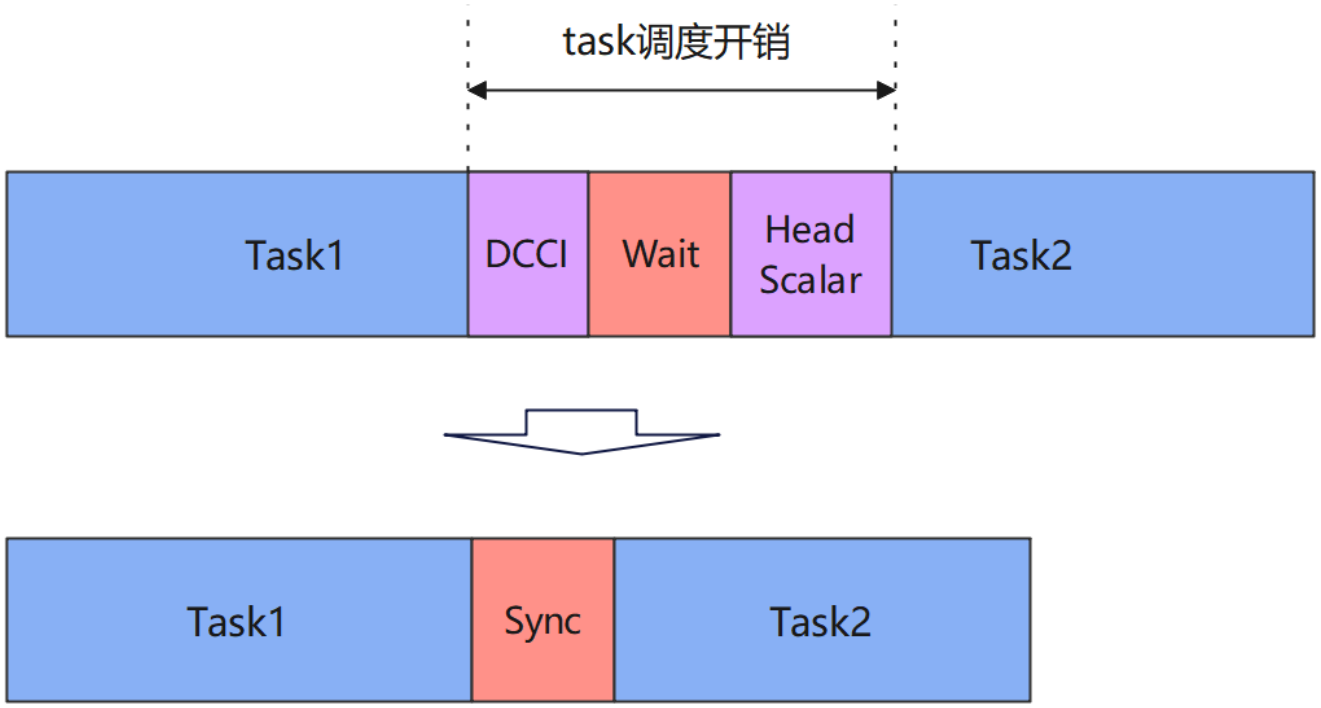
SuperKernel技术原理

原理介绍：

SuperKernel 是一种面向网络图模型的调度优化技术。其核心思想是：基于网络图模型中算子的先验信息（如算子类型、前后序依赖关系等），结合即时编译（JIT）能力，将整个网络模型重新编译为单一算子，从而显著降低算子调度开销。

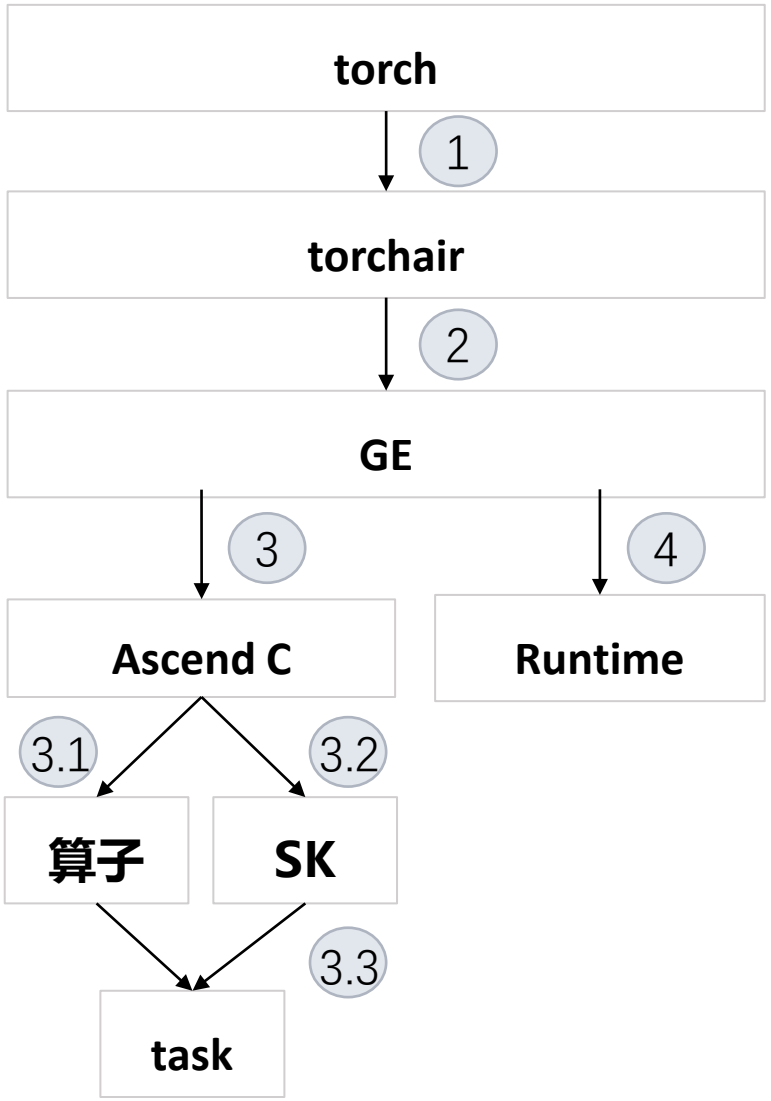
以DeepSeekV3网络为例， DeepSeekV3一共有60+层的网络，每一层有20+算子，一次的调度开销约为1-3us，整网一次推理调度开销约为2-3ms。

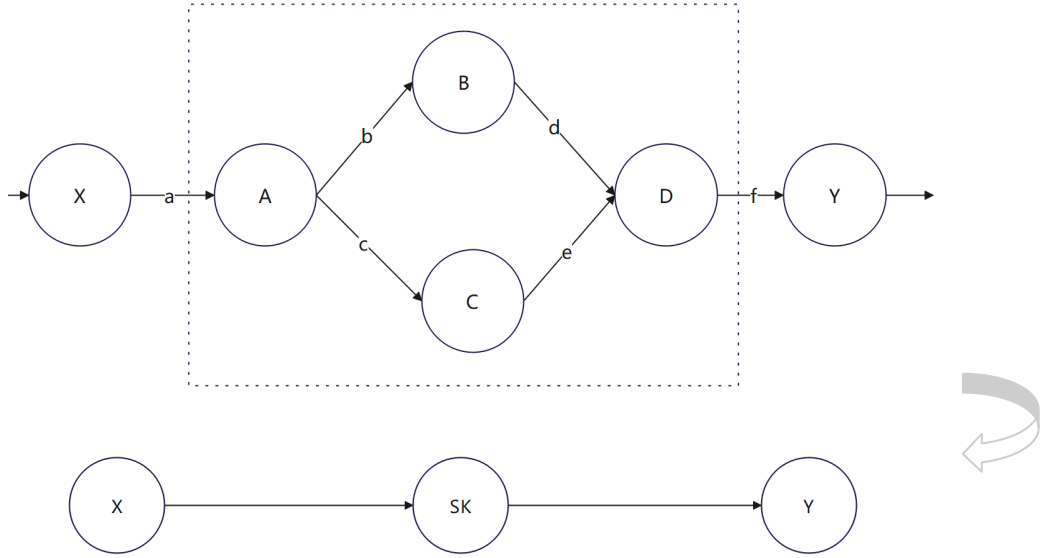
Super Kernel初衷就是节省这2-3ms的调度开销。



CANN

SuperKernel的基本流程





1、开发者编写torch脚本调用算子；

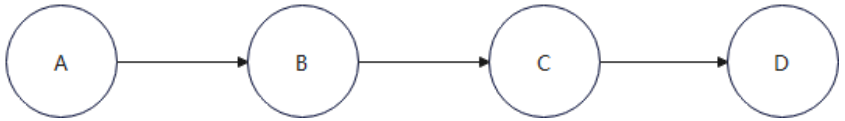
2、torchair标记算子的SuperKernel Scope信息；

3、GE图编译，根据Scope调用SuperKernel编译：
3.1 子节点算子jit编译；
3.2 SK算子调用代码生成和编译，并链接子算子；
3.3 SK task生成；

4、GE加载task执行。

CANN

SuperKernel代码示意



未开启SuperKernel情况下：

```
Launch(a, 40,stm, x,y,a_out);
Launch(b, 40, stm, a_out,y,b_out);
Launch(c, 40, stm, b_out,y,c_out);
Launch(d, 40, stm, c_out,y,d_out);
```

开启SuperKernel后：

```
Launch(sk, 40, stm, x,y,a_out,d_out);
```

原始kernel代码：

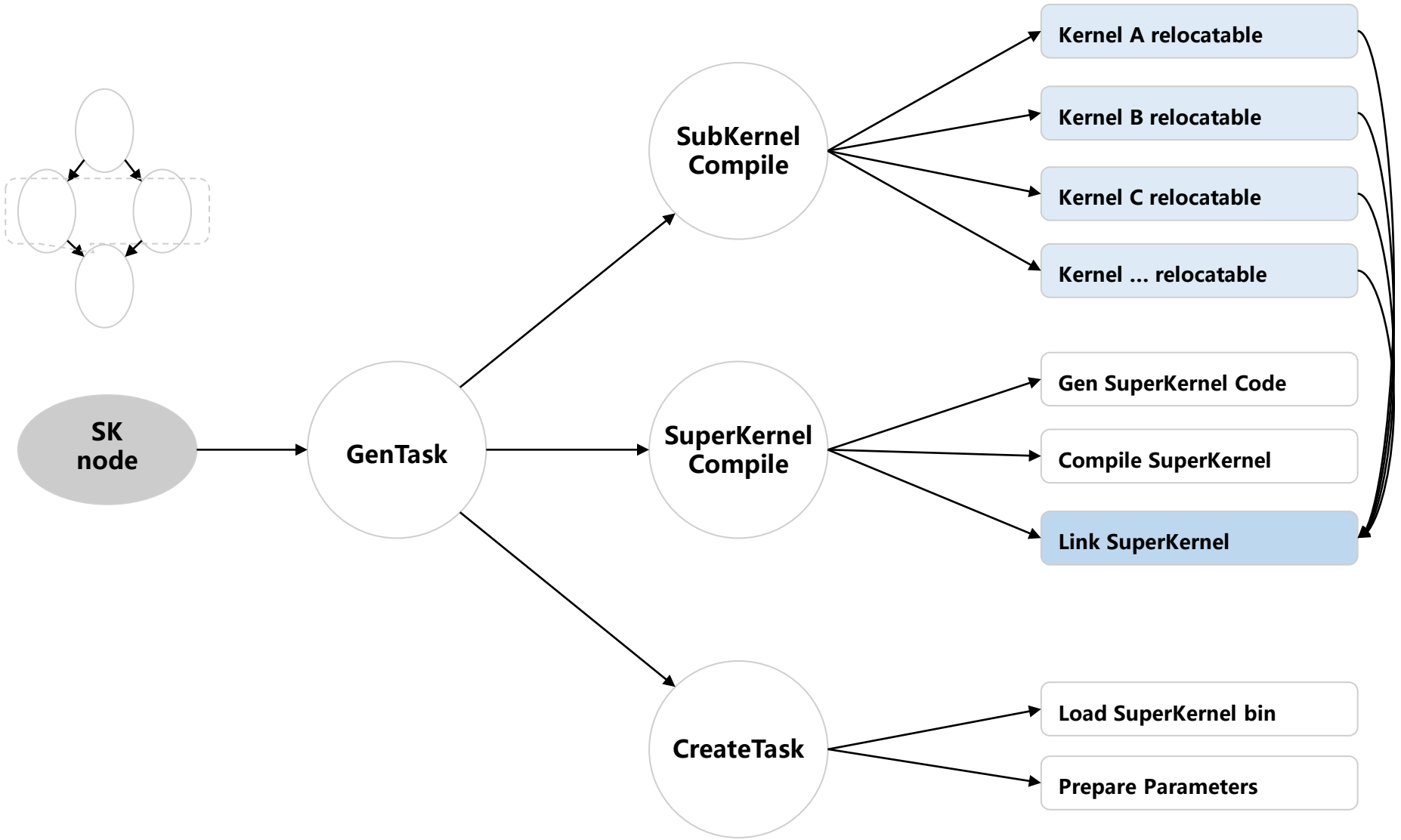
```
extern "C" __global__ __aicore__ void
add_100(GM_ADDR x, GM_ADDR y, GM_ADDR z)
{
   KernelAdd op;
   op.Init(x, y, z);
   op.Process();
}
```

SuperKernel代码：

```
extern "C" __aicore__ void add_100(GM_ADDR x, GM_ADDR y, GM_ADDR z);
extern "C" __aicore__ void add_200(GM_ADDR x, GM_ADDR y, GM_ADDR z);
extern "C" __aicore__ void add_300(GM_ADDR x, GM_ADDR y, GM_ADDR z);
extern "C" __aicore__ void add_400(GM_ADDR x, GM_ADDR y, GM_ADDR z);
extern "C" __global__ __aicore__ void add_skp(GM_ADDR x, GM_ADDR y,
GM_ADDR z)
{
   KERNEL_TASK_TYPE_DEFAULT(KERNEL_TYPE_MIX_AIV_1_0);
   preload((const void *)add_100,8);
   add_100(x, y, z);
   preload((const void *)add_200,PRE_LD_CNT);
   sync_all(w, 1);
   add_200(x, y, z);
   preload((const void *)add_300,PRE_LD_CNT);
   sync_all(w, 2);
   add_300(x, y, z);
   preload((const void *)add_400,PRE_LD_CNT);
   sync_all(w, 3);
   add_400(x, y, z);
}
```

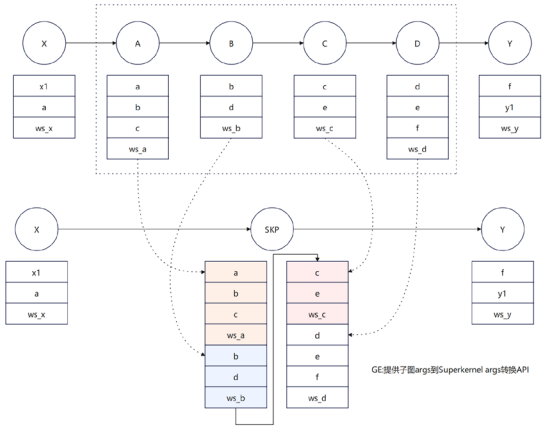
CANN

SuperKernel编译流程介绍



relocatbale: 基于function，参数通寄存器或者栈传递；
executable: 参数通过SPR传递，初始化栈基址；

SuperKernel代码生成：根据子kernel的依赖和执行顺序生成调用代码，并自动插入同步。



CANN

SuperKernel技术原理

进阶优化—ICache Preload 优化：

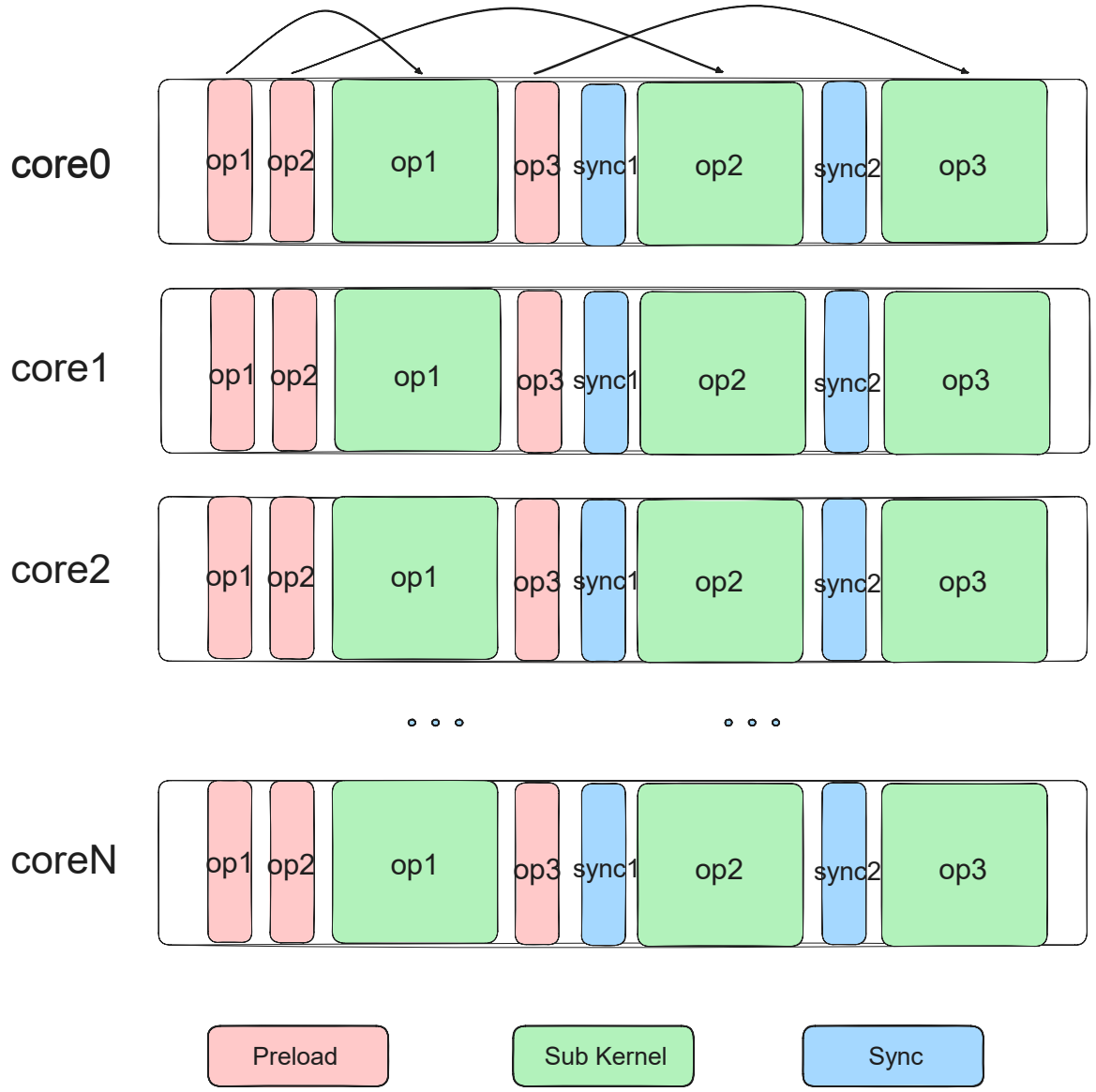
问题：

SuperKernel 融合全部算子后，其二进制体积较大。系统在加载算子时通常仅预取入口处指令，导致 SuperKernel 内部大量指令未被预加载至指令缓存（ICache），从而引发较高的 ICache Miss。

解决方案：

为子kernel提前进行preload代码段。Preload 一般按照Sub kernel的函数大小进行，2K对齐。

Load策略为当前子算子开始执行前，预加载其后续子算子的代码段。



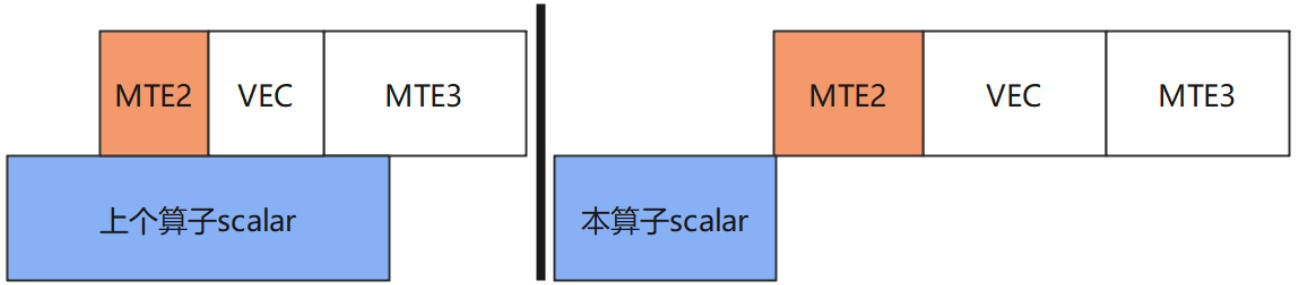
CANN

SuperKernel技术原理

进阶优化—Early-Start 优化：

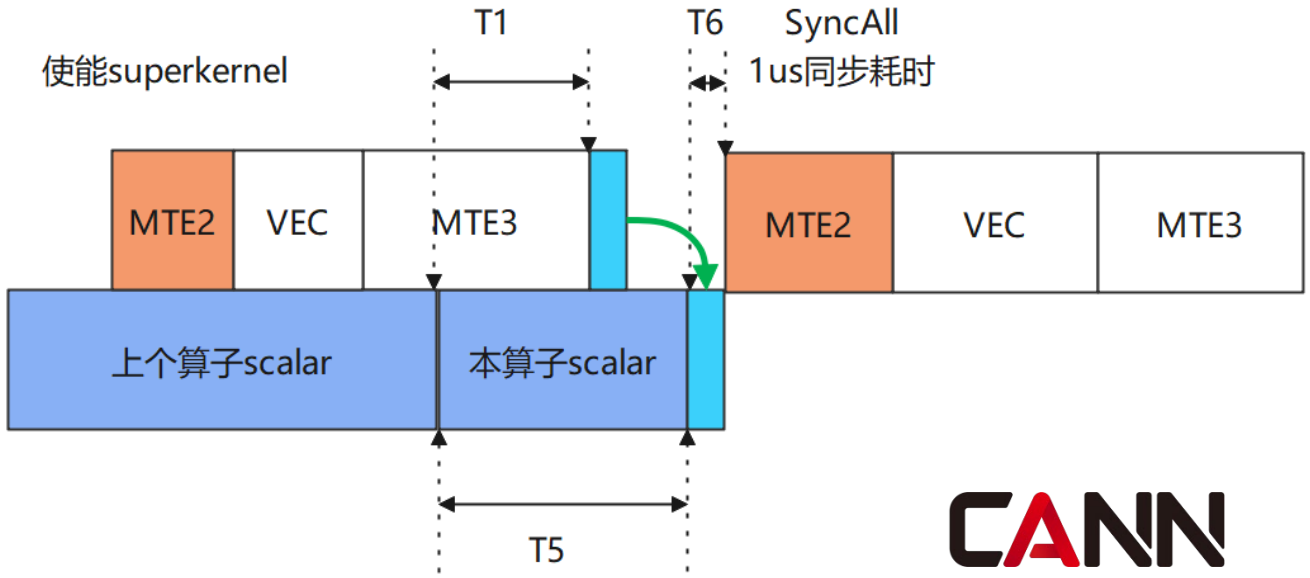
问题：

正常基于task调度或者SK内部同步前后两个算子之间需要等待所有硬件线程全部结束后才会进入下一算子。而对于算子而言，一般规律为启动时先执行scalar计算再做数据搬入计算搬出，可以利用前后流水空隙进行优化。



解决方案：

在上一个kernel的MTE3搬出的线程上做同步set，在下一个kernel的MTE2搬入前做一个wait（数据依赖）scalar可以不做kernel之间的同步等待。



CANN

SuperKernel技术原理

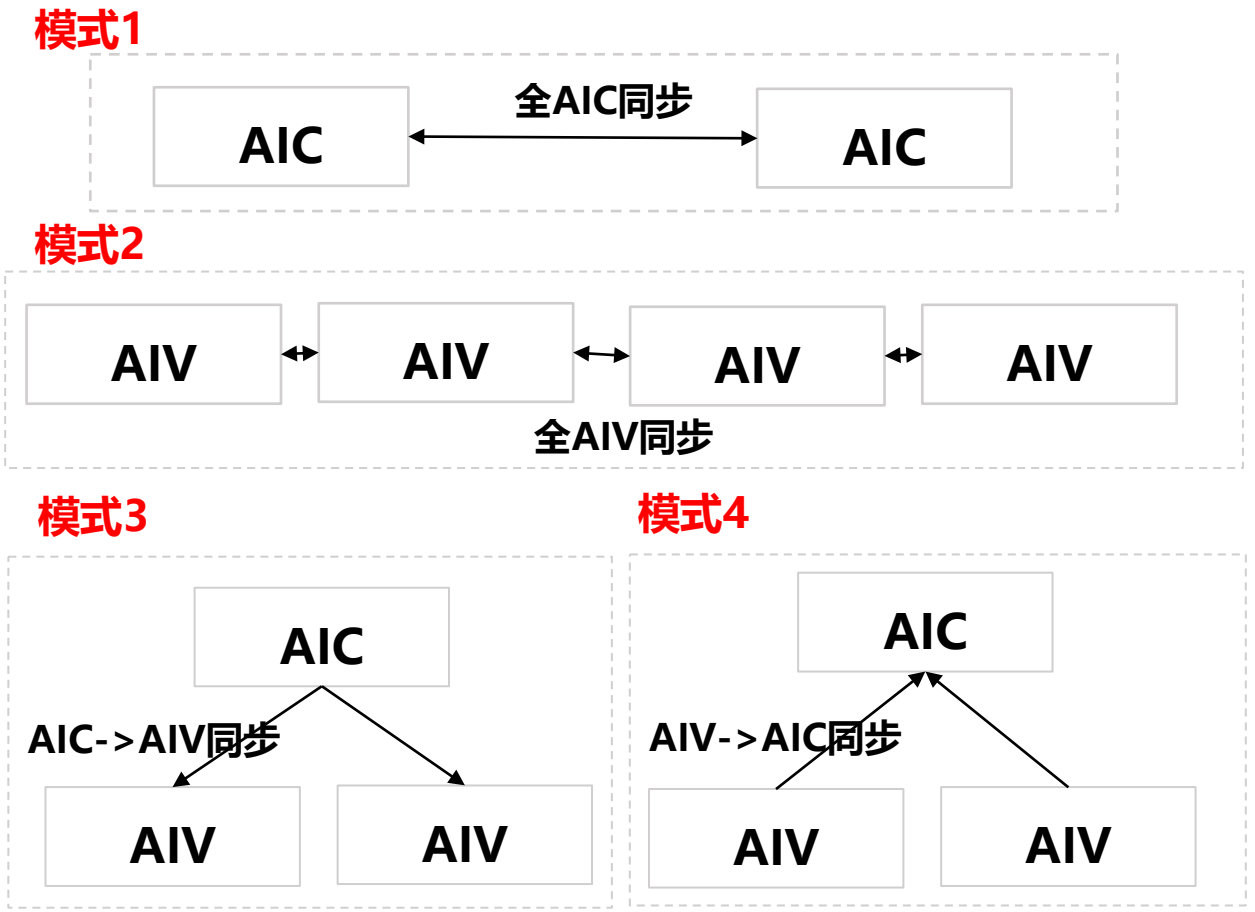
进阶优化—同步优化：

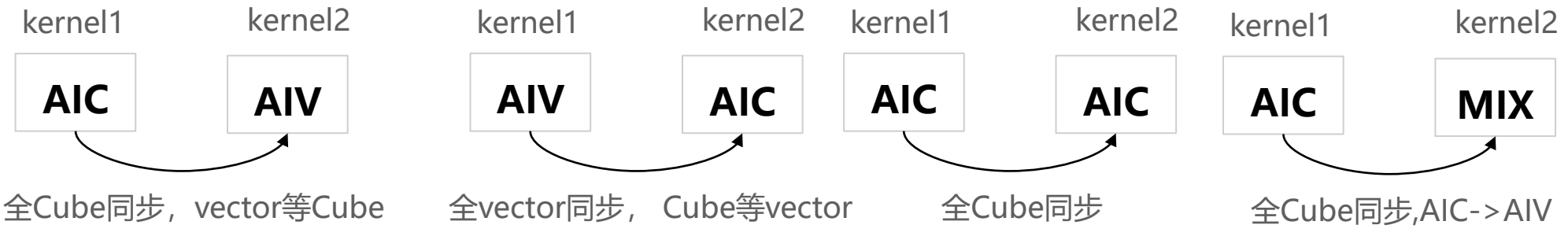
问题：

在SK两个kernel之间，解决数据依赖插入同步后，同步的开销较大，需要进一步将同步进行细粒度拆分优化。

解决方案：

在SK代码生成时，根据前后两个依赖的算子的类型，插入不同的同步模式来降低同步开销。





CANN

SuperKernel技术原理

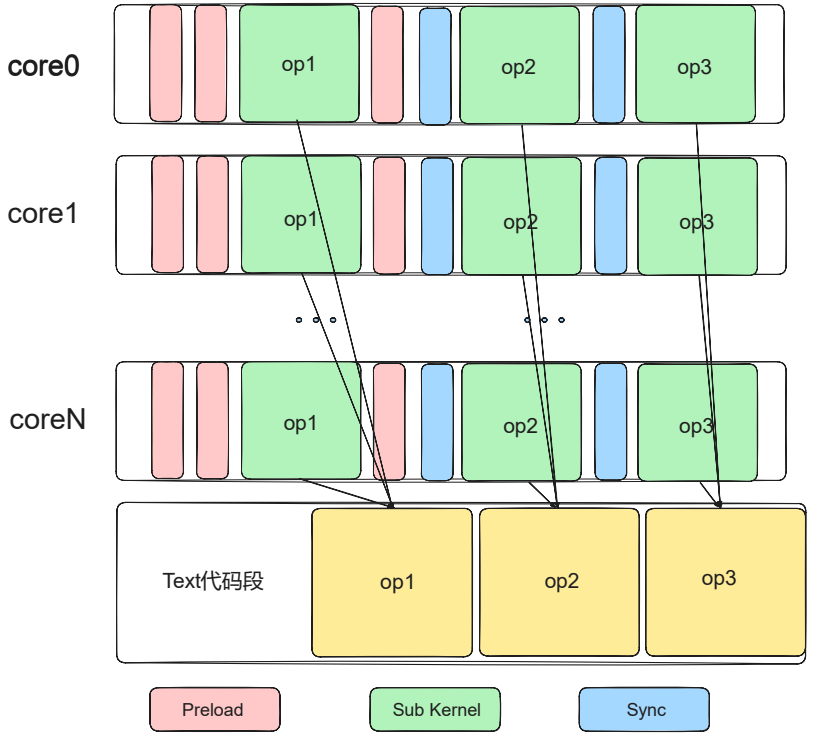
进阶优化—同地址访存优化和亲和部署：

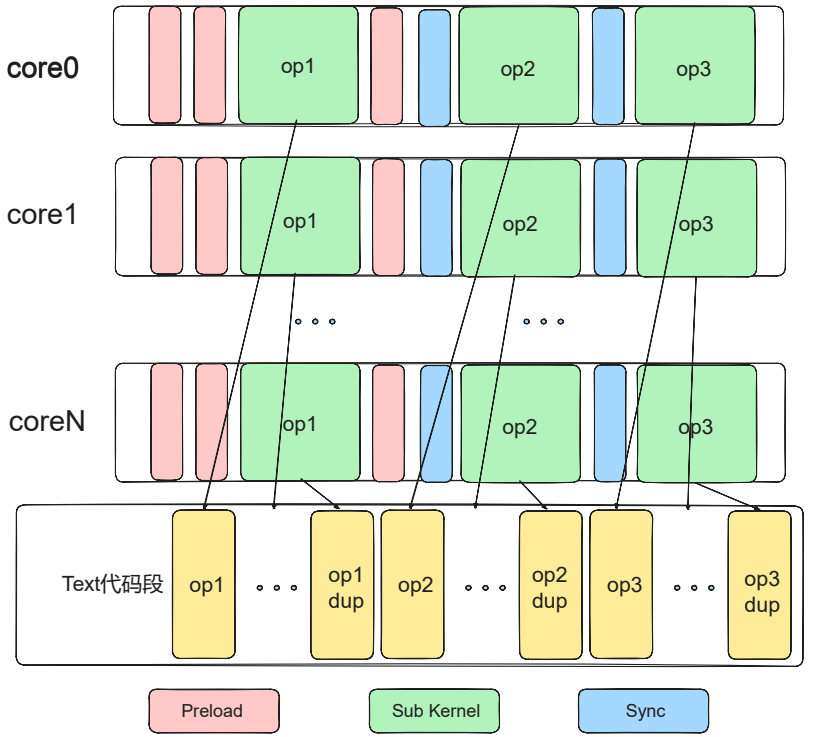
问题：

在多核系统中，当多个计算核心执行同一段代码时，会并发访问内存中的同一指令地址。这种对同一地址的并发访问会在共享的 L2 Cache 层面形成串行化访问队列，引发资源争用，削弱多核并行带来的性能增益。

解决方案：

SuperKernel 将子 Kernel 代码复制为多份副本，使不同核心能根据核 ID 映射到不同的物理地址执行。这一方法有效缓解了多核对同一指令地址的争用，显著提升算子执行效率。





CANN

SuperKernel技术原理

支持扩展—Tiling下沉执行和weight预取：

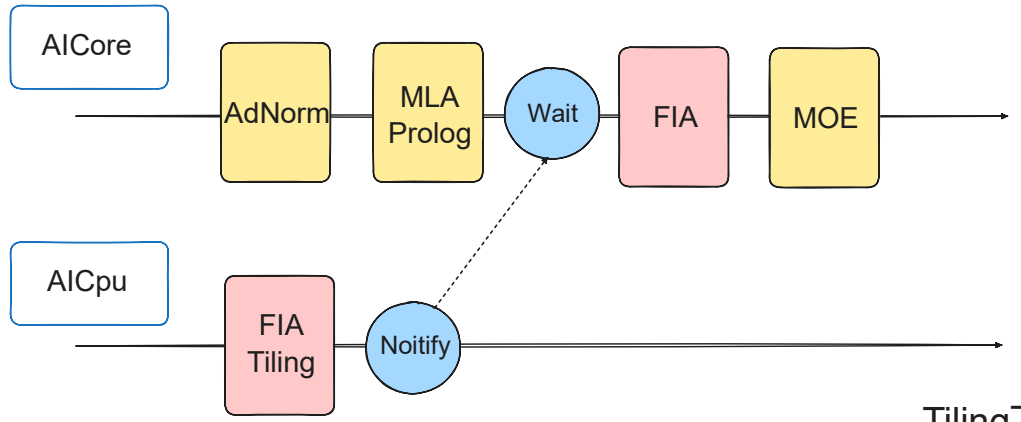
问题：

1、tiling下沉算子：算子tiling值依赖于device内存，此类算子通过将tiling加载到aicpu执行，并在tiling结束后通过同步事件通知aicore执行kernel。

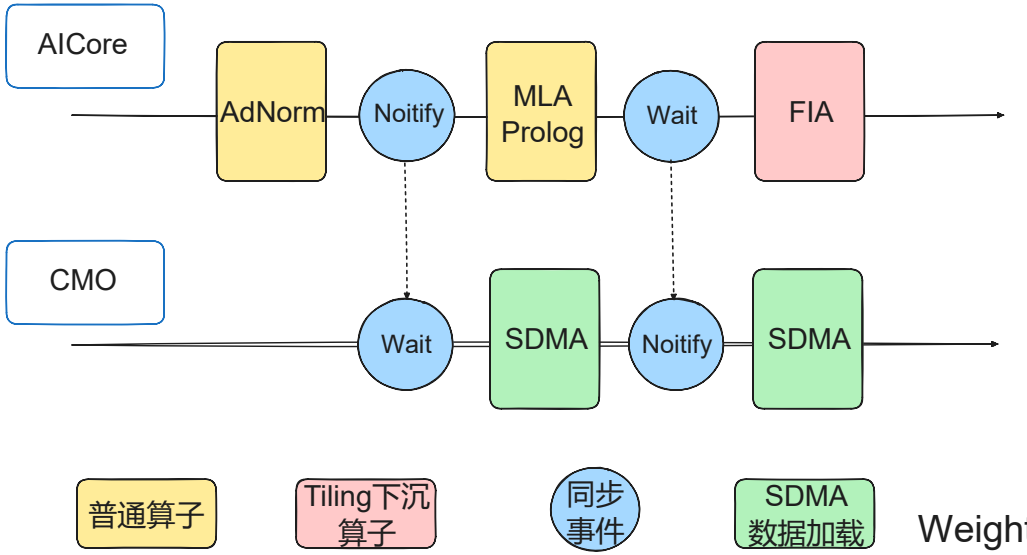
2、 Weight预加载机制在算子执行前触发CMO任务调用SDMA硬件将GM内存数据加载至L2cache。

解决方案：

提供基于内存语义的 Notify 和 Wait 事件，供SuperKernel与其他Stream的Task进行交互。



Tiling下沉



Weight预取

CANN

SuperKernel技术原理

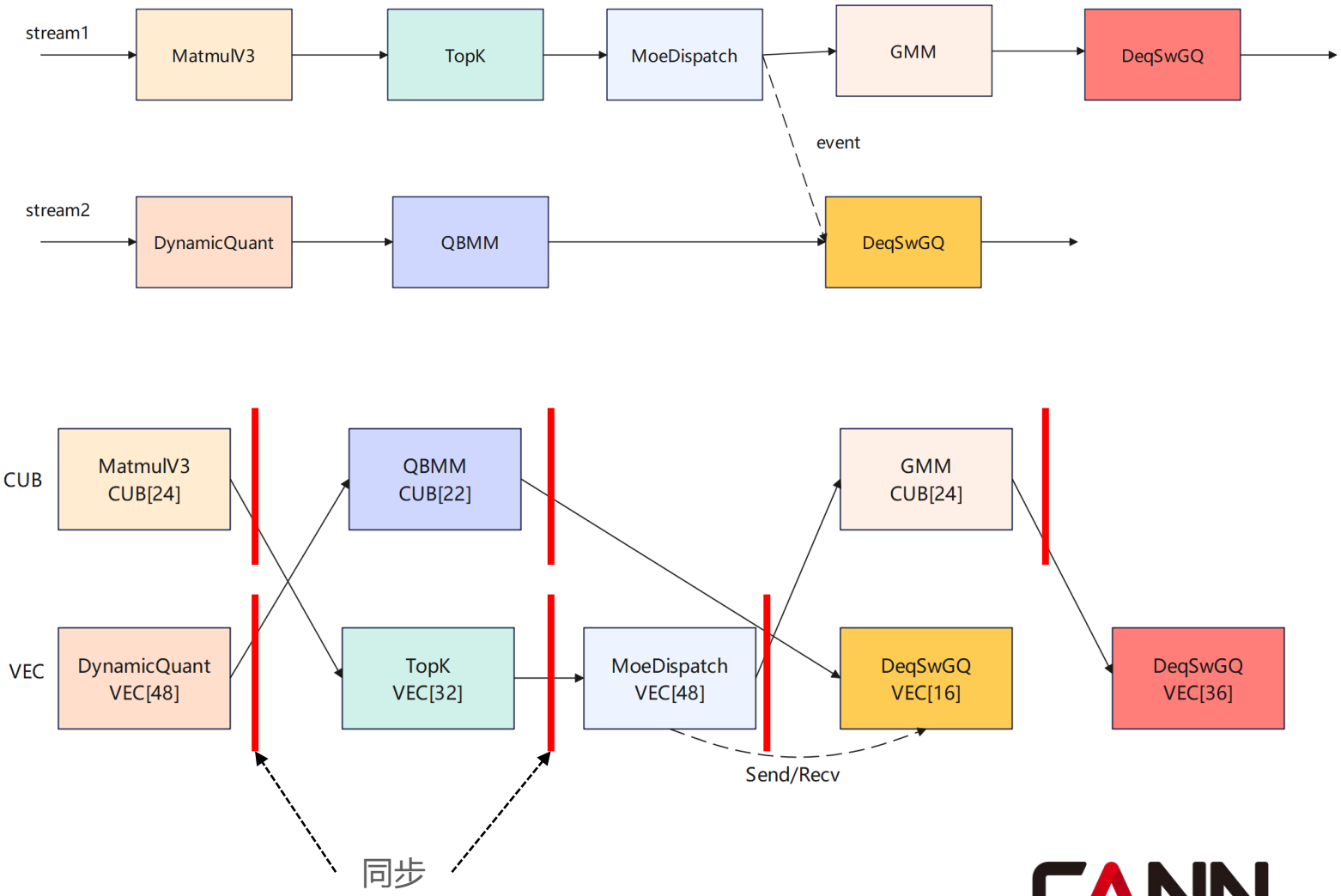
支持扩展—支持C/V多流并发：

问题：

通过多流实现C/V并发场景，在简单转换为Sk后，如果不感知依赖关系，只感知执行顺序，会导致SK内部串行执行，性能比融合前劣化。

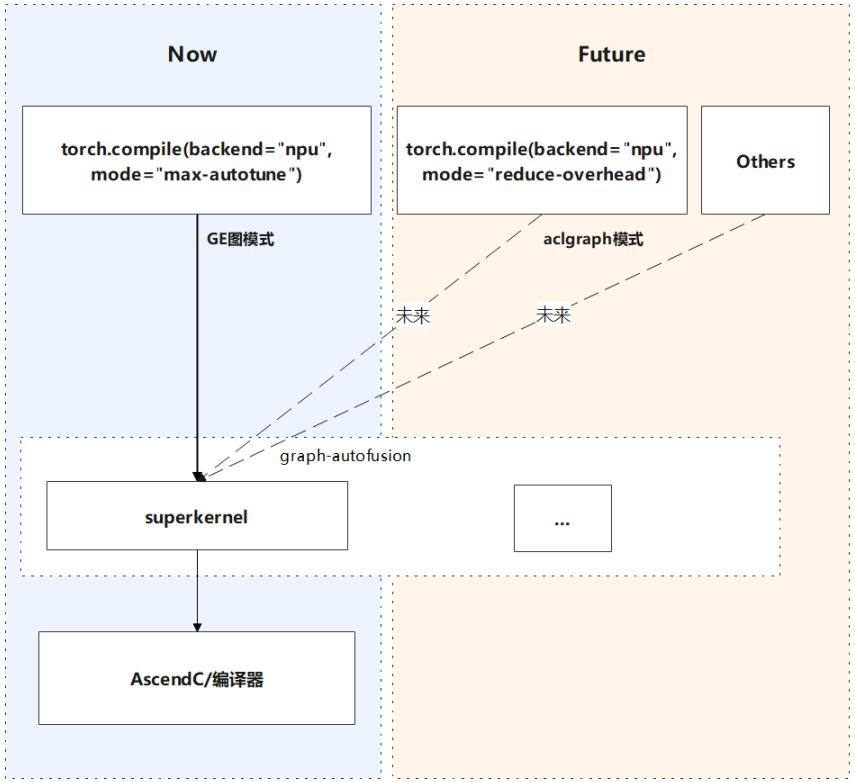
解决方案：

将算子按照类型分类，在SK内部根据Cube、Vector分配建立执行队列，并根据流的属性和event事件精准插入同步，实现C/V并发执行。



目录

SuperKernel前端框架使能及使用示例




```python
import torch
import torch_npu
import torchair as tng
import torchair._contrib.custom_torch_opsnpu_quant_matmul
from torchair.ge_concrete_graph import ge_apis as ge
from torchair.configs.compiler_config import CompilerConfig
class ModelSuperKernel(torch.nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, x1, x2, scale, offset, bias, pertoken_scale, weight_scale):
    # 将 npu_quant_matmul和npu_dequant_swiglu_quant融合为superKernel，标记为sp1
        with tng.scope.super_kernel("super_kernel_scope1"):
            quant_matmul_res_origin = torch_npu.npu_quant_matmul(x1, x2, scale, offset=offset, bias=bias,
pertoken_scale=pertoken_scale, output_dtype=torch.bfloat16)
            swiglu_res_origin = torch_npu.npu_dequant_swiglu_quant(quant_matmul_res_origin,
weight_scale=weight_scale)
        return swiglu_res_origin, quant_matmul_res_origin

config = CompilerConfig()
npu_backend = tng.get_npu_backend(compiler_config=config)
model = Network().npu()
# 省略构造input
#在npu上执行有superkernel配置的模型
model = torch.compile(model, fullgraph=True, backend=npu_backend, dynamic=False, mode="max-autotune")
npu_output = model(gmm1_x, gmm1_weight, gmm1_bias, gmm2_weight, moe1_bias, dsq_input, data1, data2, scale)
```

Superkernel支持场景

- 当前通过torch.compile支持max-autotune模式(GE图模式)
- 未来支持torch.compile下reduce-overhead模式(aclgraph)使用，并计划考虑支持更多种前端框架使能

torch.compile模式下使能

- 通过圈定范围方式，支持范围内算子融合成一个SuperKernel
- 支持范围内融合SuperKernel时自定义编译选项（eg: 'compile-options=-g’ ）
- 仅支持aicore上运行的Ascend C算子，对于其他非支持的，SuperKernel会跳过融合
- 仅支持静态shape，动态shape需要通过torch._dynamo.mark_static()标记该输入为静态

https://www.hiascend.com/document/detail/zh/Pytorch/710/modthirdparty/torchairuseguide/torchair_00035.html#ZH-CN_TOPIC_0000002389658173__section1658819498155

SuperKernel性能收益--- Example收益对比

No SuperKernel			
OP Type	Core Type	Count	Total Time(us)
GroupedMatmul	MIX_AIC	15	126.26
Transpose	AI_VECTOR_CORE	10	90.02
MoeGatingTopK	AI_VECTOR_CORE	5	68.32
Tile	AI_VECTOR_CORE	4	24.96
DequantSwigluQuant	AI_VECTOR_CORE	5	24.18
ReduceMeanD	MIX_AIV	1	16.36
ConcatV2D	AI_VECTOR_CORE	5	14.9
ReduceMeanD	AI_VECTOR_CORE	4	14.14
SplitVD	AI_VECTOR_CORE	1	10.04
MatMul	AI_CORE	2	6.26
Cast	AI_VECTOR_CORE	3	3.96
Data	AI_VECTOR_CORE	1	3.3
StridedSliceD	AI_VECTOR_CORE	3	3.18
AutomaticBufferFusionOp	AI_VECTOR_CORE	1	1.66
Total		60	407.54

SuperKernel			
OP Type	Core Type	Count	Total Time(us)
SuperKernel	MIX_AIC	5	172.4
Transpose	AI_VECTOR_CORE	10	92.42
Tile	AI_VECTOR_CORE	4	24.66
SuperKernel	MIX_AIV	4	18.48
ReduceMeanD	MIX_AIV	1	16.34
ConcatV2D	AI_VECTOR_CORE	5	14.74
ReduceMeanD	AI_VECTOR_CORE	4	14.24
SplitVD	AI_VECTOR_CORE	1	10.12
MatMul	AI_CORE	2	8.6
Cast	AI_VECTOR_CORE	3	4.08
Data	AI_VECTOR_CORE	1	3.72
StridedSliceD	AI_VECTOR_CORE	3	3.1
AutomaticBufferFusionOp	AI_VECTOR_CORE	1	1.76
Total		44	384.66

SupperKernel性能优化数据对比

- 算子总数 60->44 (25% ↓)
- 算子执行性能407us->384us（5% ↓ ）

SuperKernel性能收益---客户实际场景性能收益

在XX平台的灰度版本中，通过启用混合专家（MoE）功能并对涉及的8个算子进行SuperKernel融合优化，整体网络延迟降低了1.7毫秒，显著提升了系统响应速度。在XX客户 64P设备上，通过采用96 Batch的特定配置，实现了4.8毫秒的性能提升，优化了大规模数据批处理的效率。

在自验证平台实验中，结合CANN提供的7个内置优化算子与自定义的12个算子，成功将解码阶段的延迟从31毫秒降低到25毫秒，性能提升幅度达到20%。此外，针对MMoE（Multi-gate Mixture-of-Experts）推荐网络，通过基于SuperKernel对TBE算子进行实验性质穿刺扩展，实现了整网性能10%至20%的提升，。

Graph-autofusion仓交流

群聊：CANN开源社区交流1群

该二维码7天内(11月4日前)有效，重新进入将更新

代码仓地址：
https://gitcode.com/cann/graph-autofusion

欢迎Star及参与贡献，共建CANN生态！

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