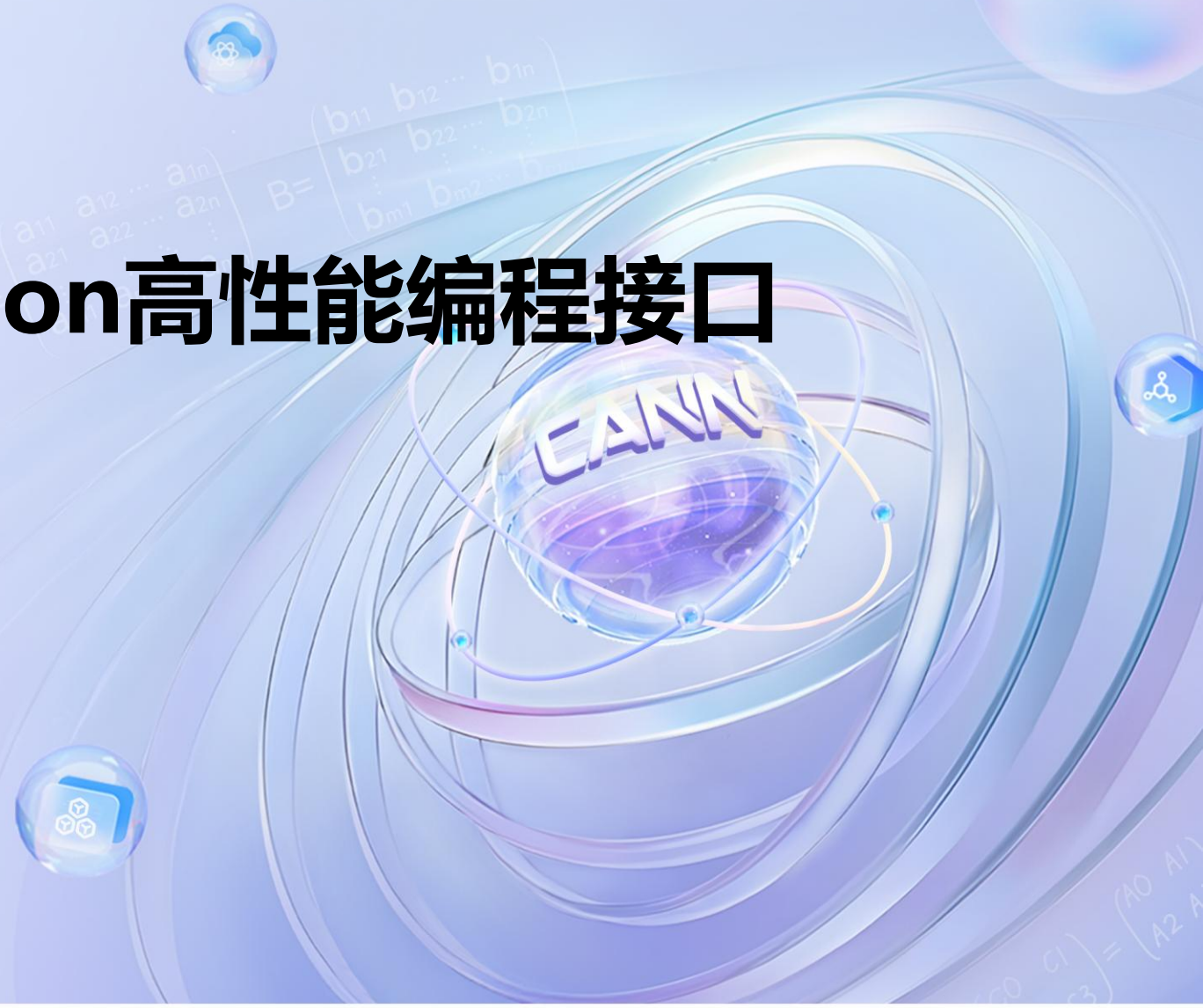
PyAsc: 完备的Python高性能编程接口

作者：秦名扬
时间：2025-12-02



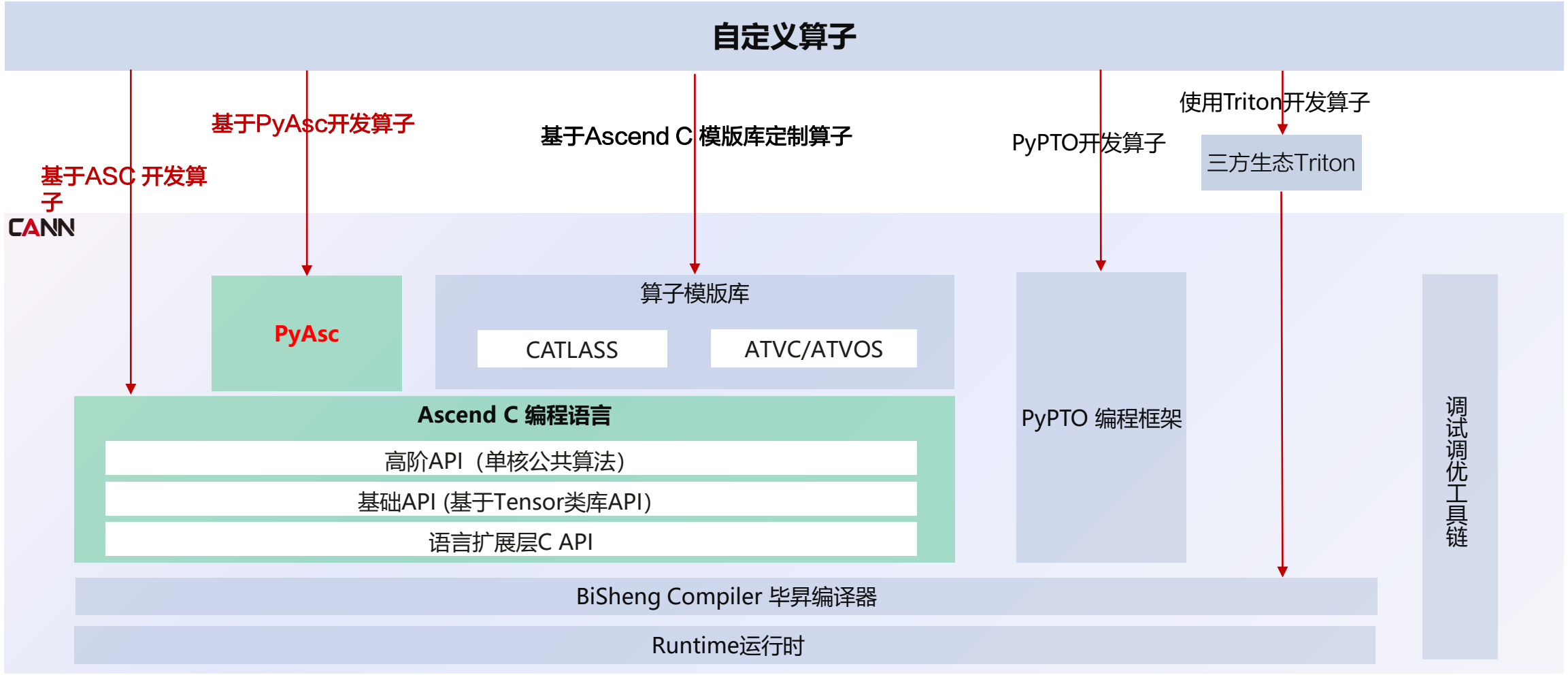
CANN

目录

- Part 1 什么是PyAsc?
- Part 2 PyAsc技术原理和示例
- Part 3 PyAsc调试调优方法

CANN

Ascend C 编程体系



CANN

PyAsc VS Ascend C

```python
@overload
def add(dst: LocalTensor, src0: LocalTensor, src1: LocalTensor, count: int, is_set_mask: bool = True) -> None:
    ...

@overload
def add(dst: LocalTensor, src0: LocalTensor, src1: LocalTensor, mask: int, repeat_times: int,
        repeat_params: BinaryRepeatParams, is_set_mask: bool = True) -> None:
    ...

@overload
def add(dst: LocalTensor, src0: LocalTensor, src1: LocalTensor, mask: List[int], repeat_times: int,
        repeat_params: BinaryRepeatParams, is_set_mask: bool = True) -> None:
    ...
```

共同点：

- 支持同样的**昇腾AI处理器**
- 支持同样的**异构并行+SPMD编程模型**
- 支持同样的**基础API**

不同点：

- PyAsc支持Torch.Tensor作为参数，**无需额外的Torch封装**
- PyAsc使用通用Python语法，**无需学习C++ Template**
- PyAsc支持JIT按需编译，**不用编译好所有模板**

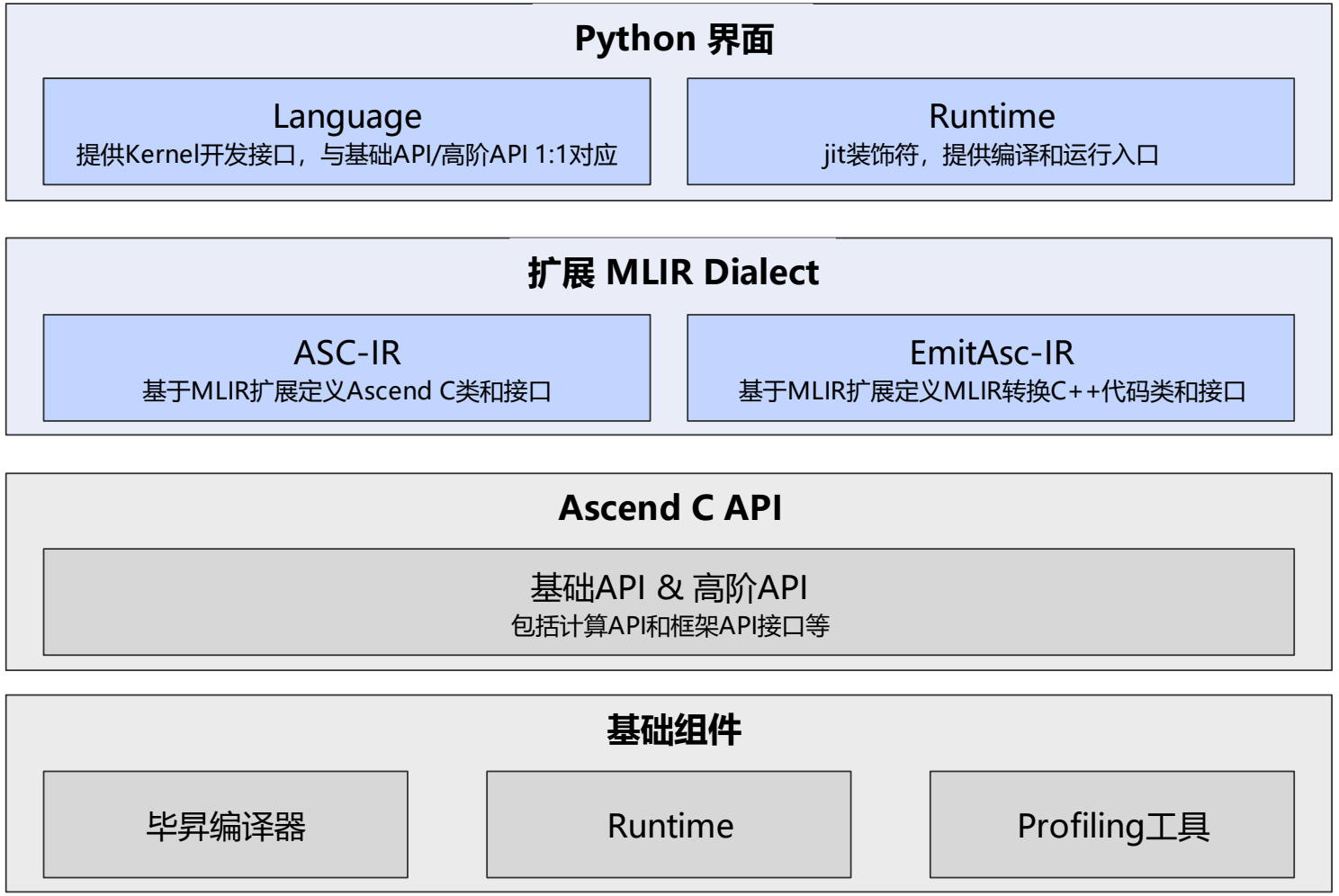
```cpp
template <typename T>
__aicore__ inline void Add(const LocalTensor<T>& dst, const LocalTensor<T>& src0,
                           const LocalTensor<T>& src1, const int32_t& count);

template <typename T, bool isSetMask = true>
__aicore__ inline void Add(const LocalTensor<T>& dst, const LocalTensor<T>& src0,
                           const LocalTensor<T>& src1, uint64_t mask[], const uint8_t repeatTime,
                           const BinaryRepeatParams& repeatParams);

template <typename T, bool isSetMask = true>
__aicore__ inline void Add(const LocalTensor<T>& dst, const LocalTensor<T>& src0,
                           const LocalTensor<T>& src1, uint64_t mask, const uint8_t repeatTime,
                           const BinaryRepeatParams& repeatParams);
```


CANN

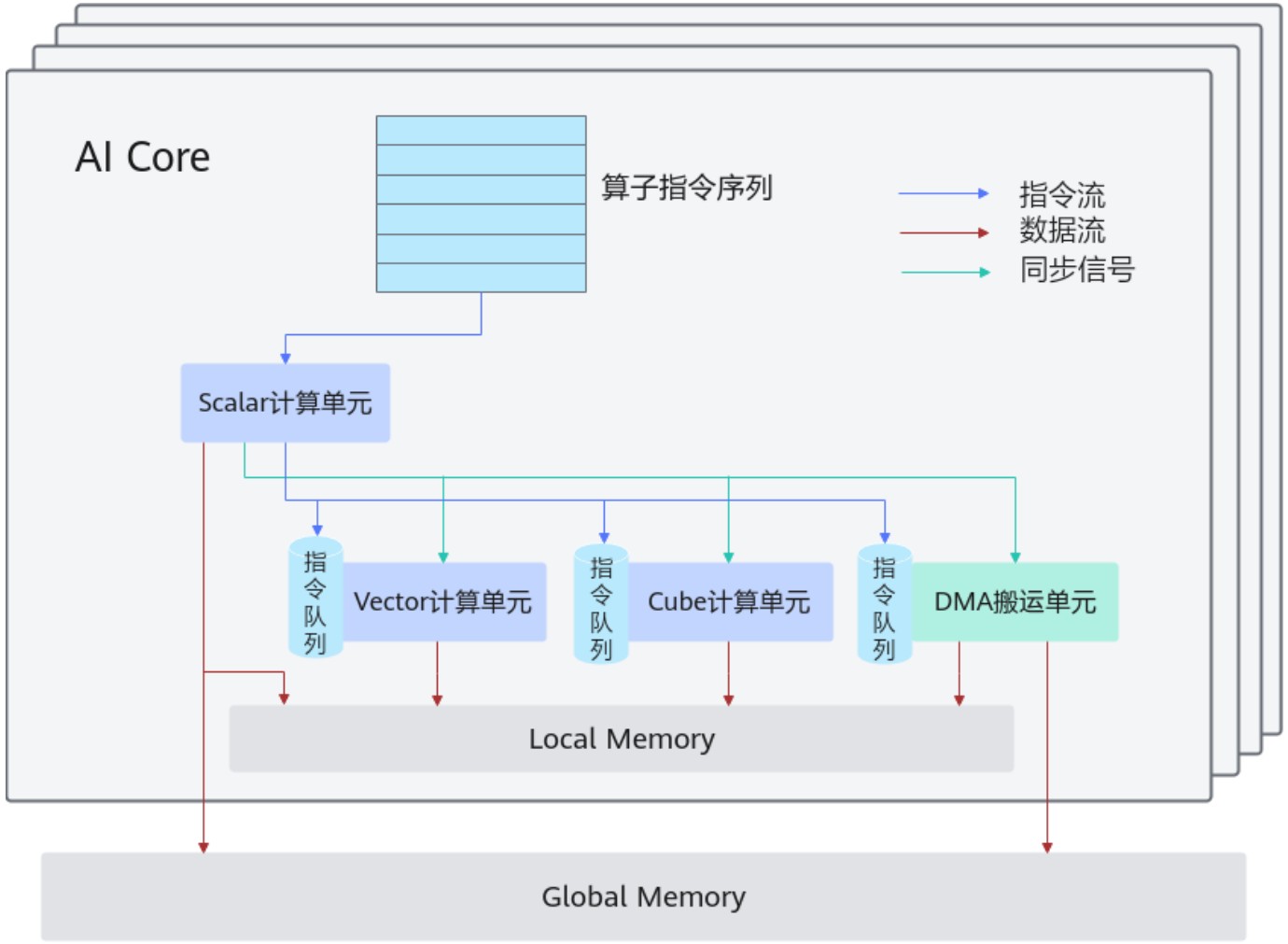
什么是PyAsc?



- **Python原生**：PyAsc是一种用于编写高效自定义算子的编程语言，**原生支持python**标准语法，可直接与**Torch和TorchNpu**协作。
- **JIT编译和运行**：基于PyAsc编写的算子程序，通过JIT编译和运行时调度，运行在**昇腾AI处理器**上。
- **芯片全量能力**： PyAsc编程接口与Ascend C接口 一一映射，提供**芯片全量的编程能力**，目前正在逐渐开放中。

CANN

昇腾AI处理器——抽象硬件架构



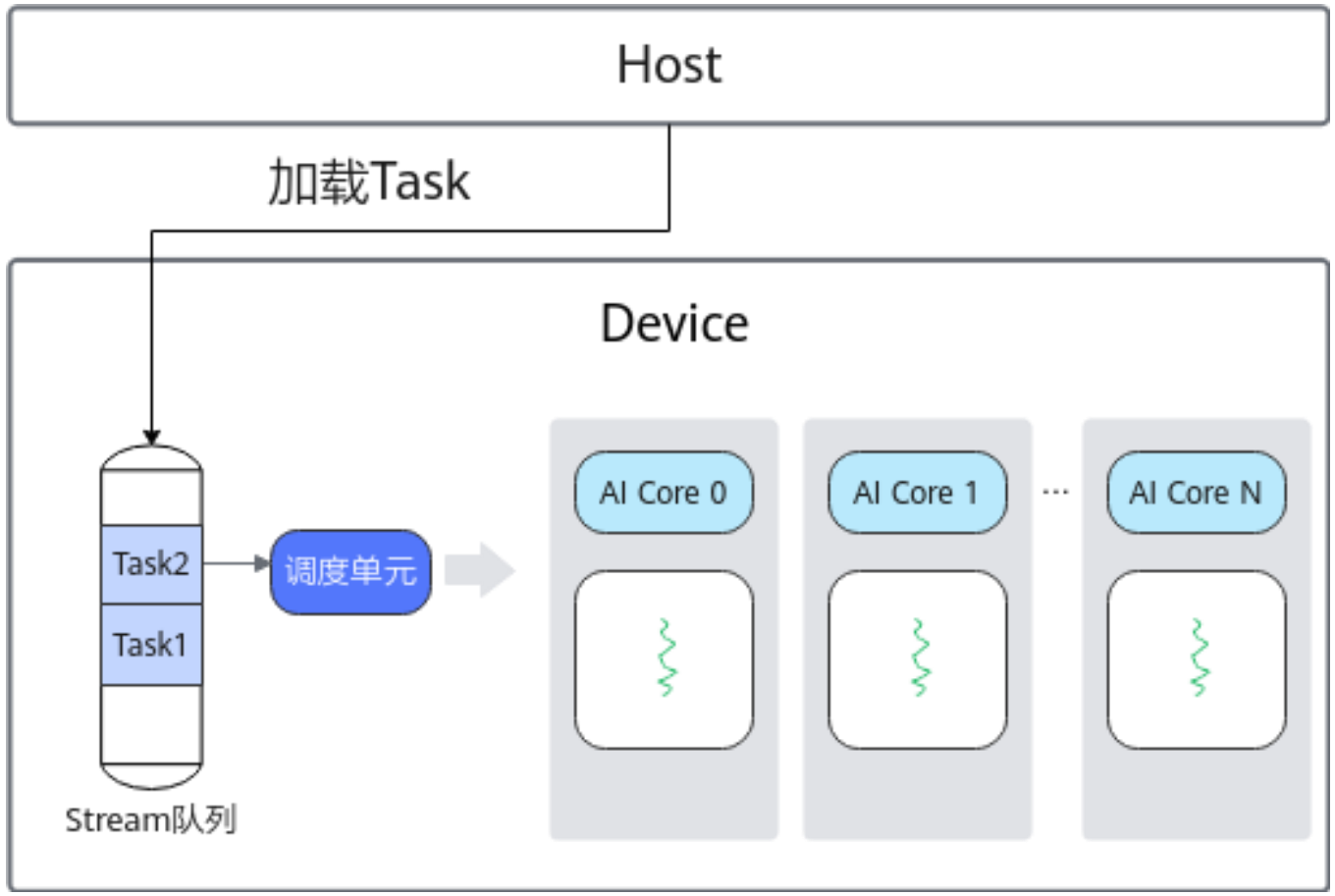
AI Core是昇腾AI处理器的计算核心，昇腾AI处理器内部有多个AI Core。AI Core中包含**计算单元、存储单元、搬运单元**等核心组件。

- 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**。
  - 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：负责数据搬运，包括Global Memory和Local Memory之间的数据搬运，以及不同层级Local Memory之间的数据搬运。

CANN

编程模型——异构并行

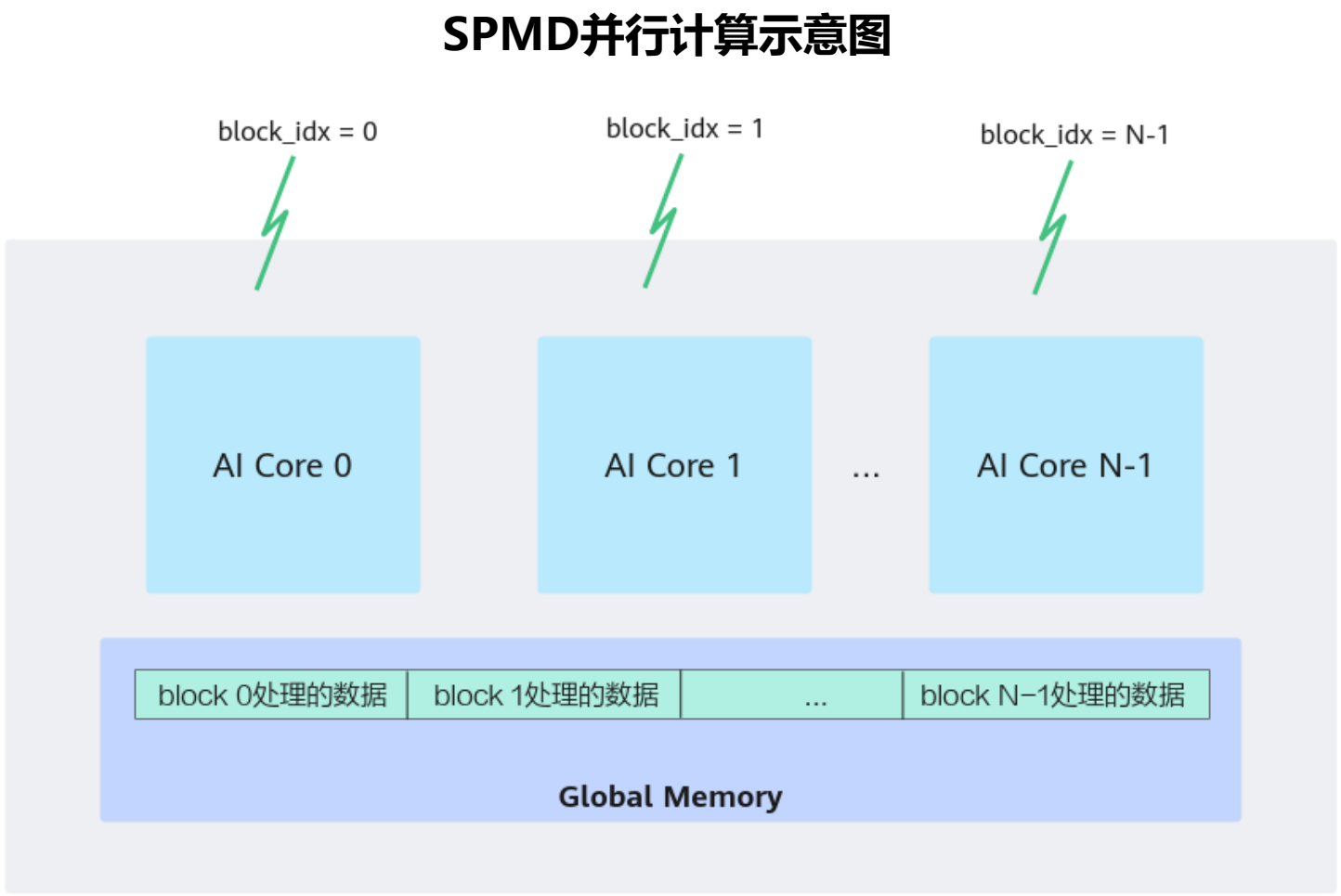
Kernel调度示意图



- 在PyAsc异构并行编程模型中，系统被分为了Host侧和Device侧，两者协同完成计算任务：
  - Host侧主要负责**运行时管理**，包括存储、设备以及Stream管理等，确保任务的高效调度；
  - Device侧，会执行开发者编写的**Kernel核函数**，完成批量数据的矩阵运算、向量运算等计算密集型的任务，用于计算加速。
- 一个Kernel按照以下步骤下发到AI Core上执行：
  - 运行时管理模块根据开发者设置的**核数和任务类型**启动对应的Task；
  - Task从Host加载到Device的**Stream运行队列**；
  - 调度单元将就绪的Task分配到空闲AI Core执行。

CANN

编程模型——SPMD模型



- PyAsc算子编程使用**SPMD（Single-Program Multiple-Data）**模型，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。
- 具体到PyAsc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**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。

PyAsc代码仓目录介绍

```
├── bin                 # 工具文件
├── docs                # 说明文档
│   ├── figures           ## 文档图片
│   └── python-api        ## 编程接口文档
├── include             # 后端头文件和td文件
│   └── ascir             ## ascir头文件和td文件
├── lib                 # 后端源文件
│   ├── Dialect           ## mlir方言定义源文件
│   ├── TableGen          ## tablegen扩展代码文件
│   └── Target            ## mlir目标代码转换源文件
├── python              # python前端代码
│   ├── asc               ## 用户可见的python包
│   ├── src               ## pybind相关代码，cpp格式
│   ├── test              ## python格式的测试用例集
│   └── tutorials         ## 供用户参考的样例集
└── test                # 后端的测试用例集
    ├── Dialect           ## mlir方言定义模块测试用例
    ├── Target            ## mlir目标代码转换模块测试用例
    └── tools             ## 后端工具相关测试用例
```

- **docs:** 模块与架构介绍文档、编程接口文档、算子调试调优指南等。
- **ascir:** 包含ASC-IR定义的TableGen文件和头文件。
- **asc:** 用户编写算子kernel所需要的接口、编译和运行接口、访问python语法树生成后端mlir的代码。
- **tutorials:** 供用户参考的样例代码以及样例说明。

代码仓链接：https://gitcode.com/cann/pyasc

CANN

PyAsc代码仓路标

2025.11.30 基本能力开放

- PyAsc开源
- 支持**910B**常用基础API
- 支持**jit编译和运行**等基本能力

2026.Q1 易用性提升

- 支持**自动同步**
- 支持**自动内存管理**
- 支持更多高阶API

2026.Q2 更多硬件& 高性能能力

- 支持**950**新增基础API
- 支持950新增**SIMT API**
- 支持类Cute DSL编程

https://gitcode.com/cann

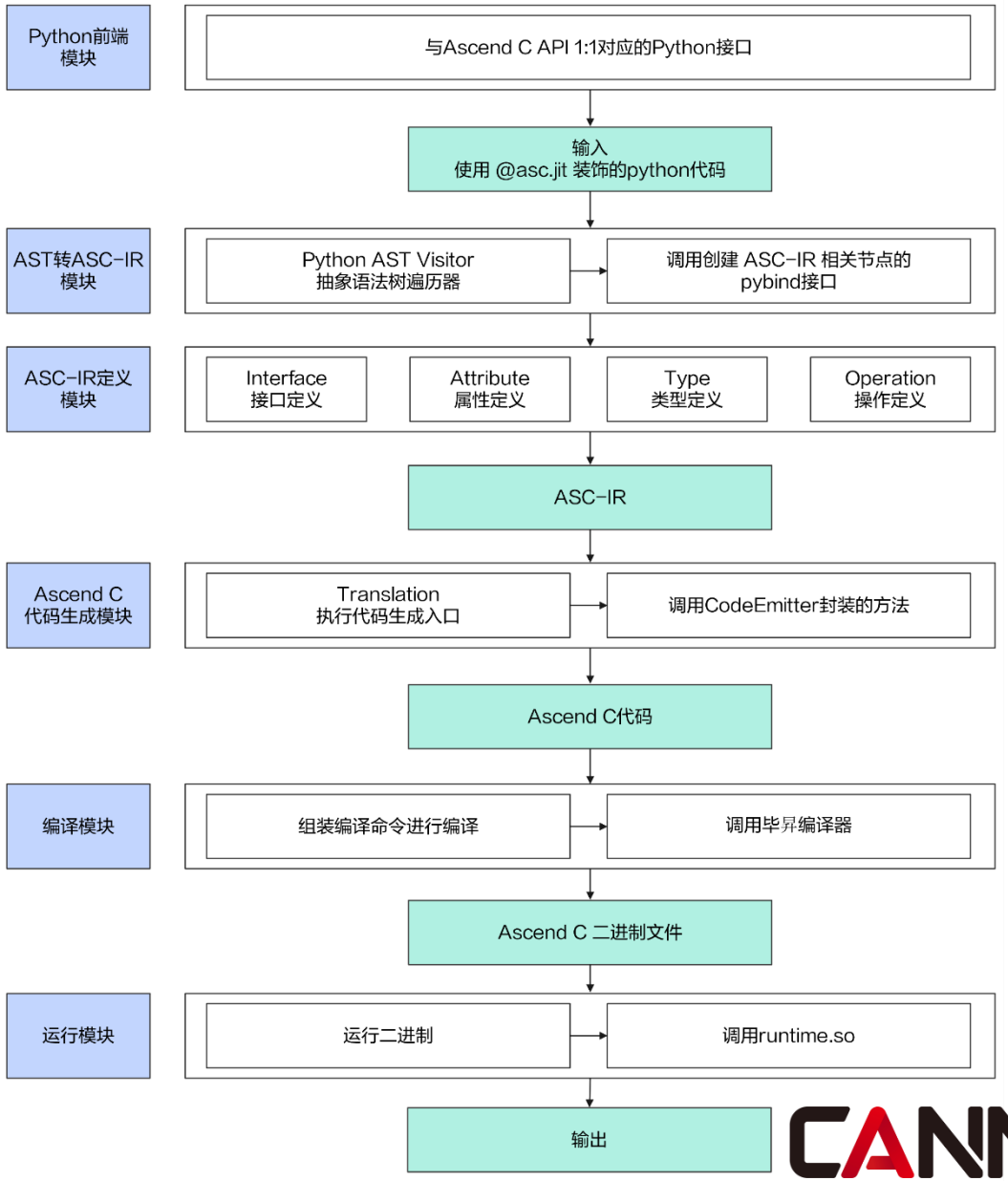
CANN

目录

- Part 1 什么是PyAsc?
- Part 2 PyAsc技术原理和示例
- Part 3 PyAsc调试调优方法

PyAsc技术原理

1. **Python前端模块**：提供与Ascend C API接口**一一对应**的Python编程接口。
2. **AST转ASC-IR模块**：将**Python AST**（Abstract Syntax Tree，抽象语法树）转换为**ASC-IR**（Ascend C中间表示）的功能。
3. **ASC-IR定义模块**：基于MLIR扩展**自定义Dialect**——ASC-IR。
4. **Ascend C代码生成模块**：实现 MLIR 中间表示（ASC-IR）到 **Ascend C 代码**的转换逻辑。
5. **编译和运行模块**：通过JIT(Just-In-Time，即时编译)机制拉起，开发者用装饰器**@asc.jit**修饰Kernel函数，拉起整个Kernel函数的编译和运行。



PyAsc算子示例——Kernel

算子示例目录结构

```
python/tutorials
├── 01_add                      # 手动插入同步流水的Add算子样例
│   ├── add.py
│   └── README.md
├── 02_add_framework            # Ascend C框架插入流水同步的Add算子样例
│   ├── add_framework.py
│   └── README.md
├── 03_matmul_mix               # MIX模式的Matmul算子样例
│   ├── matmul_mix.py
│   └── README.md
├── 04_matmul_cube_only         # 纯Cube模式的Matmul算子样例
│   ├── matmul_cube_only.py
│   └── README.md
├── 05_matmul_leakyrelu         # Matmul Leaky Relu算子样例
│   ├── matmul_leakyrelu.py
│   └── README.md
└── README.md                   # 当前提供的算子样例总览说明
```

① asc.jit装饰器修饰Kernel函数，在设备侧上运行。

② 输入输出为全局内存地址GlobalAddress，指向外部存储Global Memory。

③ 遵循编程范式，调用copy_in → compute → copy_out，完成数据搬入 → 计算 → 数据搬出整个算子处理流程。

④ 在compute函数中，调用**add**的python api完成元素求和矢量计算。

⑤ vadd_kernel核函数，通过小括号()传入输入输出参数，通过中括号[]传入启动配置（类同于Ascend C的<<<…>>>调用方式）。

⑥ 支持与torch协作，通过torch构造输入输出tensor完成计算。

Add算子示例

```python
@asc.jit                                                                                                    # ①
def vadd_kernel(x: asc.GlobalAddress, y: asc.GlobalAddress, z: asc.GlobalAddress, block_length: int,       # ②
                tile_length: asc.ConstExpr[int]):
    offset = asc.get_block_idx() * block_length
    x_gm = asc.GlobalTensor()
    y_gm = asc.GlobalTensor()
    z_gm = asc.GlobalTensor()
    x_gm.set_global_buffer(x + offset)
    y_gm.set_global_buffer(y + offset)
    z_gm.set_global_buffer(z + offset)
    pipe = asc.TPipe()
    in_queue_x = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    in_queue_y = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    out_queue_z = asc.TQue(asc.TPosition.VECOUT, BUFFER_NUM)
    pipe.init_buffer(in_queue_x, BUFFER_NUM, tile_length * x.dtype.sizeof())
    pipe.init_buffer(in_queue_y, BUFFER_NUM, tile_length * y.dtype.sizeof())
    pipe.init_buffer(out_queue_z, BUFFER_NUM, tile_length * z.dtype.sizeof())
    for i in range(TILE_NUM * BUFFER_NUM):
        copy_in(i, x_gm, y_gm, in_queue_x, in_queue_y, tile_length)                                         # ③
        compute(z_gm, in_queue_x, in_queue_y, out_queue_z, tile_length)
        copy_out(i, z_gm, out_queue_z, tile_length)

@asc.jit
def compute(z_gm: asc.GlobalTensor, in_queue_x: asc.TQue, in_queue_y: asc.TQue, out_queue_z: asc.TQue,
            tile_length: asc.ConstExpr[int]):
    # "z_gm" is passed here to obtain dtype
    x_local = in_queue_x.deque(z_gm.dtype)
    y_local = in_queue_y.deque(z_gm.dtype)
    z_local = out_queue_z.alloc_tensor(z_gm.dtype)
    asc.add(z_local, x_local, y_local, tile_length)                                                         # ④
    out_queue_z.enque(z_local)
    in_queue_x.free_tensor(x_local)
    in_queue_y.free_tensor(y_local)

def vadd_launch(x: torch.Tensor, y: torch.Tensor) -> torch.Tensor:                                          # ⑥
    z = torch.zeros_like(x)
    total_length = z.numel()
    block_length = (total_length + USE_CORE_NUM - 1) // USE_CORE_NUM                                        # ⑤
    tile_length = block_length // TILE_NUM // BUFFER_NUM
    vadd_kernel[USE_CORE_NUM, rt.current_stream()](x, y, z, block_length, tile_length)
    return z


def vadd_custom(backend: config.Backend):
    config.set_platform(backend)
    device = "npu" if config.Backend(backend) == config.Backend.NPU else "cpu"
    size = 8 * 2048
    x = torch.rand(size, dtype=torch.float32, device=device)                                                # ⑥
    y = torch.rand(size, dtype=torch.float32, device=device)
    z = vadd_launch(x, y)
    assert torch.allclose(z, x + y)
```

具体代码示例请参考： https://gitcode.com/cann/pyasc/blob/master/python/tutorials/02_add_framework/add_framework.py

PyAsc算子示例——MLIR

Python代码

```python
@asc.jit
def vadd_kernel(x: asc.GlobalAddress, y: asc.GlobalAddress, z: asc.GlobalAddress,
                block_length: int, tile_length: asc.ConstExpr[int]):
    offset = asc.get_block_idx() * block_length
    x_gm = asc.GlobalTensor()
    y_gm = asc.GlobalTensor()
    z_gm = asc.GlobalTensor()
    x_gm.set_global_buffer(x + offset)
    y_gm.set_global_buffer(y + offset)
    z_gm.set_global_buffer(z + offset)
    pipe = asc.TPipe()
    in_queue_x = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    in_queue_y = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    out_queue_z = asc.TQue(asc.TPosition.VECOUT, BUFFER_NUM)
    pipe.init_buffer(in_queue_x, BUFFER_NUM, tile_length * x.dtype.sizeof())
    pipe.init_buffer(in_queue_y, BUFFER_NUM, tile_length * y.dtype.sizeof())
    pipe.init_buffer(out_queue_z, BUFFER_NUM, tile_length * z.dtype.sizeof())
    for i in range(TILE_NUM * BUFFER_NUM):
        copy_in(i, x_gm, y_gm, in_queue_x, in_queue_y, tile_length)
        compute(z_gm, in_queue_x, in_queue_y, out_queue_z, tile_length)
        copy_out(i, z_gm, out_queue_z, tile_length)
```

scf Dialect:
表示控制流（如循环）

MLIR

```
func.func @vadd_kernel(%arg0: memref<?xf32, 22> {emitasc.kernel_arg = #emitasc<kernel_arg explicit>} loc
  ascendc.set_ffts_base_addr %arg4 : memref<?xui64, 22> loc(#loc)
  %c2_i32 = arith.constant 2 : i32 loc(#loc)
  %c512_i32 = arith.constant 512 : i32 loc(#loc)
  %c0_i32 = arith.constant 0 : i32 loc(#loc)
  %c16_i32 = arith.constant 16 : i32 loc(#loc)
  %c1_i32 = arith.constant 1 : i32 loc(#loc)
  %c128_i32 = arith.constant 128 : i32 loc(#loc)
  %0 = ascendc.get_block_idx : i32 loc(#loc3)
  %1 = arith.muli %0, %arg3 : i32 loc(#loc4)
  %2 = arith.index_cast %1 : i32 to index loc(#loc5)
  %3 = emitasc.ptr_offset %arg0[%2] : memref<?xf32, 22>, memref<?xf32, 22> loc(#loc5)
  %4 = ascendc.global_tensor : !ascendc.global_tensor<*xf32> loc(#loc5)
  ascendc.global_tensor.set_global_buffer %4, %3 : !ascendc.global_tensor<*xf32>, memref<?xf32, 22> loc(
  %5 = emitasc.ptr_offset %arg1[%2] : memref<?xf32, 22>, memref<?xf32, 22> loc(#loc6)
  %6 = ascendc.global_tensor : !ascendc.global_tensor<*xf32> loc(#loc6)
  ascendc.global_tensor.set_global_buffer %6, %5 : !ascendc.global_tensor<*xf32>, memref<?xf32, 22> loc(
  %7 = emitasc.ptr_offset %arg2[%2] : memref<?xf32, 22>, memref<?xf32, 22> loc(#loc7)
  %8 = ascendc.global_tensor : !ascendc.global_tensor<*xf32> loc(#loc7)
  ascendc.global_tensor.set_global_buffer %8, %7 : !ascendc.global_tensor<*xf32>, memref<?xf32, 22> loc(
  %9 = ascendc.pipe loc(#loc8)
  %10 = ascendc.queue : <vec_in, 2> loc(#loc9)
  %11 = ascendc.queue : <vec_in, 2> loc(#loc10)
  %12 = ascendc.queue : <vec_out, 2> loc(#loc11)
  ascendc.pipe.init_queue %9, %10, %c2_i32, %c512_i32 : !ascendc.queue<vec_in, 2>, i32, i32 loc(#loc12)
  ascendc.pipe.init_queue %9, %11, %c2_i32, %c512_i32 : !ascendc.queue<vec_in, 2>, i32, i32 loc(#loc13)
  ascendc.pipe.init_queue %9, %12, %c2_i32, %c512_i32 : !ascendc.queue<vec_out, 2>, i32, i32 loc(#loc14)
  scf.for %arg5 = %c0_i32 to %c16_i32 step %c1_i32  : i32 {
    %13 = ascendc.que_bind.alloc_tensor %10 : !ascendc.queue<vec_in, 2>, !ascendc.local_tensor<*xf32> lo
    %14 = ascendc.que_bind.alloc_tensor %11 : !ascendc.queue<vec_in, 2>, !ascendc.local_tensor<*xf32> lo
    %15 = arith.muli %arg5, %c128_i32 : i32 loc(#loc18)
    %16 = ascendc.global_tensor.subindex %4[%15] : !ascendc.global_tensor<*xf32>, i32, !ascendc.global_t
    ascendc.data_copy_l2 %13, %16, %c128_i32 : !ascendc.local_tensor<*xf32>, !ascendc.global_tensor<*xf3
    %17 = ascendc.global_tensor.subindex %6[%15] : !ascendc.global_tensor<*xf32>, i32, !ascendc.global_t
    ascendc.data_copy_l2 %14, %17, %c128_i32 : !ascendc.local_tensor<*xf32>, !ascendc.global_tensor<*xf3
    ascendc.que_bind.enque_tensor %10, %13 : !ascendc.queue<vec_in, 2>, !ascendc.local_tensor<*xf32> loc
    ascendc.que_bind.enque_tensor %11, %14 : !ascendc.queue<vec_in, 2>, !ascendc.local_tensor<*xf32> loc
```

arith Dialect:
表示常量定义和算术操作

Ascend C Dialect:
表示Ascend C API

PyAsc算子示例——PyAsc VS Ascend C

PyAsc生成的Ascend C代码

```
AscendC::TPipe v15;
AscendC::TQue<AscendC::TPosition::VECIN, 2> v16;
AscendC::TQue<AscendC::TPosition::VECIN, 2> v17;
AscendC::TQue<AscendC::TPosition::VECOUT, 2> v18;
v15.InitBuffer(v16, c2_i32, c512_i32);
v15.InitBuffer(v17, c2_i32, c512_i32);
v15.InitBuffer(v18, c2_i32, c512_i32);
for (int32_t v19 = c0_i32; v19 < c16_i32; v19 += c1_i32) {
  AscendC::LocalTensor<float> v20 = v16.AllocTensor<float>();
  AscendC::LocalTensor<float> v21 = v17.AllocTensor<float>();
  int32_t v22 = v19 * c128_i32;
  AscendC::GlobalTensor<float> v23 = v10[v22];
  AscendC::DataCopy(v20, v23, c128_i32);
  AscendC::GlobalTensor<float> v24 = v12[v22];
  AscendC::DataCopy(v21, v24, c128_i32);
  v16.EnQue(v20);
  v17.EnQue(v21);
  AscendC::LocalTensor<float> v25 = v16.DeQue<float>();
  AscendC::LocalTensor<float> v26 = v17.DeQue<float>();
  AscendC::LocalTensor<float> v27 = v18.AllocTensor<float>();
  AscendC::Add(v27, v25, v26, c128_i32);
  v18.EnQue(v27);
  v16.FreeTensor(v25);
  v17.FreeTensor(v26);
  AscendC::LocalTensor<float> v28 = v18.DeQue<float>();
  AscendC::GlobalTensor<float> v29 = v14[v22];
  AscendC::DataCopy(v29, v28, c128_i32);
  v18.FreeTensor(v28);
}
return;
}
```

原生的Ascend C代码

```
__aicore__ inline void Process()
{
    int32_t loopCount = TILE_NUM * BUFFER_NUM;
    for (int32_t i = 0; i < loopCount; i++) {
        CopyIn(i);
        Compute(i);
        CopyOut(i);
    }
}

__aicore__ inline void CopyIn(int32_t progress)
{
    AscendC::LocalTensor<half> xLocal = inQueueX.AllocTensor<half>();
    AscendC::LocalTensor<half> yLocal = inQueueY.AllocTensor<half>();
    AscendC::DataCopy(xLocal, xGm[progress * TILE_LENGTH], TILE_LENGTH);
    AscendC::DataCopy(yLocal, yGm[progress * TILE_LENGTH], TILE_LENGTH);
    inQueueX.EnQue(xLocal);
    inQueueY.EnQue(yLocal);
}
__aicore__ inline void Compute(int32_t progress)
{
    AscendC::LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    AscendC::LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    AscendC::LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    AscendC::Add(zLocal, xLocal, yLocal, TILE_LENGTH);
    outQueueZ.EnQue<half>(zLocal);
    inQueueX.FreeTensor(xLocal);
    inQueueY.FreeTensor(yLocal);
}
__aicore__ inline void CopyOut(int32_t progress)
{
    AscendC::LocalTensor<half> zLocal = outQueueZ.DeQue<half>();
    AscendC::DataCopy(zGm[progress * TILE_LENGTH], zLocal, TILE_LENGTH);
    outQueueZ.FreeTensor(zLocal);
}
```

目录

- Part 1 什么是PyAsc?
- Part 2 PyAsc技术原理和示例
- Part 3 PyAsc调试调优方法

PyAsc调试方法

为了方便开发者在**NPU上板**，**Model仿真**场景下进行算子功能调试，PyAsc当前提供了**printf、dump_tensor**接口供开发者调用。其中printf主要用于打印标量和字符串信息，dump_tensor用于打印指定Tensor的数据。

printf示例

调用代码示例

```
@asc.jit
def vadd_kernel(x: asc.GlobalAddress, y: asc.GlobalAddress, z: asc.GlobalAddress, block_length: int,
                tile_length: asc.ConstExpr[int]):
    offset = asc.get_block_idx() * block_length
    x_gm = asc.GlobalTensor()
    y_gm = asc.GlobalTensor()
    z_gm = asc.GlobalTensor()
    x_gm.set_global_buffer(x + offset)
    y_gm.set_global_buffer(y + offset)
    z_gm.set_global_buffer(z + offset)
    pipe = asc.TPipe()
    in_queue_x = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    in_queue_y = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    out_queue_z = asc.TQue(asc.TPosition.VECOUT, BUFFER_NUM)
    pipe.init_buffer(in_queue_x, BUFFER_NUM, tile_length * x.dtype.sizeof())
    pipe.init_buffer(in_queue_y, BUFFER_NUM, tile_length * y.dtype.sizeof())
    pipe.init_buffer(out_queue_z, BUFFER_NUM, tile_length * z.dtype.sizeof())
    # 使用printf打印字符串
    asc.printf("Start.\\n")
    for i in range(TILE_NUM * BUFFER_NUM):
        # 使用printf打印标量信息，当前循环次数
        asc.printf("Current index is %d.\\n", i)
        copy_in(i, x_gm, y_gm, in_queue_x, in_queue_y, tile_length)
        compute(z_gm, in_queue_x, in_queue_y, out_queue_z, tile_length)
        copy_out(i, z_gm, out_queue_z, tile_length)
```

使用printf打印字符串

使用printf打印标量信息，当前循环次数

执行结果日志

```
opType=v, DumpHead: AIV-7, CoreType=AIV, block dim=8, total_block_num=8,
CANN Version: 8.3.RC1, TimeStamp: 20250806001525499
Start.
Current index is 0.
Current index is 1.
Current index is 2.
Current index is 3.
Current index is 4.
Current index is 5.
Current index is 6.
Current index is 7.
Current index is 8.
Current index is 9.
Current index is 10.
Current index is 11.
Current index is 12.
Current index is 13.
Current index is 14.
Current index is 15.
```

dump_tensor示例

调用代码示例

```
@asc.jit
def vadd_kernel(x: asc.GlobalAddress, y: asc.GlobalAddress, z: asc.GlobalAddress, block_length: int,
                tile_length: asc.ConstExpr[int]):
    offset = asc.get_block_idx() * block_length
    x_gm = asc.GlobalTensor()
    y_gm = asc.GlobalTensor()
    z_gm = asc.GlobalTensor()
    x_gm.set_global_buffer(x + offset)
    y_gm.set_global_buffer(y + offset)
    z_gm.set_global_buffer(z + offset)
    pipe = asc.TPipe()
    in_queue_x = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    in_queue_y = asc.TQue(asc.TPosition.VECIN, BUFFER_NUM)
    out_queue_z = asc.TQue(asc.TPosition.VECOUT, BUFFER_NUM)
    pipe.init_buffer(in_queue_x, BUFFER_NUM, tile_length * x.dtype.sizeof())
    pipe.init_buffer(in_queue_y, BUFFER_NUM, tile_length * y.dtype.sizeof())
    pipe.init_buffer(out_queue_z, BUFFER_NUM, tile_length * z.dtype.sizeof())
    # 使用dump_tensor打印Global Memory的tensor
    asc.dump_tensor(x_gm, 0, 32)
    for i in range(TILE_NUM * BUFFER_NUM):
        copy_in(i, x_gm, y_gm, in_queue_x, in_queue_y, tile_length)
        compute(z_gm, in_queue_x, in_queue_y, out_queue_z, tile_length)
        copy_out(i, z_gm, out_queue_z, tile_length)
```

使用dump_tensor打印Global Memory的tensor数据

执行结果日志

```
opType=v, DumpHead: AIV-0, CoreType=AIV, block dim=8, total_block_num=8, block_remain_len=1048288,
block_initial_space=1048576, rsv=0, magic=5aa5bccd
CANN Version: 8.3.RC1, TimeStamp: 20250806001525499
DumpTensor: desc=0, addr=112f5400, data_type=float32, position=GM, dump_size=32
[0.447473, 0.994458, 0.992582, 0.382010, 0.953588, 0.827148, 0.639453, 0.837255, 0.766262, 0.975809
, 0.386220, 0.077225, 0.454802, 0.317456, 0.487141, 0.919555, 0.965576, 0.675870, 0.635633, 0.28582
5, 0.015032, 0.389543, 0.454321, 0.230220, 0.794838, 0.166961, 0.479756, 0.153436, 0.347653, 0.9740
92, 0.668703,
0.423896]
```

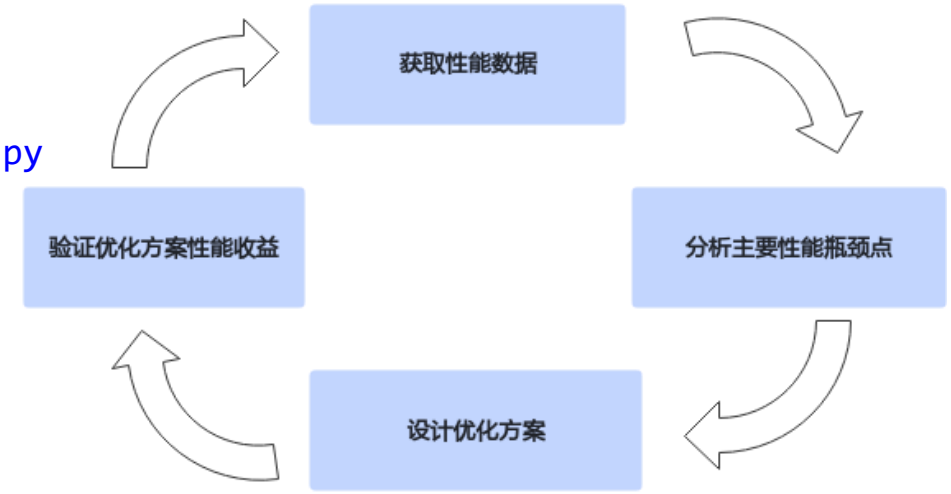
PyAsc调优方法

为了帮助开发者快速完成高性能算子开发，PyAsc支持算子性能调优工具**msprof**，用于采集和分析运行在昇腾AI处理器上的AI任务各个运行阶段的关键性能指标，用户可根据输出的性能数据，快速定位软、硬件性能瓶颈，提升AI任务性能分析的效率。

- Step1：msprof --help；确认当前环境是否支持msprof，并查看相关命令选项。
- Step2：完成上板性能数据采集，以Add算子为例，在代码仓根目录执行命令如下：

  ```
  msprof --output=./output python3 ./python/tutorials/01_add/add.py
  ```

- Step3：分析性能数据，设计优化方案。
- Step4：重复Step2和Step3，持续优化算子性能直至满足目标。



K	L	M	N	O	P	Q	R	S	T	U	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	AU
Task Wait T	Block Dim	Mix Block D	HF32 Eligib	Input Shap	Input Data	Input Form	Output Sha	Output Dat	Output For	Context ID	aiv_time(us	aiv_total_cy	aiv_vec_tim	aiv_vec_rati	aiv_scalar_t	aiv_scalar_r	aiv_mte2_ti	aiv_mte2_ra	aiv_mte3_ti	aiv_mte3_ra	aiv_icache_	cube_utilization(%)	
0	8	0	YES	N/A	N/A	N/A	N/A	N/A	N/A	N/A	8.25	122094	0.35	0.042	5.658	0.686	3.369	0.408	1.888	0.229	0.019	0	

Step2命令执行后，生成的op_summary_*.csv性能数据结果

CANN

msprof工具资料请参考：https://www.hiascend.com/document/detail/zh/canncommercial/83RC1/devaids/Profiling/atlasprofiling_16_0001.html

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