北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

基于Triton的大模型通用算子库FlagGems技术分享

北京智源人工智能研究院

2024年6月15日

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

开 场 及 活 动 介 绍

李涛 | 智源AI系统研究组研发负责人

2024-06-15

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

活动议程

01 FlagGems介绍和开发指南 —李之昕 20min

02 FlagGems创新实践：运行时优化 — 李之昕 15min

03 FlagGems创新实践：自动代码生成 — 陈飞宇 20min

04 自由讨论环节 — 30min

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

FlagGems介 绍 和 开 发 指 南

李之昕 | 智源编译器研发工程师

2024-06-15

FlagGems算子库

https://github.com/FlagOpen/FlagGems

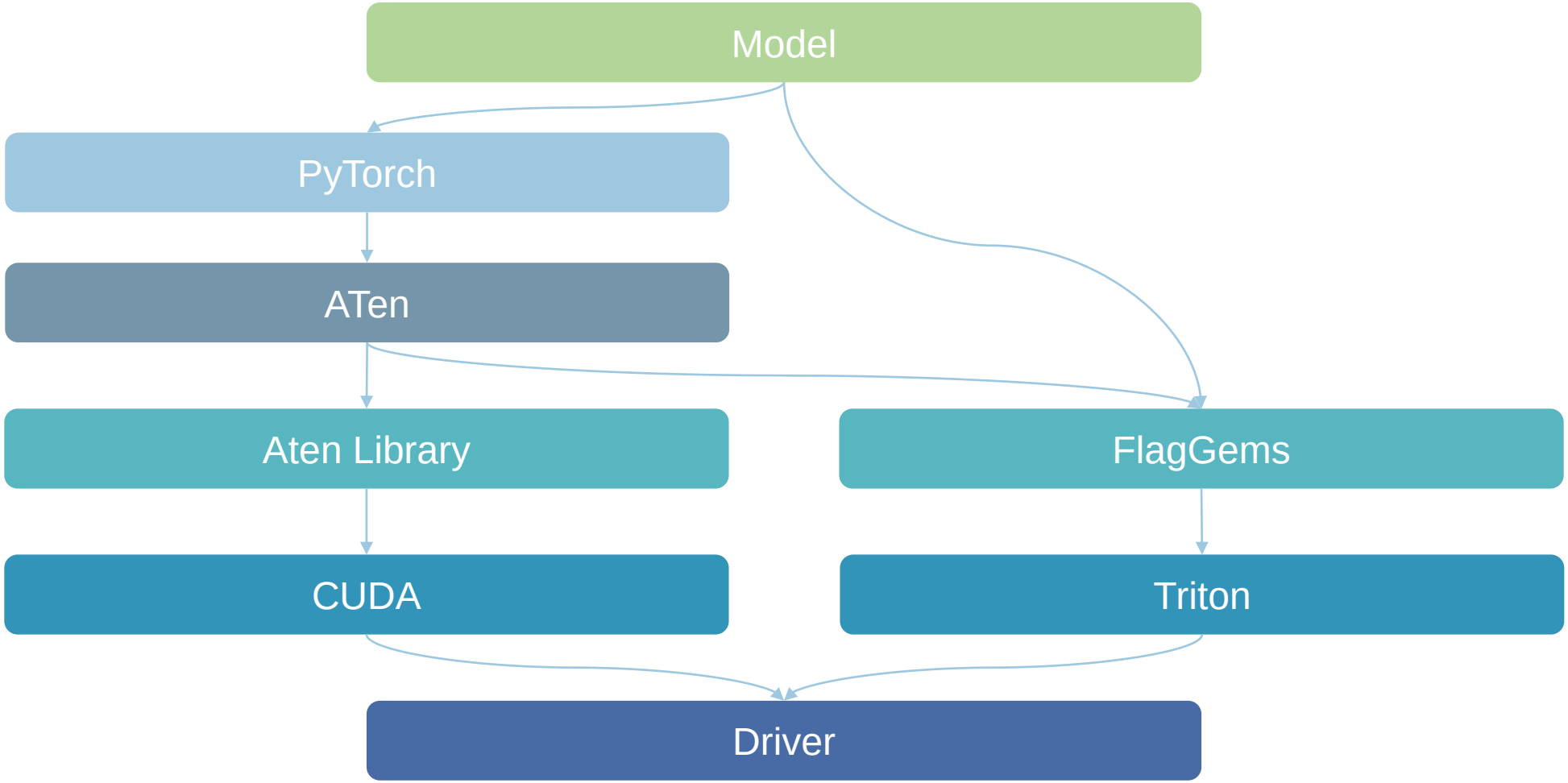
一个使用**OpenAI Triton**语言实现的算子库

高性能
通用算子性能与torch eager持平
融合算子性能优于torch eager

广覆盖
V2.0共实现66个算子，替换76个Aten接口
预计V3.0完成133+算子，基本覆盖大模型需求

轻量级
多款硬件支持，涵盖NVIDIA与国产芯片
安装使用简便，无缝嵌入PyTorch
开发优化效率高，快速更新迭代

FlagGems算子库



FlagGems算子库

- 在进程中永久启用

```python
import torch
import flag_gems
flag_gems.enable()

M, N, K = 1024, 1024, 1024
A = torch.randn((M, K), dtype=torch.float16, device="cuda")
B = torch.randn((K, N), dtype=torch.float16, device="cuda")
C = torch.mm(A, B)
```

- 在进程局部临时启用

```python
import torch
import flag_gems

M, N, K = 1024, 1024, 1024
A = torch.randn((M, K), dtype=torch.float16, device="cuda")
B = torch.randn((K, N), dtype=torch.float16, device="cuda")
with flag_gems.use_gems():
    C = torch.mm(A, B)
```

- 直接调用

```python
import torch
import flag_gems

M, N, K = 1024, 1024, 1024
A = torch.randn((M, K), dtype=torch.float16, device="cuda")
B = torch.randn((K, N), dtype=torch.float16, device="cuda")
C = flag_gems.mm(A, B)
```

FlagGems算子清单

LinearAlgebra

BLAS	Reduction	
addmm	cumsum	amax
bmm	mean	argmax
linear*	sum	max
matmul**	prod	min
mm	var_mean	
mv		
outer		

NeuralNetwork

BasicMath

Fusion

Backward

* linear底层调用addmm实现

** matmul底层调用bmm, mm, mv实现

FlagGems算子清单

LinearAlgebra

NeuralNetwork

Activation	Normalization	Other
gelu	layer_norm*	dropout***
relu	group_norm**	triu
silu	vector_norm	cross_entropy_loss
softmax	rms_norm	native_dropout
log_softmax	native_layer_norm	
sigmoid	native_group_norm	

BasicMath

Fusion

Backward

* layer_norm底层调用native_layer_norm实现

** group_norm底层调用native_group_norm实现

*** dropout底层调用native_dropout实现

FlagGems算子清单

LinearAlgebra

NeuralNetwork

BasicMath

Mathmatical		Logical		Reduction
add	abs	bitwise_and	eq	all
div	exp	bitwise_not	ge	any
sub	reciprocal	bitwise_or	gt	
mul	rsqrt	__or__	le	
pow	neg	isinf	lt	
rsub	cos	isnan	ne	
clamp	sin			
where	tanh			

Fusion

Backward

FlagGems算子清单

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

- LinearAlgebra
- NeuralNetwork
- BasicMath
- Fusion

Mathmatical	Pointwise	Attention
skip_layernorm	gelu_and_mul	rotary_position_embedding
skip_rmsnorm	silu_and_mul	

- Backward

FlagGems算子清单

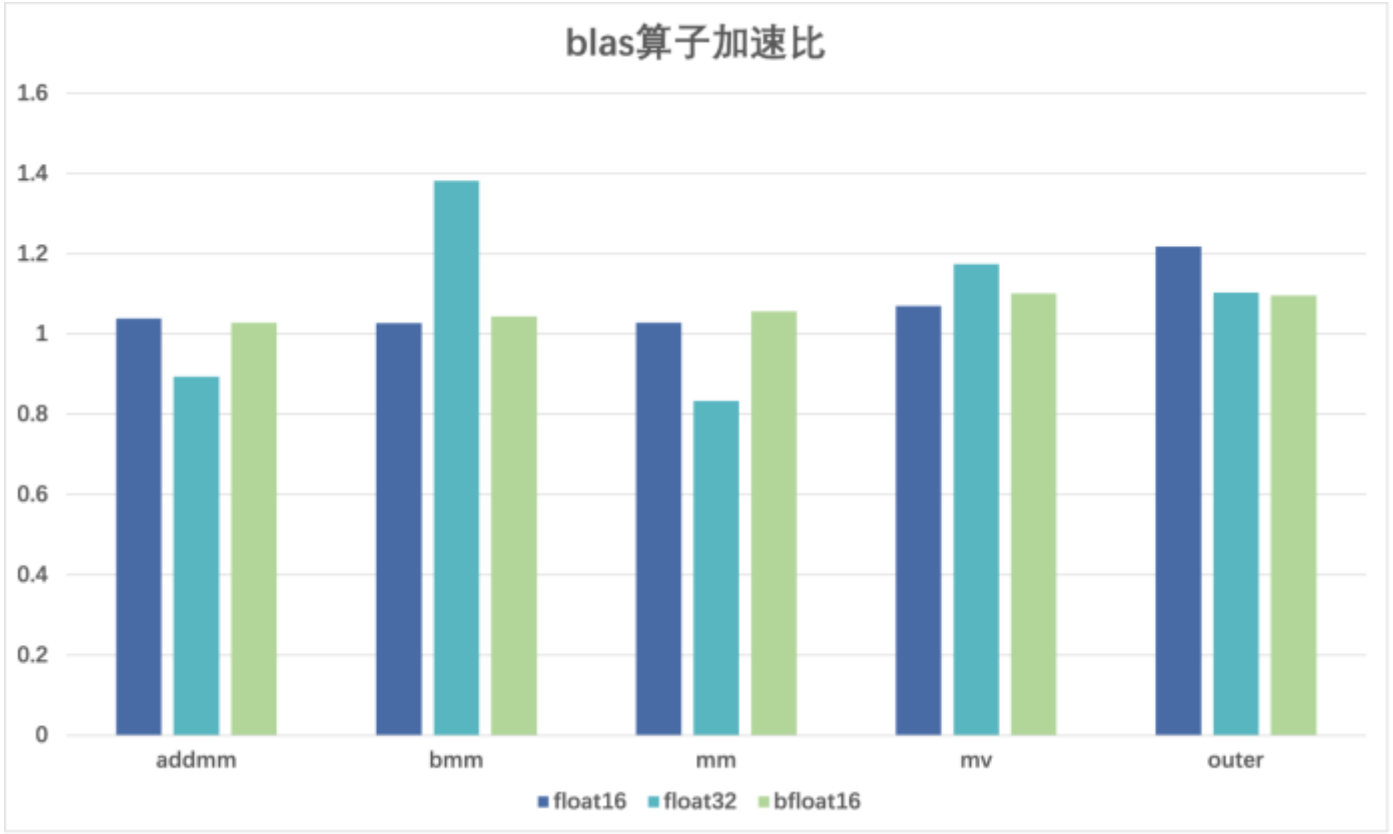
- LinearAlgebra
- NeuralNetwork
- BasicMath
- Fusion
- Backward*

Pointwise	Reduction	Other
native_dropout_backward	native_layer_norm_backward	outer_backward
threshold_backward	_softmax_backward_data	cross_entropy_loss_backward
silu_backward	_log_softmax_backward_data	
tanh_backward	native_group_norm_backward	
sigmoid_backward		

* Backward算子未计入总数

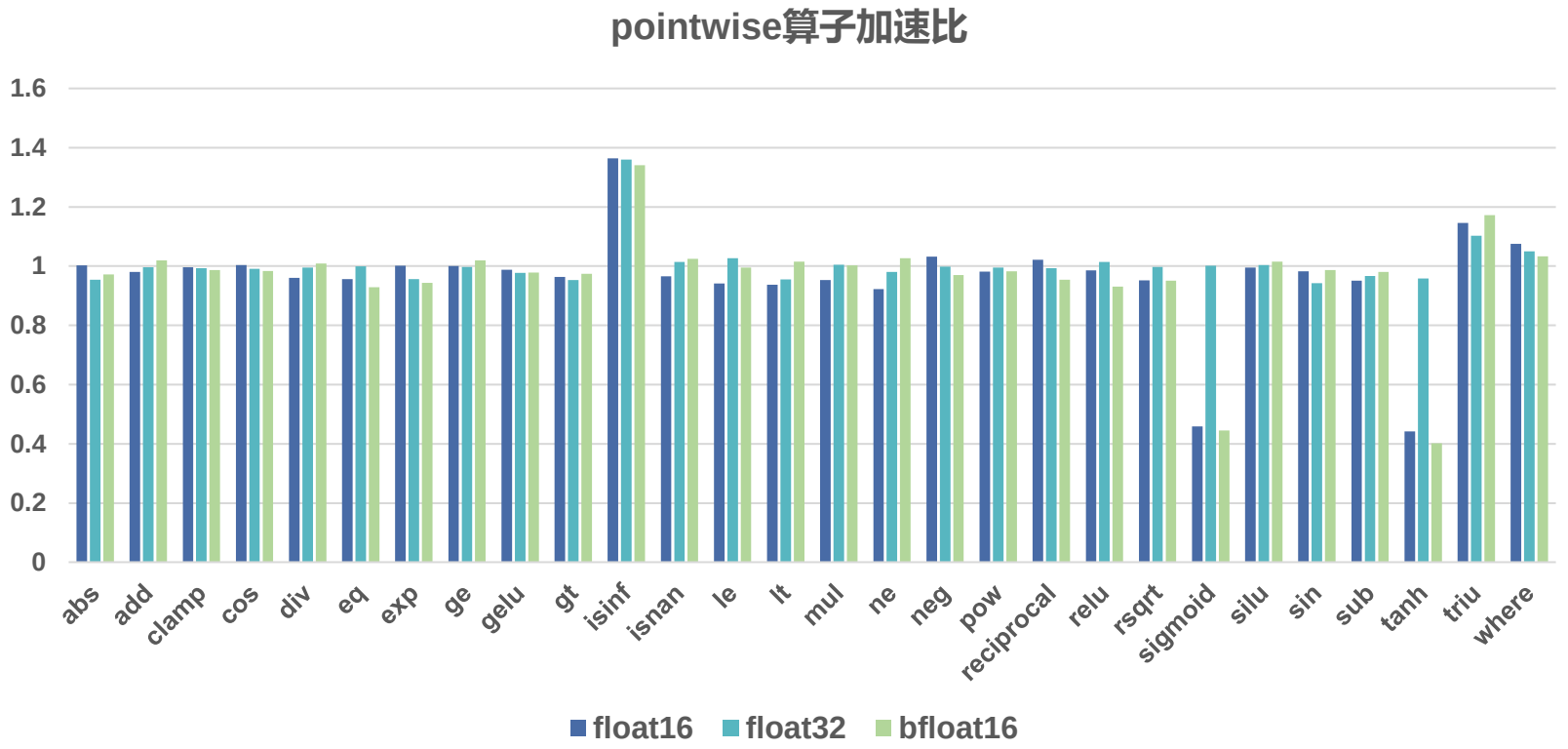
FlagGems与Aten Kernel性能对比

- 计算密集型：blas线性代数计算
  - 单精度计算性能接近cuda
  - 半精度计算性能追平cuda



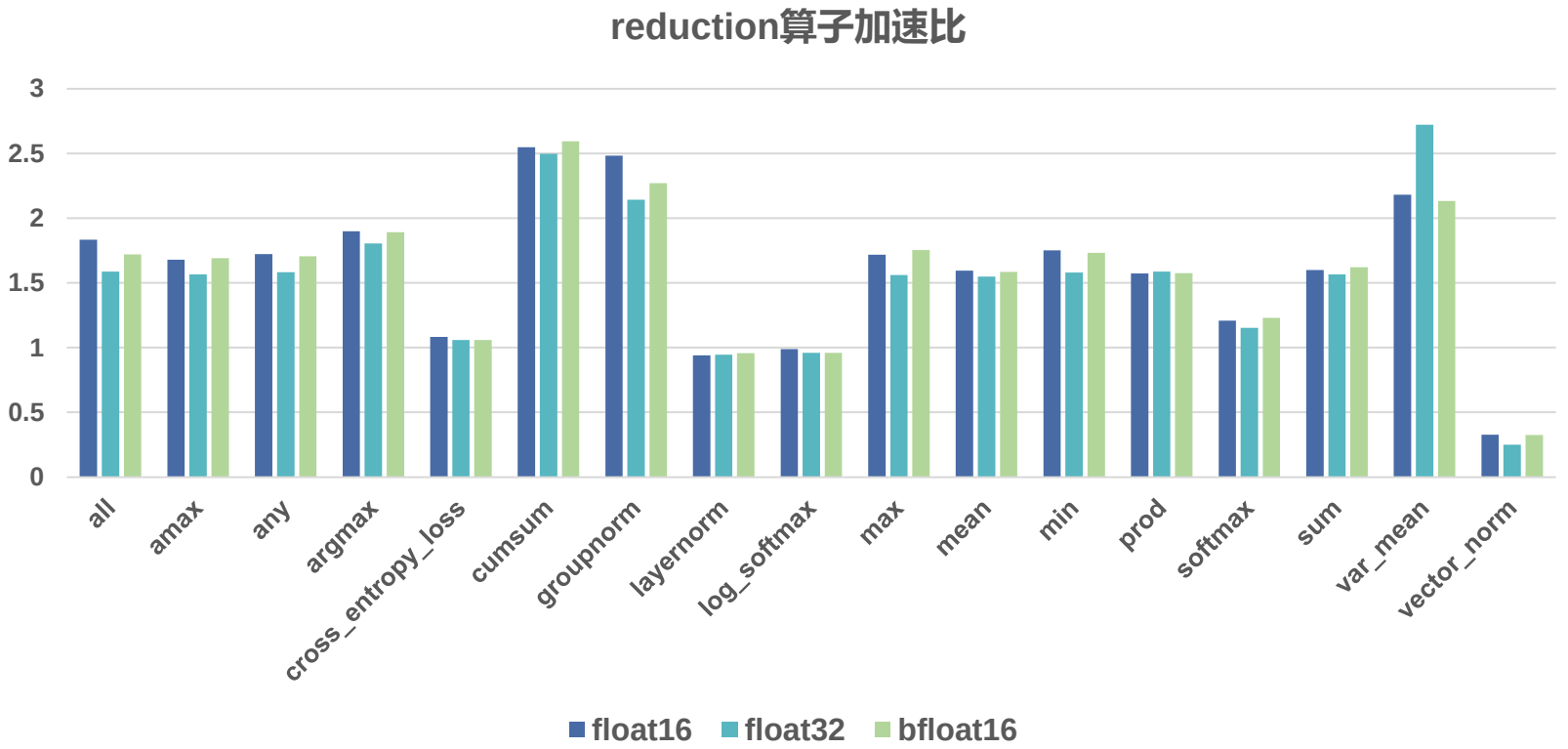
FlagGems与Aten Kernel性能对比

- 访存密集型：pointwise对位计算
  - 多数算子各种精度计算性能基本追平cuda
  - 少数算子受triton language功能接口的数据类型限制，upcast会增大半精度的时间成本



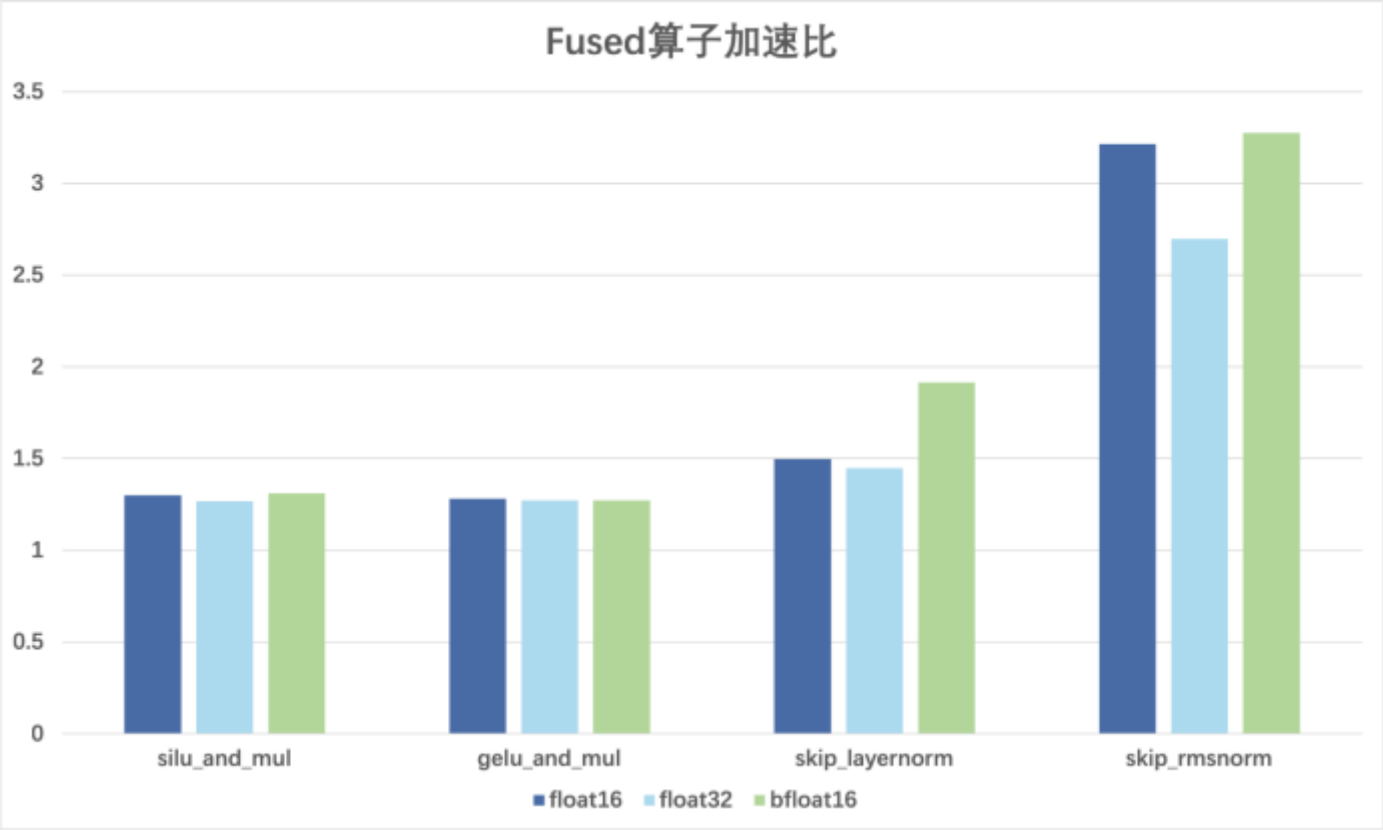
FlagGems与Aten Kernel性能对比

- 访存密集型：reduction归约计算
  - 多数算子各精度计算性能超过cuda
  - 少数算子计算性能劣于cuda，仍需进一步优化



FlagGems与Aten Kernel性能对比

- 融合算子
  - 全部算子、各种精度计算性能超过cuda



FlagGems生态

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

联合开发

BAAI 智源研究院

中科加禾 CORE SIGMA

SILICONFLOW

支持平台

NVIDIA

Cambricon 寒武纪

天数智芯 Iluvatar CoreX

METAX 沐曦

HYGON

摩尔线程 MOORE THREADS

昆仑芯 KUNLUNXIN

支持模型

Google Bert Infer

Google

Meta Llama2 Infer

BAAI Aquila Train

悟道·天鹰 WUDAO Aquila

FlagGems开发

算子实现（仅前向）

FlagGems/src/flag_gems/

自动生成方法

仅限pointwise算子

装饰pointwise_dynamic

打印DEBUG日志记录调用

外层包装与aten接口对齐

```
import triton
import triton.language as tl
import logging
from ..utils import pointwise_dynamic

@pointwise_dynamic
@triton.jit
def cos_func(x):
    return tl.cos(x.to(tl.float32))

def cos(A):
    logging.debug("GEMS COS")
    O = cos_func(A)
    return O
```

手写方法

无法自动生成的算子

装饰libentry

打印DEBUG日志记录调用

外层包装与aten接口对齐

```
import triton
import triton.language as tl
import logging
from ..utils import libentry

@libentry
@triton.jit
def cos_func(…):
    …

def cos(A):
    logging.debug("GEMS COS")
    O = cos_func(A)
    return O
```

初始化导入 __init__.py

FlagGems开发

算子实现（前反向）

FlagGems/src/flag_gems/

编写方法

核函数编写方法不变

可装饰libentry或pointwise_dynamic

创建算子类，继承torch.autograd.Function

定义静态方法forward和backward

使用ctx保存反向所需变量

前向有几个输入，反向就要有几个输出

非张量输入对应的反向输出为None

外层包装与aten对齐

```python
import torch
import triton
import triton.language as tl
import logging
from ..utils import pointwise_dynamic

@pointwise_dynamic
@triton.jit
def tanh_forward(x):
return tl.math.tanh(x.to(tl.float32))

@pointwise_dynamic
@triton.jit
def tanh_backward(y, dy):
return dy * (1.0 - tl.math.pow(y.to(tl.float32), 2))
```

```python
class Tanh(torch.autograd.Function):
@staticmethod
def forward(ctx, A):
logging.debug("GEMS TANH FORWARD")
O = tanh_forward(A.to(torch.float32))
ctx.save_for_backward(O)
return O.to(A.dtype)

@staticmethod
def backward(ctx, out_grad):
logging.debug("GEMS TANH BACKWARD")
(out,) = ctx.saved_tensors
in_grad = tanh_backward(out, out_grad)
return in_grad

def tanh(A):
return Tanh.apply(A)
```

初始化导入 __init__.py

FlagGems开发

算子注册

FlagGems/src/flag_gems/__init__.py

- 仅替换ATen的单算子需要注册，ATen未包含的算子、融合算子无需注册
- 使用torch.library系列接口实现
- 在PyTorch/aten/src/ATen/native/native_functions.yaml中查看aten算子接口标准
- 只替换前向
  - lib.impl("cos", **cos**, "CUDA")
- 替换前向与反向
  - lib.impl("tanh", **tanh**, "AutogradCUDA")

FlagGems开发

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

算子精度验证

FlagGems/tests/

分析算子类别

- unary_pointwise
- binary_pointwise*
- reduction
- blas
- special

获取测试参数

- POINTWISE_SHAPES
- REDUCTION_SHAPES
- MNK_SHAPES
- FLOAT_DTYPES
- INT_DTYPES
- SCALARS

选择精度标准

- gems_assert_close
- gems_assert_equal
- others

判断转换需求

- to_reference
- to_cpu**
- upcast

编写测试代码

```
@pytest.mark.parametrize("shape", POINTWISE_SHAPES)
@pytest.mark.parametrize("dtype", FLOAT_DTYPES)
def test_accuracy_cos(shape, dtype):
    inp = torch.randn(shape, dtype=dtype, device="cuda")
    ref_inp = to_reference(inp, True)

    ref_out = torch.cos(ref_inp)
    with flag_gems.use_gems():
    res_out = torch.cos(inp)

    gems_assert_close(res_out, ref_out, dtype)
```

* 三元及以上的pointwise算子也放在test_binary_pointwise_ops.py中

** 部分国产芯片不支持float64，需要把参考值在CPU host端计算，在运行时增加参数 --device cpu

FlagGems开发

算子精度验证

原则：在保证算法正确的前提下，可以允许假阳（False Positive），不能出现假阴（False Negative）

- 与torch.float64的eager模式计算结果比较
- torch.allclose标准

$$|input - other| \leq atol + rtol \times |other|$$

- Machine epsilon
  - 根据IEEE标准，由于浮点数的近似规则，无法识别小于machine epsilon的相对差异

$$|x - x'| \leq |x| \times (1 + \varepsilon)$$

data type	fraction	epsilon	decimal resolution
float16	10	$2^{-20} \approx 0.000976562$	0.001
float32	23	$2^{-23} \approx 1.19209e-07$	1.3e-06
bfloat16	7	$2^{-7} \approx 0.0078125$	0.01

- Reduction
  - 随着计算次数增加，可能引入更大的误差，tolerance也应该相应增长
  - 输入数据存在负数，会抵消计算结果的绝对值，使rtol的实际容忍范围缩小
  - 以归约规模为系数，增大atol取值

FlagGems开发

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

算子性能测试

FlagGems/benchmark/

分析算子类别

- pointwise
- reduction
- blas

获取测试参数

- POINTWISE_BATCH
- REDUCTION_BATCH
- BLAS_BATCH
- SIZES
- INT_DTYPES

配置参数生成

- unary_arg
- unary_int_arg
- binary_args
- binary_int_args
- ternary_args
- other

判断评测需求

- cuda_perf*
- end2end_perf**

编写测试代码

```
@pytest.mark.parametrize("dtype", FLOAT_DTYPES)
def test_perf_cos(dtype):
    bench = Benchmark(
        op_name="cos",
        torch_op=torch.cos,
        arg_func=unary_arg,
        dtype=dtype,
        batch=POINTWISE_BATCH,
        sizes=SIZES,
    )
    bench.run()
```

* CUDA性能使用triton.testing.do_bench测试

** end2end性能使用python time.time测试，在运行时增加参数 --mode cpu

FlagGems开发

模型测试

- torchbench标准
  - 与torch eager模式的计算结果对比
  - 若torch.allclose(atol=1e-3, rtol=1e-3)通过，则认为计算正确
  - 若cosine_similarity不低于0.99，则认为计算正确
- 测试四组prompt
  - "How are you today?”
  - "What is your name?”
  - "Who are you?”
  - "Where are you from?"

谢谢聆听！

欢迎访问开源社区：

https://github.com/FlagOpen/FlagGems

https://github.com/FlagOpen/FlagAttention

Triton中国社区微信群

扫码添加微信，私信“加群”

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

活动议程

01 FlagGems介绍和开发指南 —李之昕 20min

02 **FlagGems创新实践：运行时优化 — 李之昕 15min**

03 FlagGems创新实践：自动代码生成 — 陈飞宇 20min

04 自由讨论环节 — 30min

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

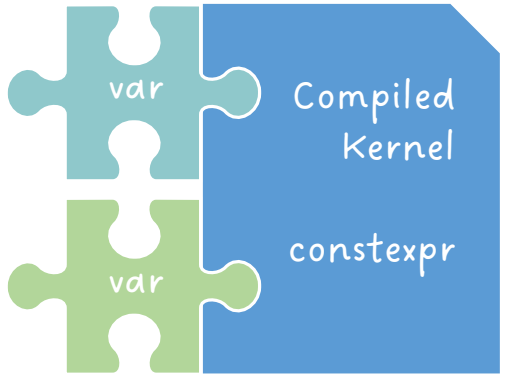
FlagGems创新实践：运行时优化

李之昕 | 智源编译器研发工程师

2024-06-15

Triton 运行时

- tl.constexpr
  - 在kernel中特化的固定参数，编译时已知
  - 通常需要在函数声明中以annotation形式声明
  - 可以是：常数 / 常数列表 / 常数元组 / 表达式
  - 可以参与：基本数学运算 / 比较 / 逻辑运算 / 位运算



  - 无需声明也会成为被特化的变量
    - 张量数据类型
    - num_warps / num_ctas / num_stages

```
@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
```

Triton 运行时

Config
kernel的可选配置参数，包括constexpr和num_warps / num_ctas / num_stages

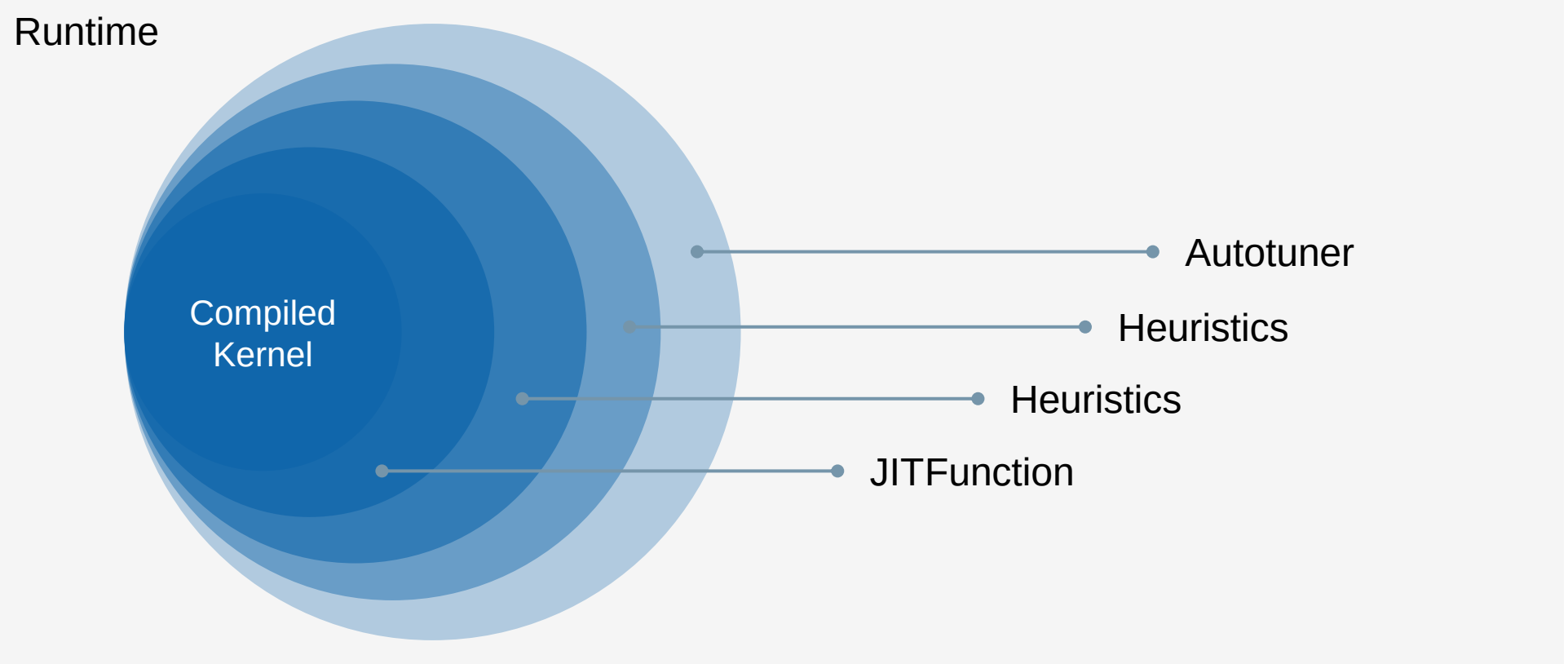
Autotuner (optional)
使用自动调优方法选择kernel的配置参数Config

Heuristics (optional)
使用启发式方法计算kernel配置参数constexpr

JITFunction (required)
执行kernel的即时编译

三种方式均可指定编译配置参数：Autotuner / Heuristics / 运行时传参

Triton 运行时




```python
@triton.autotune(configs=[triton.Config(kwargs={'BLOCK_SIZE': 32})], key=['N'], warmup=1, rep=1)
@triton.heuristics({'EVEN_N': lambda nargs: nargs['N'] % 2 == 0})  # test kwargs
@triton.heuristics({'EVEN_src': lambda nargs: nargs['src'].data_ptr() % 2 == 0})  # test args
@triton.jit
def _kernel(dst, src, N, BLOCK_SIZE: tl.constexpr, EVEN_N: tl.constexpr, EVEN_src: tl.constexpr):
    tl.store(dst, EVEN_N)
    tl.store(dst + 1, EVEN_src)
```

Triton 运行时

- Autotuner
  - 通过编译、运行和测定，来选择耗时最小的配置与kernel
  - 可以自定义剪枝策略在调优前缩小配置空间

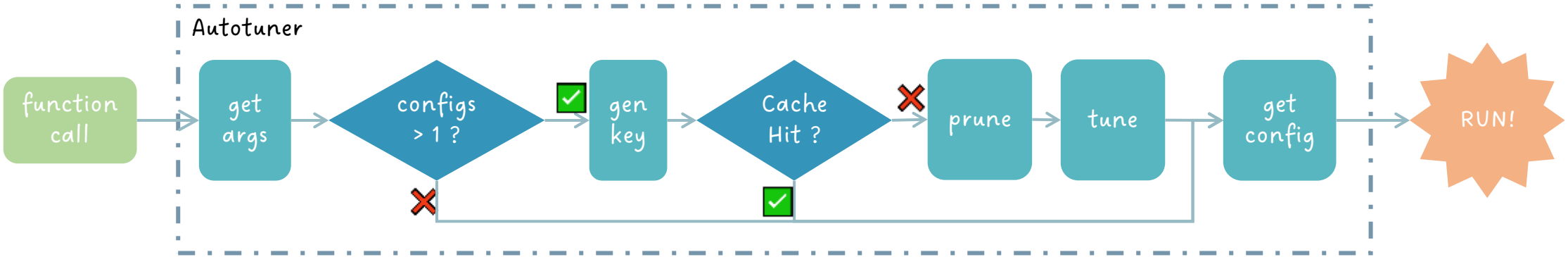
  - 参数
    - 可选的常量配置空间：list of Config
    - 用于选择配置的变量键值：list of runtime arguments
    - 测定kernel所需的循环配置：warmup, rep
    - 剪枝可用的策略模型：perf_model, top_k, early_config_prune

  - 默认参数
    - num_warps=4, num_stages=2, num_ctas=1
    - num_ctas只面向SM 90以上的架构

```
@triton.autotune(
    configs=[
        triton.Config({
            'BLOCK_SIZE_M': 128,
            'BLOCK_SIZE_N': 128,
            'BLOCK_SIZE_K': 32,
            'NUM_SM': 84,
        }),
        triton.Config({
            'BLOCK_SIZE_M': 128,
            'BLOCK_SIZE_N': 128,
            'BLOCK_SIZE_K': 32,
            'NUM_SM': 128,
        }),
        triton.Config({
            'BLOCK_SIZE_M': 64,
            'BLOCK_SIZE_N': 64,
            'BLOCK_SIZE_K': 32,
            'NUM_SM': 84,
        }),
        triton.Config({
            'BLOCK_SIZE_M': 64,
            'BLOCK_SIZE_N': 64,
            'BLOCK_SIZE_K': 32,
            'NUM_SM': 128,
        }),
    ],
    key=['group_size'],
)
```

Triton 运行时

- Autotuner
  - 配置缓存config cache
    - Key：键值及其数据类型
    - Value：调优获得的最佳配置参数
  - 跳过条件
    - 相同键值可读取缓存，无需重新调参
    - 参数空间中Config配置唯一，无需调参



Triton 运行时

- Heuristics
  - 给定所需配置参数，以及启发式的计算表达式
  - 可使用运行时传入参数进行计算
  - 当heuristics装饰在autotune内层，也可以使用调优所得参数进行计算

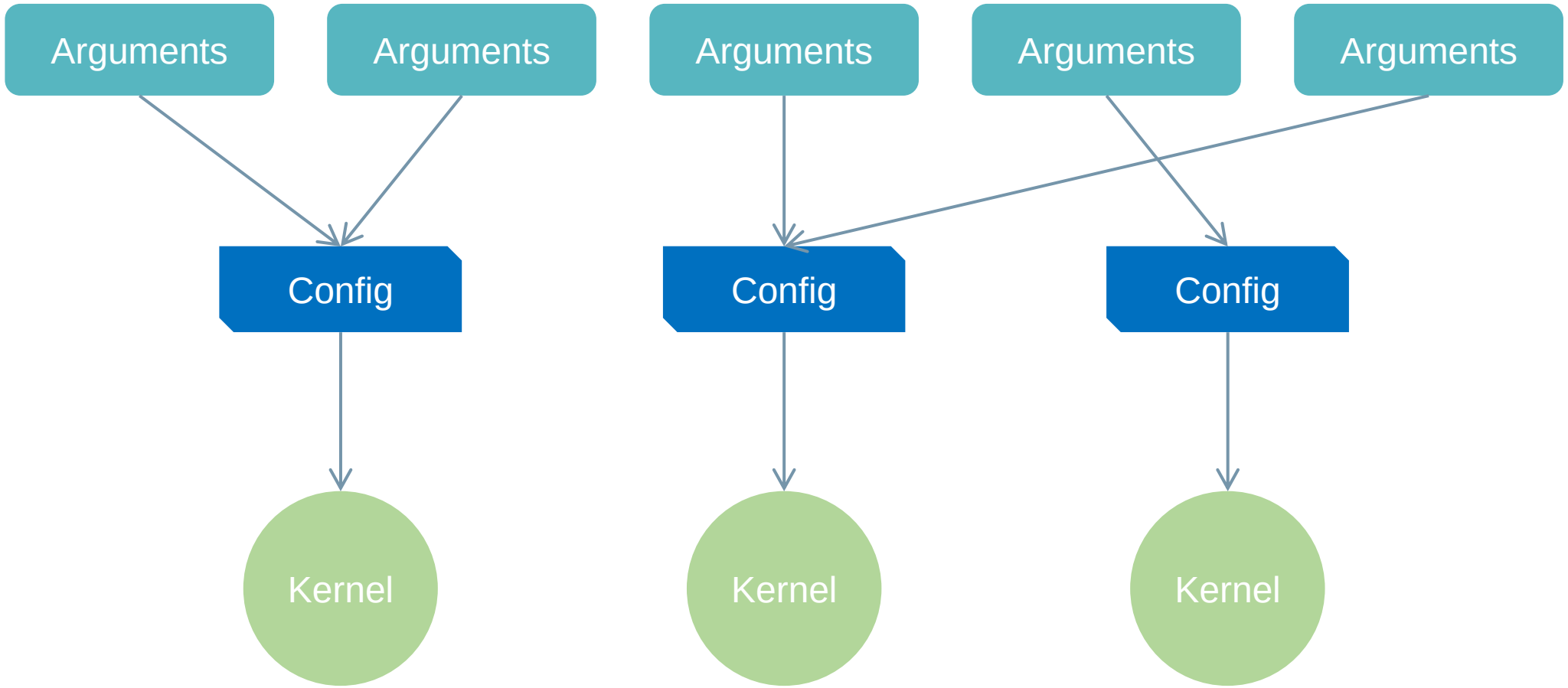
  - values为字典形式，key为参数名，value为可调用的表达式
  - 在调参开销过大的场景下，可以使用启发式方法迅速获得参数

```
@triton.heuristics(
    values={
        "BLOCK_N": lambda args: triton.next_power_of_2(args["N"]),
        "num_warps": lambda args: (
            4 if args["N"] <= 1024 else (8 if args["N"] <= 2048 else 16)
        ),
    },
)
```

Triton 运行时

- JITFunction
  - 维护kernel cache：缓存已编译的kernel
  - 键值构造：调用inspect绑定参数
    - 参数名
    - 常量参数值、特化参数值
    - 非常量参数值及其annotation数据类型
  - v2键值（在v3得到优化）
    - Cuda版本号
    - 特化参数：张量地址对齐，标量数值对齐
  - 缓存键值可直接取出kernel运行
  - 新键值需编译kernel，缓存后运行
  - 其他
    - 参数包装、设备检查、stream检查

Triton 运行时

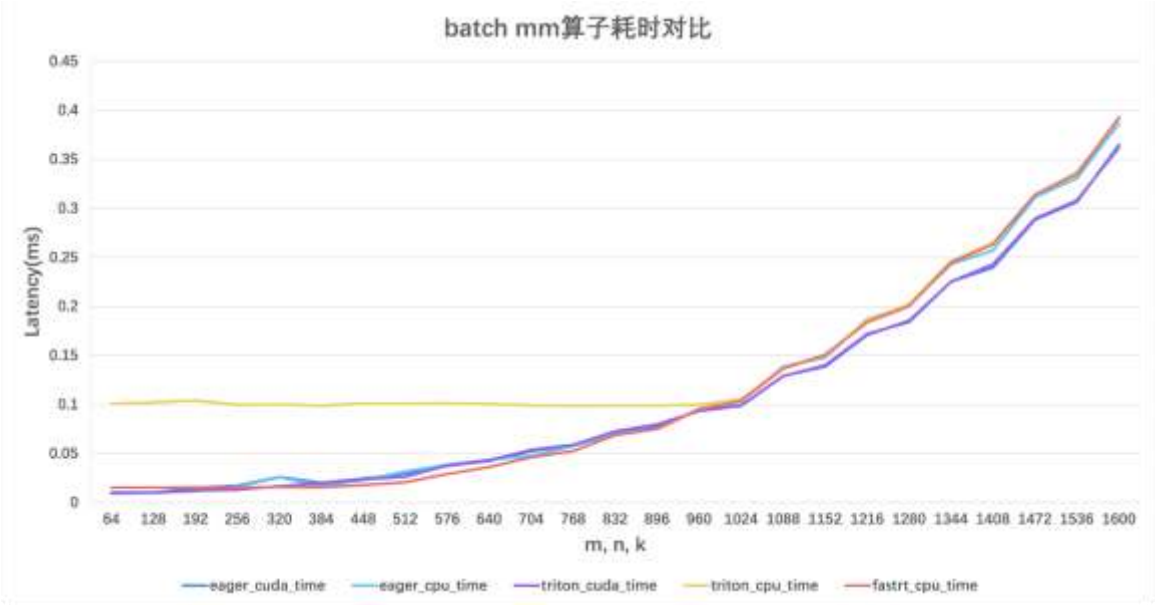

Arguments
Arguments
Arguments
Arguments
Arguments
Config
Config
Config
Kernel
Kernel
Kernel

FlagGems运行时机制

Triton v2 Runtime开销

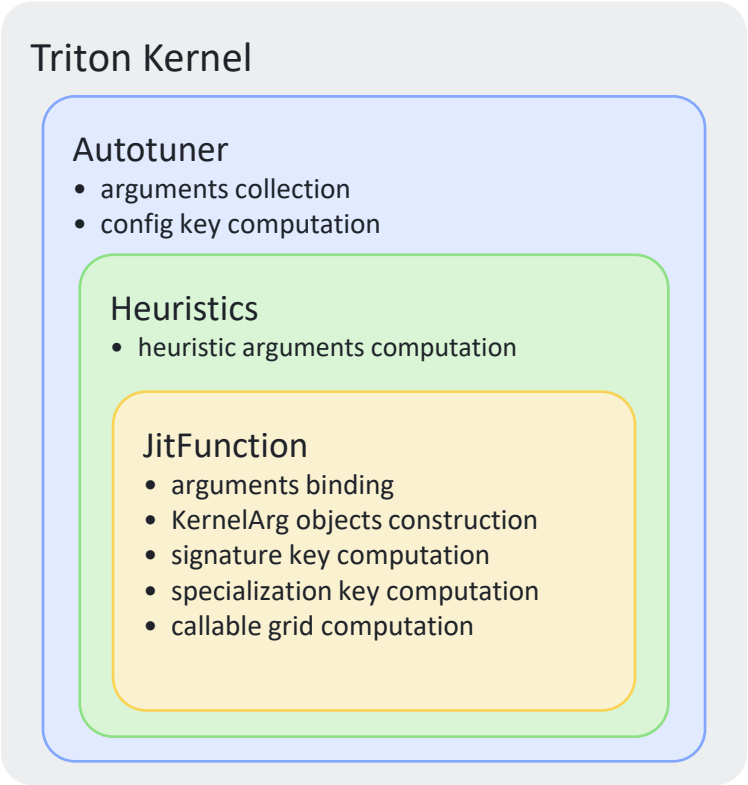
- 以batch matmul算子为例
  - A100平台
  - 取batch=10，dtype=torch.float16
  - 充分warmup和repeat
- 优化的triton kernel可以达到与eager kernel持平的性能
- 而triton的CPU的端到端开销远高于torch eager
  - 大规模算子开销接近
  - 小规模算子上差距更加明显
- fastrt：kernel外置，运行时只计算最简cache key

triton kernel的运行时开销是性能差距的主要原因



FlagGems运行时机制

- 运行时开销开源
  - Autotuner
  - Heuristics
  - JitFunction
- 调参之后的每次执行都会调用所有层次装饰器的run函数
- JitFunction.run是主要瓶颈
  - 参数绑定调用python inspect
  - KernelArg类型包装
  - cache key计算

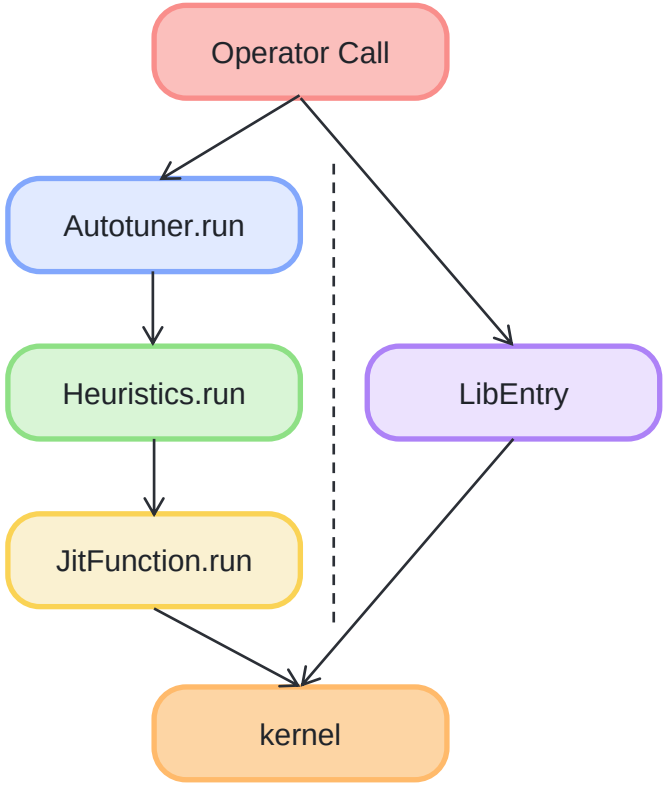



```
@triton.autotune(
    configs=configs(),
    key=["M", "N", "K"]
)
@triton.heuristics(
    {
        "DIVISIBLE_M": lambda args: args["M"] % args["TILE_M"] == 0,
        "DIVISIBLE_N": lambda args: args["N"] % args["TILE_N"] == 0,
        "DIVISIBLE_K": lambda args: args["K"] % args["TILE_K"] == 0,
    }
)
@triton.jit
def bmm_kernel(*args):
    ...
```

FlagGems运行时机制

- 策略
  - 构造LibEntry独立维护kernel cache
  - 绕过Autotuner、Heuristics和JitFunction的runtime
- 功能
  - 仅需在triton kernel前装饰即可
  - 支持Autotuner、Heuristics、JitFunction的直接包装
  - 首次执行时进入原调用路径，不影响调参功能的正常使用
- 优化
  - 无需经过运行时类型的嵌套调用
  - 节省了重复的参数处理，无需绑定和类型包装
  - 简化了cache key格式，减少不必要的键值计算

```
1  @libentry()
2  @triton.autotune(...)
3  @triton.heuristics(...)
4  @triton.jit
5  def bmm_kernel(*args):
6          ...
```

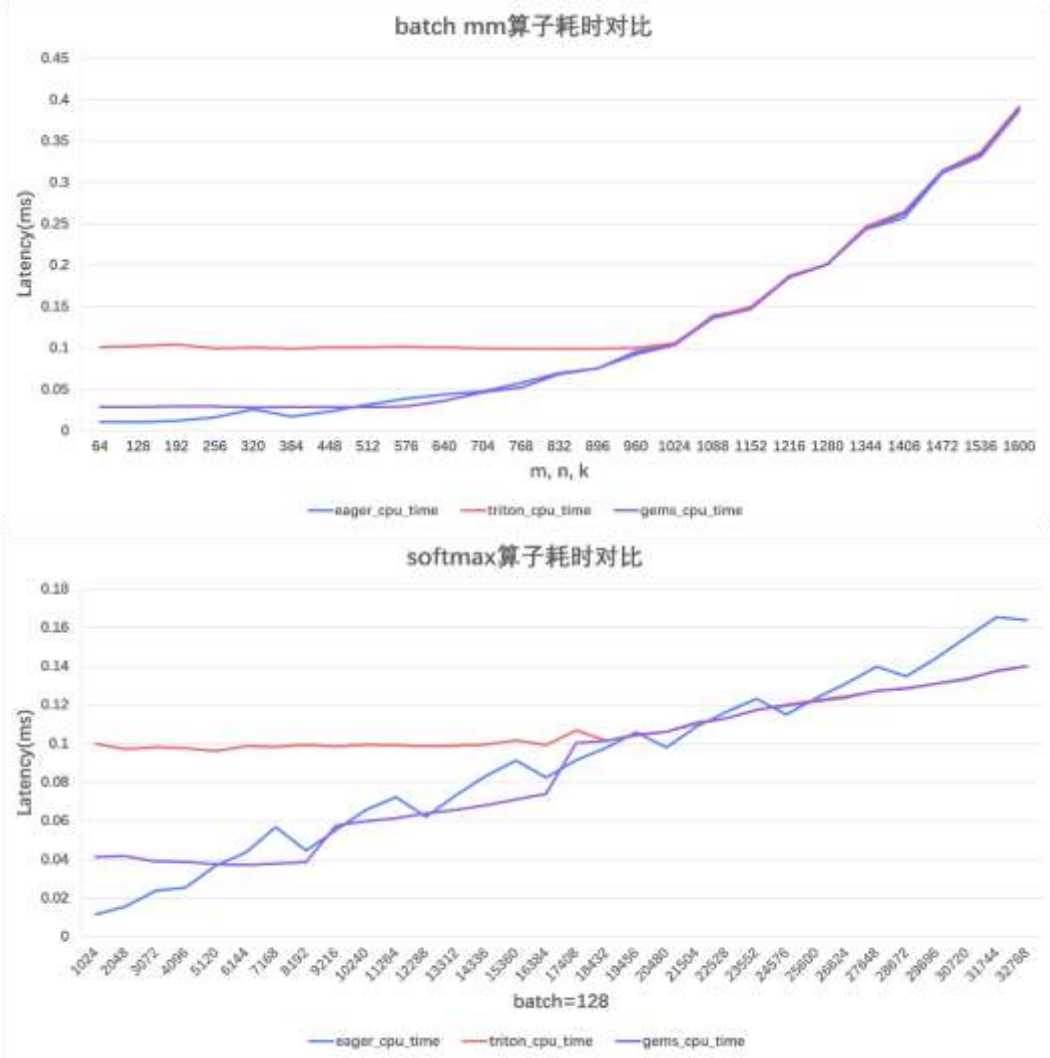


FlagGems运行时机制

- 效果
  - cpu端到端显著低于未优化triton，节省近70%
  - runtime耗时无法完全消除，只能逼近torch eager
- 局限
  - 必需键值比fast rt实验场景更复杂
  - 相比原cache key，存在一定的信息损失
  - 独立的kernel cache额外占用了存储空间
  - 只在小规模算子上效果明显

    大规模算子的cuda发射周期短于kernel执行周期

    瓶颈不在runtime



谢谢聆听！

欢迎访问开源社区：

https://github.com/FlagOpen/FlagGems

https://github.com/FlagOpen/FlagAttention

Triton中国社区微信群

扫码添加微信，私信“加群”

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

活动议程

01 FlagGems介绍和开发指南 —李之昕 20min

02 FlagGems创新实践：运行时优化 — 李之昕 15min

03 **FlagGems创新实践：自动代码生成 — 陈飞宇 20min**

04 自由讨论环节 — 30min

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

FlagGems创新实践：自动代码生成

陈飞宇 | 智源编译器研发工程师

2024-06-15

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Pointwise 算子自动代码生成

01 为什么使用 Code Generation

02 核心：PointwiseDynamicFunction

03 实现机制

04 性能分析

05 总结与展望

pointwise 算子

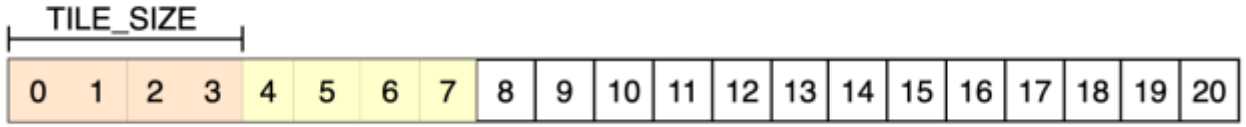
如何用 triton 写一个功能较全的 pointwise 算子。从最简单的 vector add 开始。

```
@triton.jit
def add_kernel(a_ptr, b_ptr, o_ptr, N, TILE_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offsets = pid * TILE_SIZE + tl.arange(0, TILE_SIZE)
    mask = offsets < N

    a = tl.load(a_ptr + offsets, mask=mask)
    b = tl.load(b_ptr + offsets, mask=mask)
    o = a + b

    tl.store(o_ptr + offsets, o, mask=mask)

def add(a, b):
    o = torch.empty_like(a)
    N = a.numel()
    TILE_SIZE = 512
    grid = triton.cdiv(N, TILE_SIZE)
    add_kernel[grid](a, b, o, N, TILE_SIZE)
    return o
```



上述代码的问题：

- 当 tensor 形状不同，需要 broadcast 是无法处理；
- tensor 数据不连续，broadcast 或，或者 permute 过，或者数据是分散的情况不能处理。

原因来自其加载和保存部分的地址偏移量的计算，就是预设计算首地址后连续的一篇内存区域。

解法一: 使用 wrapper 处理 non-contiguous


```
def add_func_naive(x, y):
    # supports broadcasting now
    x, y = torch.broadcast_tensors(x, y)

    # support non-contiguous now
    x = x.contiguous()
    y = y.contiguous()
    o = torch.empty(x.shape, dtype=x.dtype, device=x.device)


    TILE_SIZE = 512
    num_warps = 4
    N = x.numel()
    grid = triton.cdiv(N, TILE_SIZE), 1, 1
    add_kernel[grid](x, y, o, N, TILE_SIZE, num_waprs=num_warps)
    return o
```

使用 wrapper 调用 torch.broadcast_tensor 来处理 broadcasting, 使用 contiguous 来处理数据不连续的问题。

使用 warpper 处理 broadcast 和非连续的输入 tensor, 性能不一定好，这个过程本身就可能有一次数据 copy.

解法二：支持 stride

假设 a, b 来自两个大的 tensor 上的不连续切片。

```
a = torch.randn(200)[::2]
b = torch.randn(300)[::3]
```

修改其内存偏移量计算方式即可，需要传入三个 tensor 各自的 stride.

```
a = tl.load(a_ptr + offsets * stride_a, mask=mask)
b = tl.load(b_ptr + offsets * stride_b, mask=mask)

tl.store(o_ptr + offsets * stride_o, o, mask=mask)
```

解法二：支持 stride

假设 a, b 来自两个 2d tensor 上的不连续切片。

```
a = torch.randn(200，200)[::2, ::2]
b = torch.randn(300, 300)[::3, ::3]
```

```
mask = tid < (M * N)
i1 = tid % N
i0 = tid // N

a = tl.load(a_ptr + i0 * a_stride0 + i1 * a_stride1, mask=mask)
b = tl.load(b_ptr + i0 * b_stride0 + i1 * b_stride1, mask=mask)
o = a + b

tl.store(o_ptr + i0 * o_stride0 + i1 * o_stride1, o, mask=mask)
```

数组不连续，必须考虑每个维度的 stride, 因此需要计算每个维度的 index。（i0, i1)

有了各个维度的索引，以及 tensor 每个维度的 stride 就可以求出偏移量。

Q: 怎么把每个 tensor 的 stride 传给 JITFunction 呢?
A: Triton JITFunction 不接受 list tuple 等类型的参数，只能展开作为多个 int 来传。而 tensor 的 rank 在编写 jit function 的时候是不确定的。无法写出一份适用所有情况的代码。

```
TypeError: Unsupported type <class 'tuple'> for (128, 128)
```

计算内存偏移量的代码模式化，所有的 pointwise 运算原理上具有相似性，可以采用**自动代码生成**的方式。

Triton JITFunction 不支持 Tuple 参数

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

```python
@triton.jit
def add_kernel(a, o, iteration_space, a_strides, o_strides, BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr):
    # NOPE, you cannot do this!
    pid0, pid1 = tl.program_id(0), tl.program_id(1)
    m_offsets = pid0 * BLOCK_M + tl.arange(0, BLOCK_M)
    n_offsets = pid1 * BLOCK_N + tl.arange(0, BLOCK_N)
    a_s1, a_s2 = a_strides
    o_s1, o_s2 = o_strides
    s1, s2 = iteration_space
    a_ptrs = a + m_offsets[:, None] * a_s1 + n_offsets * a_s2
    o_ptrs = o + m_offsets[:, None] * o_s1 + n_offsets * o_s2
    mask = (m_offsets[:, None] < s1) & (n_offsets < s2)
    a_block = tl.load(a_ptrs, mask=mask)
    v = tl.sin(a_block)
    tl.store(o_ptrs, v, mask=mask)
```

```
Traceback (most recent call last):
  File "/home/clement/projects/triton_snippets/index/try_to_use_tuple.py", line 24, in <module>
    add_kernel[(4, 4, 1)](x, y, (128, 128), (128, 1), (128, 1), 32, 32)
  File "/home/clement/.virtualenvs/triton_dev/lib/python3.10/site-packages/triton/runtime/jit.py", line 434, in run
    sig_key = tuple(arg.signature_key() for arg in args if not arg.param.is_constexpr)
  File "/home/clement/.virtualenvs/triton_dev/lib/python3.10/site-packages/triton/runtime/jit.py", line 434, in <genexpr>
    sig_key = tuple(arg.signature_key() for arg in args if not arg.param.is_constexpr)
  File "/home/clement/.virtualenvs/triton_dev/lib/python3.10/site-packages/triton/runtime/jit.py", line 168, in signature_key
    return JITFunction._key_of(self.value)
  File "/home/clement/.virtualenvs/triton_dev/lib/python3.10/site-packages/triton/runtime/jit.py", line 229, in _key_of
    raise TypeError(f"Unsupported type {type(arg)} for {arg}")
TypeError: Unsupported type <class 'tuple'> for (128, 128)
```

内存偏移量计算方式

(program id -> task id) -> multi index -> memory offset

triton 编程里各个 CTA 的差异是从 CTA id (program id) 开始的，类似 cuda 的 blockIdx 和 threadIdx.

Pointwise Operation 总是可以视为若干个独立标量运算, 这些 task 可以**展平**视为 1d 的 vector. 每个 task 有自己的 task id. 每个 CTA 计算 TILE_SIZE 个 task. (**1d task space**)

```
tid = pid * TILE_SIZE + tl.arange(0, TILE_SIZE)
```

例:
task space: (5, 5)
tile size: 4

task id

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

program id

0	0	0	0	1
1	1	1	2	2
2	2	3	3	3
3	4	4	4	4
5	5	5	5	6

内存偏移量计算方式

program id -> (task id -> multi index) -> memory offset

通过取余和整除，将 task id 对应于每个 tensor 上的数据的多维索引计算出来。
Q: 为什么要使用 multi-index?
A: 每个维度的 stride 都是独立的，不预设 tensor 内存连续。所以 task id 无法直接对应到内存偏移量。

```
i1 = tid % N
i0 = tid // N
```

task id

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

index 0

0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4

index 1

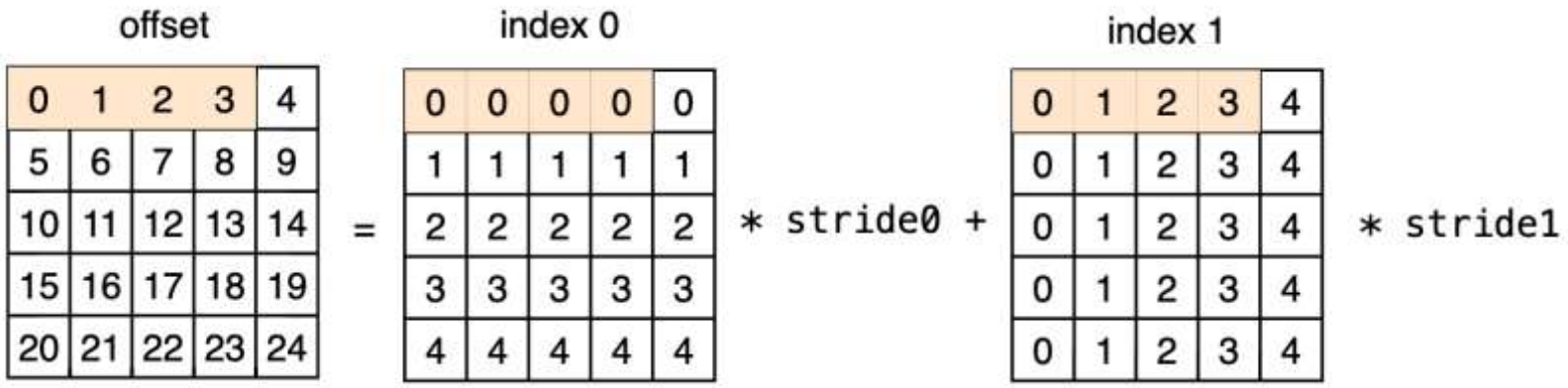
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4

内存偏移量计算方式

program id -> task id -> (multi index -> memory offset)

每个 tensor 在每个维度都有一个 stride，表示该维度索引增加1带来的内存偏移。多维索引和 strides 的内积就是内存偏移量。

```
a = tl.load(a_ptr + i0 * a_stride0 + i1 * a_stride1, mask=mask)
b = tl.load(b_ptr + i0 * b_stride0 + i1 * b_stride1, mask=mask)
tl.store(o_ptr + i0 * o_stride0 + i1 * o_stride1, o, mask=mask)
```



北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Pointwise 算子自动代码生成

01 为什么使用 Code Generation

02 核心：PointwiseDynamicFunction

03 实现机制

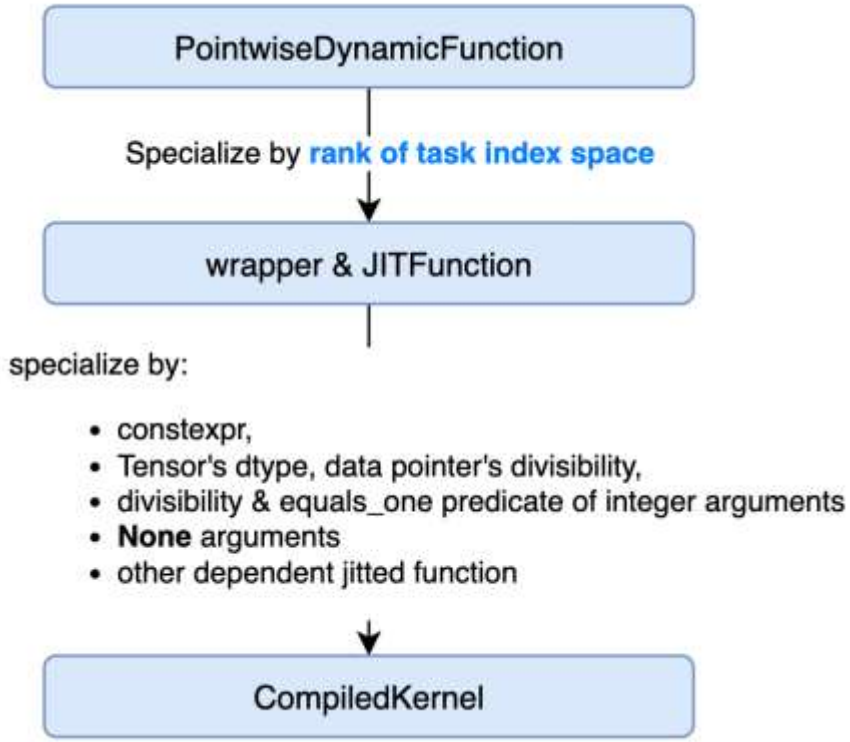
04 性能分析

05 总结与展望

核心：PointwiseDynamicFunction

给定需要执行的标量运算，自动生成能适用于 tensor 的 pointwise operation.

- 根据输入的 argument 来确定**任务索引空间的 rank**（各个 tensor broadcast 后的形状）；
- 根据 rank 按需生成特化版本的函数。
- 转发参数，调用适用的特化版函数。



一个 **PointwiseDynamicFunction** 可能对应若干个 wrapper 和 JITFunction. 此处有一层特化，由 PointwiseDynamicFunction 维护。

pointwise_dynamic 实现为一个装饰器，装饰在一个 JITFunction 上。得到一个 PointwiseDynamicFunction。

一个 **JITFunction** 可能对应若干个 CompiledKernel, 此处是另外一层特化，这是 triton 运行时维护的。

PointwiseDynamic 的思路类似。

PointwiseDynamicFunction 使用方式

最基本的用法，有多个 tensor 输入，有一个输出，输出的类型和第一个 operand 一致。

```
@pointwise_dynamic
@triton.jit
def ordinary2(x, y):
   return tl.sin(x) + tl.cos(y)
```

```
@pointwise_dynamic()
@triton.jit
def ordinary(x, y):
   return tl.sin(x) + tl.cos(y)
```

可以指定某些参数不是 tensor，而是**作为普通标量传到标量函数**。还可以指定这些参数的数据类型。

```
@pointwise_dynamic(
   is_tensor=[True, False, True],
   dtypes=[None, float, None])
@triton.jit
def axpy(x, alpha, y):
   return x * alpha + y
```

```
@pointwise_dynamic(is_tensor=[True, False, True])
@triton.jit
def axpy(x, alpha, y):
   return x * alpha + y
```

PointwiseDynamicFunction 使用方式

支持指定每个输出 tensor 的数据类型，比如 compare 类输出 torch.bool

```
@pointwise_dynamic(output_dtypes=[torch.bool])
@triton.jit
def ge(x, y):
    return x > y
```

支持指定输出个数（而不是从函数体推断）

```
@pointwise_dynamic(num_inputs=2, num_outputs=2)
@triton.jit
def f(a, b):
    return a + b, a - b
```

支持更复杂的函数体，不只是一行的函数。比如 FlagGems 中的 gelu 函数

```
@pointwise_dynamic
@triton.jit
def gelu_none(x):
    scale = 0.7071067811
    output = 0.5 * x * (1 + tl.math.erf(x * scale))
    return output
```

PointwiseDynamicFunction 使用方式

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

手写 pointwise kernel 和 wrapper

```
@triton.jit
def abs_kernel(
    X,
    Y,
    M,
    M_BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(0) * M_BLOCK_SIZE
    Y_ptrs = tl.make_block_ptr(
        Y,
        shape=(M,),
        strides=(1,),
        offsets=(pid,),
        block_shape=(M_BLOCK_SIZE,),
        order=(0,),
    )
    X_ptrs = tl.make_block_ptr(
        X,
        shape=(M,),
        strides=(1,),
        offsets=(pid,),
        block_shape=(M_BLOCK_SIZE,),
        order=(0,),
    )
    X_val = tl.load(X_ptrs)
    Y_val = tl.abs(X_val)
    tl.store(Y_ptrs, Y_val.to(X_val.dtype))


def abs(A):
    A = A.contiguous()
    O = torch.empty_like(A)
    M = A.numel()
    grid_fn = lambda meta: (triton.cdiv(M, meta["M_BLOCK_SIZE"]),)
    abs_kernel[grid_fn](A, O, M)
    return O
```

使用 pointwise dynamic 实现

```
@pointwise_dynamic
@triton.jit
def abs_func(x):
    return tl.abs(x)


def abs(A):
    O = abs_func(A)
    return O
```

缩减代码量，提高开发效率。

目前 FlagGems 中的 31 个 pointwise 算子使用 pointwise_dynamic 实现。

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Pointwise 算子自动代码生成

实现机制

索引：关于如何生成索引部分的代码；

函数嵌入: 如何把 scalar function 嵌入生成的代码中；

动态生成函数: 如何动态编译生成的代码；

代码缓存: 如何缓存生成的函数。

生成代码示例

PointwiseDynamic 生成的代码有两个部分，wrapper 和 jit function.

wrapper function 分两个：

- wapper：主要负责生成 output 并调用 warpper_out；
- warpper_out: 参数准备，计算索引空间，调用 jit function;

```
def _wrapper(in0: torch.Tensor, val0: float, in1: torch.Tensor):
    """Generated wrapper function with Pointwise: (Tensor, float, Tensor) -> (Tensor)"""
    shape = broadcast_shapes([in0.shape, in1.shape])
    out0 = torch.empty(shape, dtype=in0.dtype, device=in0.device)
    out0 = _wrapper_out(in0, val0, in1, out0)
    return out0
```

生成代码示例


```
def _wrapper_out(in0: torch.Tensor, val0: float, in1: torch.Tensor, out0: torch.Tensor):
    """Generated wrapper function with Pointwise: (Tensor, float, Tensor, Tensor) -> (Tensor)"""
    shape = broadcast_shapes([in0.shape, in1.shape])
    num_tasks = volume(shape)

    # strides of each tensor argument w.r.t the task space
    in0_strides = broadcasted_stride(in0.shape, in0.stride(), shape)
    in1_strides = broadcasted_stride(in1.shape, in1.stride(), shape)
    out0_strides = out0.stride()

    # kernel launch
    grid = triton.cdiv(num_tasks, 512), 1, 1
    _jit_function[grid](
        in0, val0, in1, out0,
        in0_strides[0], in0_strides[1], # stride for in0
        in1_strides[0], in1_strides[1], # stride for in1
        out0_strides[0], out0_strides[1], # stride for out0
        shape[0], shape[1], # task indexing space
        num_tasks, # num tasks
        tile_size=512,
        num_warps=4,
    )
    return out0
```

并非所有代码都自动生成，我们写了少量计算形状的代码，在生成的代码中使用。

生成代码示例


```
@triton.jit(do_not_specialize=['val0'])
def _jit_function(
    in0_ptr: tl.tensor, # of tl.pointer_type
    val0: float,
    in1_ptr: tl.tensor, # of tl.pointer_type
    out0_ptr: tl.tensor, # of tl.pointer_type
    in0_stride0: int, in0_stride1: int, # strides for in0
    in1_stride0: int, in1_stride1: int, # strides for in1
    out0_stride0: int, out0_stride1: int, # strides for out0
    s0: int, s1: int, # task_space
    num_tasks: int,
    tile_size: tl.constexpr,
):
    # task id & masking
    pid = tl.program_id(0)
    tid = pid * tile_size + tl.arange(0, tile_size)
    mask = tid < num_tasks

    # multi index recontruction
    i1 = tid % s1
    tid //= s1
    i0 = tid % s0

    # loads
    in0 = tl.load(in0_ptr + i0 * in0_stride0 + i1 * in0_stride1, mask=mask)
    in1 = tl.load(in1_ptr + i0 * in1_stride0 + i1 * in1_stride1, mask=mask)

    # compute
    out0 = in0 * val0 + in1

    # stores
    tl.store(out0_ptr + i0 * out0_stride0 + i1 * out0_stride1, out0, mask=mask)
```

动态生成函数

Python 中动态执行代码的方式：

- eval/exec 动态执行生成的代码字符串。或者先 compile 再 exec. 这样可以执行一些依赖当前 scope 的代码(比如依赖 exec 时 local 或 global scope 中的变量）。
- 写入到文件，然后再 compile exec.
- 写入到文件，用 importlib 来导入，这样要求模块是不依赖当前的 scope.

Triton JITFunction 本身使用了 **inspect** 库来获取函数源码，而如果没有写入到文件，单纯 exec 字符串得到的函数是没有源码信息的，会导致 triton 编译流程出错。Pointwise dynamic 的方案是写入到文件，然后在用 importlib 来导入。代码独立，也方便 debug 时直接修改生成的代码进行测试。

```
with open(cache_dir() / file_name, "wt", encoding="utf-8") as f:
        f.write(code.getvalue())

spec = importlib.util.spec_from_file_location(
   f"_gen_module_{self._scalar_fn_cache_key}", f.name)
m = importlib.util.module_from_spec(spec)
spec.loader.exec_module(m)
overload = getattr(m, "_wrapper")
```

Inline

被装饰的函数要求是一个 JITFunction. 因此其内部可以调用 tl 内的函数，而不是写 torch的函数。(这主要是为了避免去做代码翻译带来的不准确)

Inline 发生在 Python AST 层面。使用 **ast.Visitor** 来改写 AST, 然后再使用 unparse 重新生成 python 代码，嵌入 jit function 代码中。（这主要是因为 triton JITFunction 不支持传入 triton JITFunction 作为参数)

- scalar_function 中的输入形参将会被传入的实参的名字替换；
- scalar_function 中的定义的 local 变量名字会避免与当前命名空间中已经使用的名字冲突；
- scalar_function 返回表达式将会改为赋值表达式，赋值给输出变量；

```
@pointwise_dynamic(is_tensor=[True, False, True])
@triton.jit
def axpy(x, alpha, y):
   return x * alpha + y
```

```
# loads
in0 = tl.load(in0_ptr + ...)
in1 = tl.load(in1_ptr + ...)

# compute
out0 = in0 * val0 + in1

# stores
tl.store(out0_ptr +..., out0, mask=mask)
```

缓存

PointwiseDynamicFunction 的缓存：以 rank 为 key 的字典，运行时维护。

```
def __call__(self, *args):
    # It does not accept kwargs
    key = f"{self.arg_key(*args)}"
    if key in self.overloads:
        overload = self.overloads[key]
    else:
        m = ... # dynamically generated module
        overload = getattr(m, "_wrapper")
        self.overloads[key] = overload
    return overload(*args)
```

生成文件：对于 pointwise_dynamic 装饰得到的函数，影响生成的文件(wrapper & jit function) 的因素有 scalar_fn 的 hash 值和任务索引空间的 rank. 目前这两个会成为生成代码的文件名的一部分。为了在多进程环境中使用而避免文件读写冲突，每个进程独立进程代码生成。

生成的代码位于 ~/.flaggems 中，示例命名如下。

pointwise_dynamic_df4cb90608f3d65c2d1e3ffc31a41d560c2870cc8_rank_2_pid_3109092.py

hash, rank, pid

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Pointwise 算子自动代码生成

01 为什么使用 Code Generation

02 核心：PointwiseDynamicFunction

03 实现机制

04 性能分析

05 总结与展望

Broadcast

x y 都是行主序排布。x + y

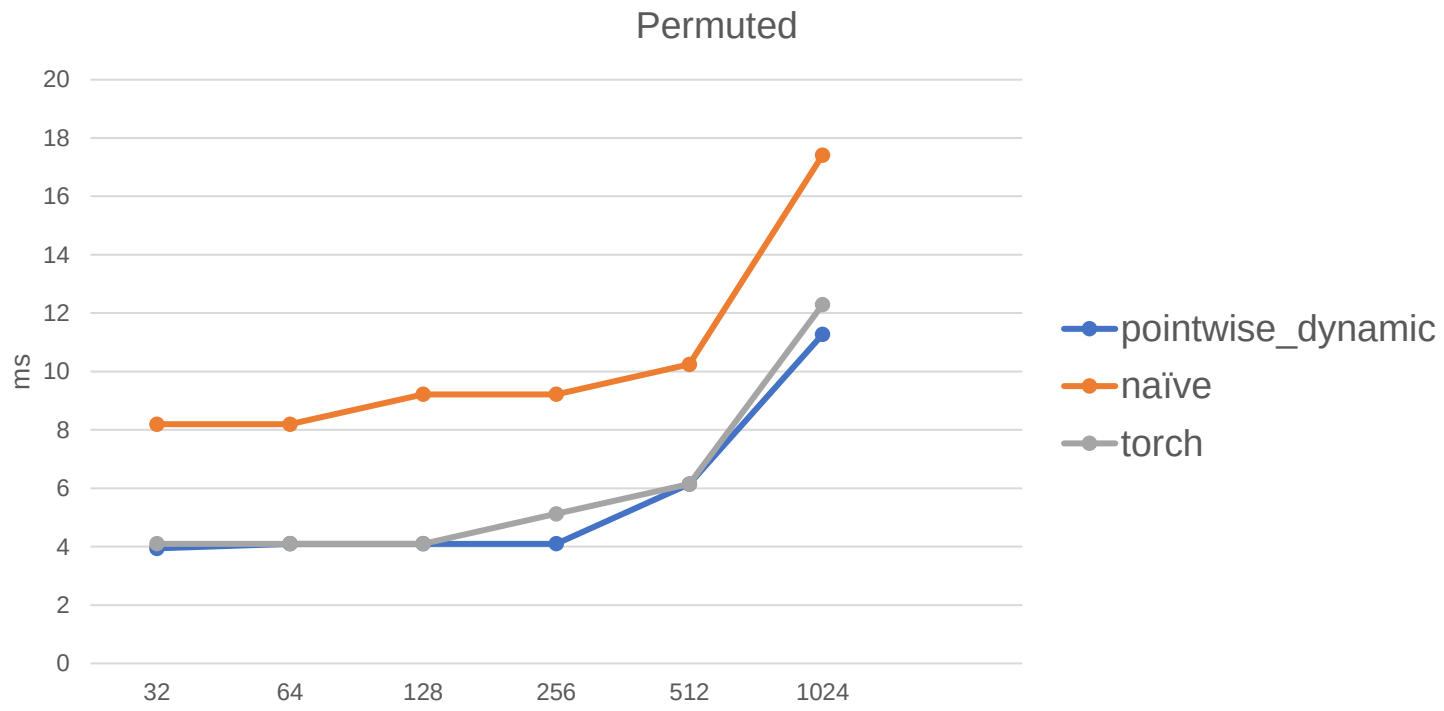
contiguous, no broadcasting
pointwise_dynamic: 4.096 naive: 4.096 torch: 4.032 | Shape: (32,), (32,)
pointwise_dynamic: 4.032 naive: 3.328 torch: 3.328 | Shape: (64,), (64,)
pointwise_dynamic: 3.840 naive: 3.840 torch: 3.744 | Shape: (128,), (128,)
pointwise_dynamic: 3.904 naive: 3.936 torch: 3.936 | Shape: (256,), (256,)
pointwise_dynamic: 4.096 naive: 4.096 torch: 4.096 | Shape: (32, 32), (32, 32)
pointwise_dynamic: 4.096 naive: 3.136 torch: 3.776 | Shape: (64, 64), (64, 64)
pointwise_dynamic: 4.096 naive: 4.096 torch: 4.096 | Shape: (128, 128), (128, 128)
pointwise_dynamic: 5.120 naive: 5.120 torch: 5.120 | Shape: (256, 256), (256, 256)

contiguous, with broadcasting
pointwise_dynamic: 4.000 naive: 3.328 torch: 3.360 | Shape: (32,), (32, 1)
pointwise_dynamic: 3.936 naive: 3.264 torch: 3.872 | Shape: (64,), (64, 1)
pointwise_dynamic: 3.968 naive: 3.360 torch: 3.776 | Shape: (128,), (128, 1)
pointwise_dynamic: 3.936 naive: 4.000 torch: 3.840 | Shape: (256,), (256, 1)
pointwise_dynamic: 4.096 naive: 4.096 torch: 3.968 | Shape: (32, 32), (1, 32)
pointwise_dynamic: 4.096 naive: 4.096 torch: 4.096 | Shape: (64, 64), (1, 64)
pointwise_dynamic: 4.096 naive: 4.096 torch: 4.096 | Shape: (128, 128), (1, 128)
pointwise_dynamic: 9245.696 naive: 9240.576 torch: 9243.616 | Shape: (25600, 25600), (1, 25600)
pointwise_dynamic: 4.096 naive: 4.096 torch: 4.096 | Shape: (25600, 1), (1, 25600)
pointwise_dynamic: 18.432 naive: 18.432 torch: 18.432 | Shape: (1024, 1, 1024), (2560, 1)

这种情况 pointwise_dynamic 生成的性能略差。因为并没有针对 contiguous 做特别处理，而是按照一般情形去计算 memory offset,所以性能差点。而且也说明 torch.broadcast_tensors 和 contiguous 确实优化不错。

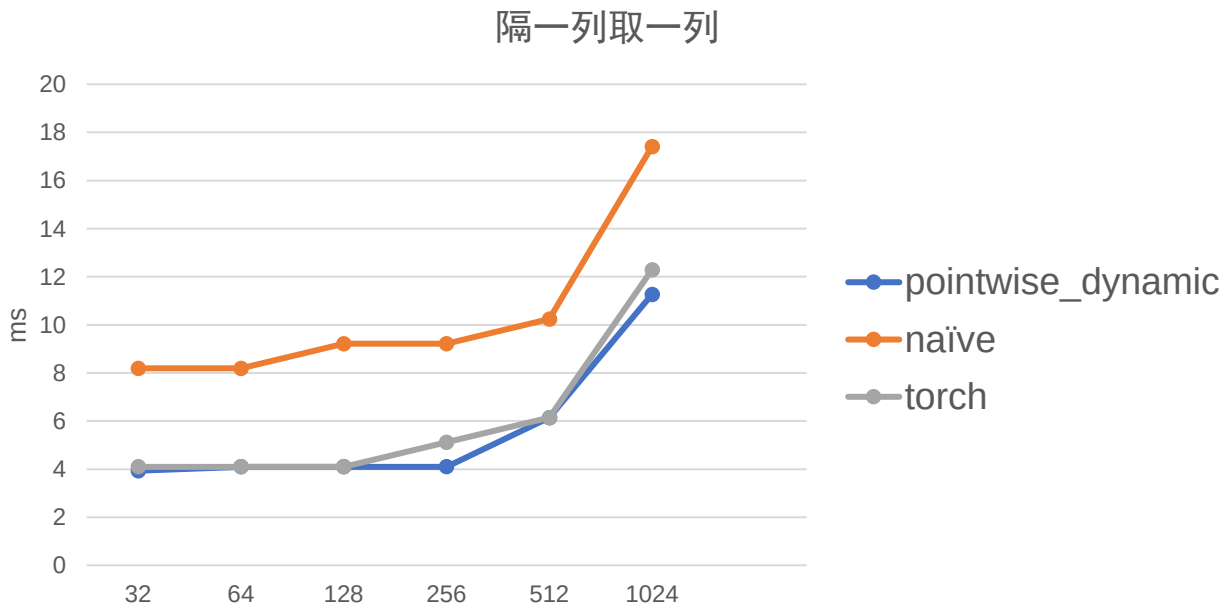
Permuted

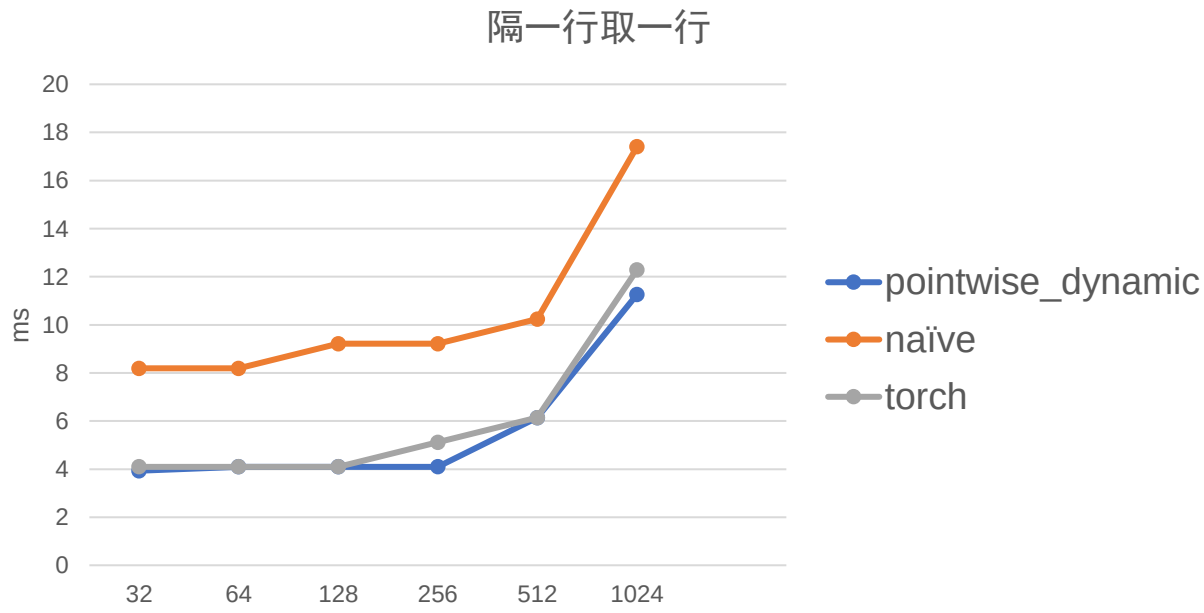
x 和 y 都是方阵，x 列主序，y 行主序。 x+ y



Strided

x, y 都是从行主序方阵上的切片。





北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

Pointwise 算子自动代码生成

01 为什么使用 Code Generation

02 核心：PointwiseDynamicFunction

03 实现机制

04 性能分析

05 总结与展望

总结与展望

在设计代码生成的时候借鉴了

- torch 的 TensorIterator, 以及 numpy nditer 来设计。其中二者是在写算子时作为 library 来使用的，提供了一个高级迭代器的功能，也可以把 map 一个标量运算函数。
- torch inductor. 它主要是用作 jit compiler, 生成的算子是针对 shape 和 stride 特化的。

后续展望

- 生成的代码后续优化 TODO：
  - 减少冗余计算，面对 contiguous 做特别处理，
  - 对输出 tensor 的布局，使用更好的算法，实现更好的内存访问模式。
  - 根据输入 tensor 的 shape 和 stride 进行分析，对任务索引空间进行混排来实现更好的内存访问模式。目前 pointwise dynamic 里任务索引空间都是行主序的。
- 函数嵌入：triton 目前不支持传入把 JITFunction 作为参数传给 JITFunction. 所以函数嵌入是通过 python AST 层来实现的。如果能让生成的代码直接调用被装饰的函数，可以做得更稳健。
- 可能支持 reduction 类算子，欢迎参与贡献。

谢谢聆听！

欢迎访问开源社区：

https://github.com/FlagOpen/FlagGems

https://github.com/FlagOpen/FlagAttention

Triton中国社区微信群

扫码添加微信，私信“加群”

北京智源人工智能研究院
BEIJING ACADEMY OF ARTIFICIAL INTELLIGENCE

自 由 讨 论 环 节

BAAI
智源研究院

加入我们：
JOIN US

社会招聘
Social Recruitment

校园招聘
Campus Recruitment

扫描二维码即刻投递简历
Scan QR code to immediately submit resume.