天数智芯

IxFormer(vLLM) 大模型推理框架使用指南

版本：V4.0.1-MR

日期：2024.8.9

适用产品：智铠 50| 智铠 100

1 声明

1.1 版权声明

1.2 免责声明

天数智芯可以随时对本文档或本文档中描述的产品进行改进和/或更改。本文档包括与天数智芯产品有关的信息，作为说明典型应用的一种方式, 因此，不一定提供足以进行生产设计的完整信息。对于本文档中内容的准确性或完整性，天数智芯不做任何陈述或保证。

1.3 联系方式

地址：上海闵行区陈行公路 2168 号 3 幢

电话：021-68886607

网址：www.iluvatar.com

Contents

2 IxFormer(vLLM) 大模型推理框架使用指南

2.1 修订说明

- COREX01-MR401-UG10-00：2024/8/9

  文档本次发布内容与 V4.0.0-MR 文档相比 (COREX01-MR400-UG10-00)，有以下更新：

  - 新增Xinference 和Ollama，提供限制说明

2.2 IxFormer 推理框架功能说明

IxFormer 是天数智芯设计的专用于大模型推理优化的加速框架，支持业界目前主流大模型的推理加速，实现大模型在天数智芯加速卡上的最佳推理性能。IxFormer 底层使用 C++ 实现，封装了多个高效算子；在 Python 层面，IxFormer 提供了 Operation, Layer, Model 等多个层次的 API，支持 FP16, INT8, INT4 等多种数据结构下的加速。当前提供的 IxFormer 版本是 v0.4.0。

IxFormer 推理框架主要包含以下内容：

- 优化算子库：基于天数智芯加速卡硬件特性，IxFormer 重新实现了丰富的算子库，算子库中包括了基础算子、融合算子以及低精度算子用于模型加速。
- 兼容设计：IxFormer 设计兼容 PyTorch 框架，并提供自动转换工具用于使用 IxFormer 算子加速 PyTorch 模型，从而可以在推理中使用并加速 PyTorch 模型。
- 内置推理模型：借助于高效算子库，IxFormer 提供了高度优化的内置模型，结合张量模型并行和流水线模型并行，实现模型性能加速。详细的模型支持列表，请参考附录 1：IxFormer 支持的模型列表。
- 推理框架支持：得益于框架的兼容性设计，IxFormer 可以使用加速算子对第三方框架进行加速，IxFormer 目前提供了对 vLLM 和 Text Generation Inference (TGI) 框架的推理加速支持。更多详细内容，请参考IxFormer 支持的推理框架说明。

IxFormer 实现了 Tensor 管理以及丰富的算子库等构建模型需要的特性，同时，IxFormer 被设计为兼容 PyTorch。因此，在推理场景中，您可以使用 IxFormer 对基于 PyTorch 的推理框架进行加速，从而支持以下重要特性：

- 支持 IxFormer 基础算子加速
- 支持 IxFormer 接口基础算子自动替换
- 支持主流大模型推理加速

详细信息，请参考IxFormer 推理指南。

2.3 安装 IxFormer 推理框架

本小节旨在介绍 IxFormer 推理框架的安装过程，当前仅支持通过 Python 库安装 IxFormer。

1. 确保您已经成功安装天数智算软件栈，详情请参考《软件栈安装指南》。

2. 向您的应用工程师获取 IxFormer 的 `.whl` 包，获取到的 `.whl` 包必须和您当前环境的 Python 版本保持一致。

   若您使用 Docker 安装软件栈，则无需额外安装 `.whl` 包，天数智芯提供的 Docker 镜像安装包内已包含 IxFormer 推理框架。

3. 使用 **pip3 install** 命令安装 IxFormer，例如：

   ```
   $ pip3 install ixformer-{v.r.m}-cp37-cp37m-linux_x86_64.whl
   ```

4. 安装完成后，在终端运行如下命令检查安装是否成功：

   ```
   $ python3 -c "import ixformer; print(ixformer.__file__)"
   ```

   如果回显版本号，表示安装成功。

2.4 IxFormer 推理指南

为了最大化模型的推理性能，IxFormer 在模型层面设计了高效的基础算子、融合算子和低精度算子，从框架层面设计了高效的推理模式。

本小节旨在指导使用 IxFormer 对模型进行推理，并介绍 IxFormer 内置的推理模型。

2.4.1 如何优化推理

IxFormer 为您提供以下方式对模型推理进行性能优化：

- 开启推理模式
- 使用更加高效的基础算子
- 自动替换基础算子
- 使用融合算子
- 使用更低精度进行推理

2.4.1.1 开启推理模式

在进行模型推理前，请您开启推理模型以最大化推理的性能。您只需要在推理前加入以下代码即可：

```
from ixformer.autograd import enable_inference_mode
enable_inference_mode()
```

2.4.1.2 使用更加高效的基础算子

对于基础算子，IxFormer 在使用上保持了和 PyTorch 相同的接口。在大部分场景下，您只需要将 `torch.xxx_func` 替换为 `ixformer.functions.xxx_func` 即可使用更高效的基础算子：

```
import numpy as np
import torch
import ixformer

x = torch.randn(2, 64, 32, dtype=torch.half, device="cuda", requires_grad=True)
w = torch.randn(32, 64, dtype=torch.half, device="cuda", requires_grad=True)
# In PyTorch
out = torch.matmul(x, w)
```

上述代码中的 `matmul` 函数可以被替换为 IxFormer 中的函数：

```
# Replace by IxFormer
out = ixformer.functions.matmul(x, w)
```

2.4.1.3 自动替换基础算子

对于较复杂的模型，手动将 PyTorch 算子替换为 IxFormer 算子仍是一件较复杂的事情。为了更加便捷地使用 IxFormer 加速 PyTorch 模型，IxFormer 提供了以下接口用于转换并加速 PyTorch 模型：

```
ixformer.contrib.torch.model_accelerator.accelerator_torch_model(
    model: torch.nn.Module = None,
    patch_functions: bool = True,
    functions: List[PatcherItem] = None,
    include_functions: List[str] = None,
    exclude_functions: List[str] = None,
    include_modules: List[Union[Type[torch.nn.Module], str]] = None,
    exclude_modules: List[Union[Type[torch.nn.Module], str]] = None)
```

上述接口主要提供两个功能：

1. 将 PyTorch 模型中部分模块的实现替换为 IxFormer 中的实现
2. 将 PyTorch 模型中部分函数的实现替换为 IxFormer 中的实现

在转换的过程中，您可以通过参数来控制需要转换的模块或函数。参数说明如下：

- `model`：输入一个可选的 PyTorch 模型。如果传入模型，则会对模型中的部分模块进行替换
- `patch_functions`：是否使用 IxFormer 中的函数替换 PyTorch 中的实现
- `functions`：自定义替换 PyTorch 的函数
- `include_functions`：需要替换的 PyTorch 函数
- `exclude_functions`：不需要替换的 PyTorch 函数
- `include_modules`：需要替换的模型中的 PyTorch 模块
- `exclude_modules`：不需要替换的模型中的 PyTorch 模块

通过调入并调用 `accelerator_torch_model` 函数，您便可以将 PyTorch 的算子调用转变为使用 IxFormer 的算子调用：

```
from ixformer.contrib.torch.model_accelerator import accelerator_torch_model
accelerator_torch_model()
...
x = torch.randn(2, 64, 32, dtype=torch.half, device="cuda", requires_grad=True)
w = torch.randn(32, 64, dtype=torch.half, device="cuda", requires_grad=True)
# 这里的矩阵乘将会使用 IxFormer 中的 matmul 算子进行计算
out = torch.matmul(x, w)
...
```

目前，accelerator_torch_model 函数在不指定替换算子的情况下，将默认替换为以下函数：

- linear
- matmul

对于其它函数，则需要手动替换。通过继承 torch.nn.Module 实现一个示例模型：

```
import torch
import torch.nn.functional as F
import ixformer.functions as ixff

class DummyClass(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = torch.nn.Linear(32,10)
        self.fc2_weight = torch.nn.Parameter(torch.empty((10, 32),dtype=torch.float16))

    def forward(self, inputs):
        x1,x2 = torch.chunk(inputs,2,dim=-1)
        x1 = x1.contiguous()
        x2 = x2.contiguous()
        x1 = self.fc1(x1)
        x2 = F.linear(x2,self.fc2_weight)
        out = torch.cat([x1,x2],dim=-1)
        return out
```

在未使用任何转换时，chunk , linear 和 cat 将使用默认的 PyTorch 函数：

```
print(torch.chunk == ixff.chunk) # >>> false
print(F.linear == ixff.linear) # >>> false
print(torch.cat == ixff.cat) # >>> false
```

使用 accelerator_torch_model 函数时，如果不传入参数，将默认替换为以下函数：

- linear
- matmul

此时，打印函数情况如下：

```
print(torch.chunk == ixff.chunk) # >>> false
print(F.linear == ixff.linear) # >>> true
print(torch.matmul == ixff.matmul) # >>> true
```

除了 chunk，其余函数都使用 IxFormer 提供的函数。使用结束后，您可以通过 clear_patch_torch_functions 函数对替换的函数进行恢复。此时，打印函数情况如下：

```
from ixformer.contrib.torch.model_accelerator import clear_patch_torch_functions
clear_patch_torch_functions()
print(torch.chunk == ixff.chunk) # >>> false
print(F.linear == ixff.linear) # >>> false
print(torch.matmul == ixff.matmul) # >>> false
```

由于默认的替换并没有对 chunk 函数进行替换，因此需要您手动进行设置：

```
from ixformer.contrib.torch.model_accelerator.converter import PatcherItem
accelerator_torch_model(functions=[PatcherItem('chunk',ixff.chunk,['torch']),])
```

设置完成后，打印函数情况如下：

```
print(torch.chunk == ixff.chunk) # >>> true
print(F.linear == ixff.linear) # >>> false
print(torch.matmul == ixff.matmul) # >>> false
```

由于手动控制时的函数替换完全由参数控制，因此默认替换将不起作用。为了替换模型中使用的所有函数，需要设置所有的函数信息：

```
accelerator_torch_model(functions=[PatcherItem('chunk',ixff.chunk,['torch']),
                                   PatcherItem('cat',ixff.cat,['torch']),
                                   PatcherItem('linear',ixff.linear,['torch.nn.functional'])])
```

此时，打印函数情况如下：

```
print(torch.chunk == ixff.chunk) # >>> true
print(F.linear == ixff.linear) # >>> true
print(torch.cat == ixff.cat) # >>> true
```

函数中 PatcherItem 的参数包括以下三项内容：

- PyTorch 中的函数名称 (字符串形式)
- 替换后的 IxFormer 函数
- 被替换的函数来源

其中，PyTorch 的函数通常位于 torch 和 torch.nn.functional 下，您可以根据原始调用的函数来源选择相应的参数。

除了替换函数，accelerator_torch_model 也可以对模型中的模块进行替换，该函数调用后会对后续的函数调用发生影响，但对模型中的模块替换只在当前模型起作用。设定一个自定义 Linear 对原始的 Linear 进行替换：PatcherItem

```
from ixformer.contrib.torch.model_accelerator import register_module_converter

class MyLiner(torch.nn.Module):
    def __init__(self, weight, bias):
        super().__init__()
        self.weight = weight
        self.bias = bias
    def forward(self, inputs):
        return ixff.linear(inputs, self.weight, self.bias)

@register_module_converter(torch.nn.Linear)
def convert_linear(name, old_mod):
    return MyLiner(old_mod.weight, old_mod.bias)

accelerator_torch_model(model,patch_functions=False,include_modules=[torch.nn.Linear])
print(isinstance(model.fc1,MyLiner)) # >>> true
model_new = DummyClass().half().cuda()
print(isinstance(model_new.fc1, MyLiner)) # >>> false
```

在上述函数中，由于设置了 patch_functions=False，因此模型中仍然调用 PyTorch 的原始算子。此时，打印函数情况如下：

```
print(torch.chunk == ixff.chunk) # >>> false
print(F.linear == ixff.linear) # >>> false
print(torch.cat == ixff.cat) # >>> false
```

除了 accelerator_torch_model 函数，IxFormer 还提供了一个上下文管理器 accelerator_context 进行模块或函数的替换。该接口与 accelerator_torch_model 类似，但其作用域仅限于特定的上下文内，上下文外将不受影响，从而增加控制的细粒度。

```
ixformer.contrib.torch.model_accelerator.accelerator_context(
    model: torch.nn.Module = None,
    patch_functions: bool = True,
    functions: List[PatcherItem] = None,
    include_functions: List[str] = None,
    exclude_functions: List[str] = None,
    include_modules: List[Union[Type[torch.nn.Module], str]] = None,
    exclude_modules: List[Union[Type[torch.nn.Module], str]] = None)
```

该函数的参数与 accelerator_torch_model 函数中的参数相同，您可以使用新的函数实现前述例子中对 matmul 的自动替换功能。

```
from ixformer.contrib.torch.model_accelerator import accelerator_context
...
x = torch.randn(2, 64, 32, dtype=torch.half, device="cuda", requires_grad=True)
w = torch.randn(32, 64, dtype=torch.half, device="cuda", requires_grad=True)
# 这里的矩阵乘将会使用 PyTorch 的实现进行计算
out = torch.matmul(x, w)
# 这里的矩阵乘将会使用 IxFormer 的实现进行计算
with accelerator_context():
    out = torch.matmul(x, w)
```

以 DummyClass 的 forward 函数为例，使用 accelerator_torch_model 函数可以控制需要替换的函数，对初始的 forward 进行简单修改：

```
def forward(self, inputs):
    with accelerator_context(functions=[PatcherItem('chunk',ixff.chunk,['torch']),
                                PatcherItem('cat',ixff.cat,['torch']),
                                PatcherItem('linear',ixff.linear,['torch.nn.functional'])]):
        x1,x2 = torch.chunk(inputs,2,dim=-1)
        x1 = x1.contiguous()
        x2 = x2.contiguous()
        print(torch.chunk == ixff.chunk) # >>> true
        x1 = self.fc1(x1)
        x2 = F.linear(x2,self.fc2_weight)
        print(F.linear == ixff.linear) # >>> true

    out = torch.cat([x1,x2],dim=-1)
    print(torch.chunk == ixff.chunk) # >>> false
    print(F.linear == ixff.linear) # >>> false
    print(torch.cat == ixff.cat) # >>> false

    return out
```

accelerator_context 函数会在退出作用域后自动恢复替换的算子。因此，需要您在其作用域下打印函数状况、构建模型和输入：

```
model = DummyClass().half().cuda()
data = torch.randn(10,64,dtype=torch.float16).cuda()
model(data)
```

在作用域内，将根据设置需要替换的函数信息进行替换，而在退出作用域后，则恢复至原始的函数。PyTorch 的众多模块最终会进行函数调用，因此使用函数替换可以完成底层的算子替换，而模型替换可能需要您搭建与 PyTorch 等价的模块。但不管如何，您都可以使用 IxFormer 提供的函数完成您所需要的替换功能。

目前，IxFormer 中提供的基础算子还在持续增加，参考ixformer.functions (基础算子) 了解详情。

如果您有替换 PyTorch 原有算子或开发新算子的需求，可以通过修改添加源代码或联系您的应用工程师进行解决。

2.4.1.4 使用融合算子

在多数情况下，IxFormer 实现的融合算子并不存在于 PyTorch 中，因此无法如自动替换基础算子 中描述的那样使用 accelerator_torch_model 函数来替换 PyTorch 的原有算子。此时，您需要手动将 IxFormer 实现的算子在模型中进行替换。

下面以 RMS_norm 的 PyTorch 实例为例进行说明：

```
class RMSNorm(nn.Module):
    def __init__(self, num_features, eps=1e-8):
        super(RMSNorm, self).__init__()

        self.eps = eps
        self.weight = nn.Parameter(torch.ones(hidden_size))

    def forward(self, x):
        mean_square = torch.mean(x**2, dim=-1, keepdim=True)
        norm = torch.sqrt(mean_square + self.eps)
        x = x / norm * self.weight

        return x
# in other model
     def __init__(self, num_features)
         self.rms_norm = RMSNorm(num_features)
         ...
     def forward(self, x):
         norm_x = self.rms_norm(x)
         ...
```

您可以使用 IxFormer 提供的融合算子进行替换：

```
from ixformer.functions import rms_norm
# in other model
    def __init__(self) :
        self.norm_weight = ...
        ...
    def forward(self, x):
        norm_x = rms_norm(x,self.norm_weight)
        ...
```

目前，IxFormer 提供的融合算子还在持续增加，参考ixformer.functions (融合算子) 了解详情。

2.4.1.5 使用更低精度进行推理

目前，IxFormer 主要支持 FP16 精度进行推理，INT8 和 INT4 精度的推理也支持。得益于 IxFormer 和 PyTorch 的兼容性，您在使用 FP16 进行推理时，无需修改相关代码即可快速推理。对于 INT8 和 INT4 推理，则需要先完成对应的模型量化操作。

2.4.2 内置模型列表

基于 IxFormer 提供的算子库及相关特性，天数智芯提供了如下模型供您进行推理 (向您的应用工程师获取相应代码和使用说明)：

模型	精度
ChatGLM	FP16
ChatGLM2	FP16
GLM130B	INT8
LLaMA	FP16
Vicuna	FP16

目前，IxFormer 内置的模型都被统一存放在 IxFormer 的 Model Zoo 中，您可以在附录 1：IxFormer 支持的模型列表 中查看所有支持的模型 (包括支持的其它推理模型)。

2.5 IxFormer 支持的推理框架说明

得益于框架的兼容性设计，IxFormer 支持使用加速算子对第三方框架进行加速。目前，IxFormer 提供了对 vLLM 和 TGI 框架进行推理加速的支持。

2.5.1 vLLM

通过在天数智芯加速卡上实现 PagedAttention 的张量缓存管理核心机制，IxFormer 提供了 vLLM 框架的推理支持。当前支持的版本为 vLLM v0.2.6。在 vLLM 的模型实现中，大部分算子使用在天数智芯加速卡上实现的高效算子，结合 vLLM 的前端处理，提升模型吞吐量。

2.5.1.1 使用 vLLM 进行快速推理

IxFormer 支持您使用 vLLM 对vLLM 支持的模型列表 中提供的模型进行快速推理。

示例

下面以 LLaMA2-13B 模型为例，说明如何使用 vLLM 进行快速推理。

从 Hugging Face 或天数智芯提供的文件 (向您的应用工程师获取相关文件) 中获取模型权重，并将其存放至环境的 `/home/data/llama2/llama2-13b` 目录下。IxFormer 支持的 vLLM 推理框架保持了同开源框架一致的使用流程。

1. 首先，从 IxFormer 中导入必要的模块：

```
from vllm import LLM, SamplingParams
```

2. 利用 LLM 和 SamplingParams 分别创建执行推理的模型实例和生成文本管理的参数实例：

```
llm = LLM(model="/home/data/llama2/llama2-13b",
          gpu_memory_utilization=0.9,
          max_num_batched_tokens=2048,
          tokenizer_mode="slow",
          tensor_parallel_size=1
         )
sampling_params = SamplingParams(temperature=0.8, top_p=0.95,
                                 max_tokens=1024,
                                 )
```

在上述代码中，

- LLM 的初始化函数会根据模型权重位置自动加载模型额配置文件
- gpu_memory_utilization 用于控制加速卡内存的占用比例
- max_num_batched_tokens 用于控制模型最大的输入 tokens 数量
- tensor_parallel_size 用于控制模型张量并行度

如果使用两张加速卡进行模型的并列推理，则可以设置 tensor_parallel_size=2。更多详细的参数信息，请参考源代码文件。

3. 设置输入文本并进行推理：

```
prompts = [
            "Shanghai is one of the most prosperous cities in China,"
          ]
outputs = llm.generate(prompts, sampling_params)
prompt = outputs[0].prompt
generated_text = outputs[0].outputs[0].text
```

generate 函数完成输入文本到 tokens 的转换、执行推理以及输出 tokens 到输出文本的转换，通过打印模型的输入输出，获得结果：

```
print(generated_text)
>>>
and it is also the largest city in the country. While visiting Shanghai, it is important to
 ↪  be  aware of the local laws and regulations, including those related to traffic safety.
 ↪  In Shanghai, traffic safety is a top priority, and the local authorities have
 ↪  implemented various measures to ensure the safety of ped
```

总结

上述流程是使用 vLLM 进行离线推理的主要流程，更多详细内容，请参考 vLLM 官方网站 或附录 1：IxFormer 支持的模型列表 (向您的应用工程师获取模型推理相应代码和使用说明)。

2.5.1.2 使用 vLLM 进行 Server 形式推理

vLLM 提供了两种 Server 推理方式，第一种为 vLLM 自身实现的 Server，可以用于 vLLM 模型的快速使用和验证，但功能性略不足。第二种为官方建议的使用基于 OpenAI 接口的 Server 推理。下面将分别介绍两种推理方式。

方式一：vLLM 自身实现的 Server 推理

1. 首先，通过以下命令启动 Server：

   ```
   python3 -m vllm.entrypoints.api_server --model /path/to/node --host 127.0.0.1 --port 12345
   ```

   其中，主要设置模型文件路径以及 Server 的启动地址，其他参数与快速推理中的相同。

2. 使用 **curl** 进行快速验证：

   ```
   curl -s 127.0.0.1:12345/generate \
      -X POST \
      -d '{
         "prompt":"Shanghai is one of the most prosperous cities in China,",
         "temperature":0.0,
         "max_tokens":128,
         "stream":"true"}' \
      -H 'Content-Type: application/json'
   ```

或者，您也可以通过 Python 脚本进行使用：

```
import requests

headers = {"User-Agent": "Test Client"}
pload = {
   "prompt": "Shanghai is one of the most prosperous cities in China,",
   "temperature": 0.0,
   "max_tokens": 128,
   "stream": False,
}
response = requests.post(http://127.0.0.1:12345/generate, headers=headers, json=pload)
data = json.loads(response.content)
output = data["text"]

print(output)
```

方式二：使用基于 OpenAI 接口的 Server 推理

由于 vLLM 自身的 Server 推理框架提供的功能较少，因此，建议您使用 OpenAI 进行 Server 推理。

1. 首先，通过以下命令启动 Server：

   ```
   python3 -m vllm.entrypoints.openai.api_server --model /path/to/node --host 127.0.0.1 --port
   ↪ 12345
   ```

2. 使用 **curl** 进行快速验证：

```
curl http://127.0.0.1:12345/v1/models

curl -s 127.0.0.1:12345/v1/completions \
   -H "Content-Type: application/json" \
   -d '{"model":"/model/name/",
      "prompt":"Shanghai is one of the most prosperous cities in China,",
      "temperature":0.0,
      "max_tokens":128,
      "stream":"true"}'
```

其中，/model/name/ 为通过第一行命令查询得到的正在运行的模型 ID，之后便可以发起请求。

或者，您也可以通过 Python 脚本进行使用：

```
from openai import OpenAI

openai_api_key = "EMPTY"
openai_api_base = "http://127.0.0.1:12345/v1"

client = OpenAI(
   api_key=openai_api_key,
   base_url=openai_api_base,
)

models = client.models.list()
model = models.data[0].id

completion = client.completions.create(
   model=model,
   prompt="Shanghai is one of the most prosperous cities in China,",
   echo=False,
   stream=False)

print(completion)
```

OpenAI 同时提供了 AsyncClient，Chat 接口等功能，详情请参考 OpenAI 官方文档。

2.5.1.3 vLLM 支持的模型列表

vLLM 目前支持以下模型 (向您的应用工程师获取相应代码和使用说明)：

- Aquila
- Baichuan & Baichuan2
- BLOOM
- ChatGLM

- GPT-2
- GPT BigCode
- GPT-NeoX
- InternLM
- LLaMA & LLaMA2
- Mixtral
- OPT
- Qwen & Qwen1.5
- Yi

以上模型都被统一存放在 IxFormer 的 Model Zoo 中，您可以在附录 1：IxFormer 支持的模型列表 中查看所有支持的模型 (包括支持的其他推理模型)。

2.5.2 TGI

TGI (Text Generation Inference) 是 HuggingFace 支持的推理部署工具，具有和 vLLM 类似的功能。IxFormer 同样提供了对 TGI 框架的推理支持，并支持使用 IxFormer 算子对模型进行加速。当前支持的版本为 TGI v1.1.0。

2.5.2.1 使用 TGI 进行快速推理

示例

下面以 LLaMA-13B 模型为例，说明如何使用 TGI 进行快速推理。

从 Hugging Face 或天数智芯提供的文件 (向您的应用工程师获取相关文件) 中获取模型权重，并将其存放至环境的 `/home/data/llama/llama-13b` 目录下。

1. 在运行服务前，需要先安装 Protoc 和 Rust 用于前端服务：

   - 安装 Protoc：

   ```
   # PROTOC_ZIP=protoc-21.12-linux-x86_64.zip
   $ curl -OL https://github.com/protocolbuffers/protobuf/releases/download/
    ↪  v21.12/$PROTOC_ZIP
   $ sudo unzip -o $PROTOC_ZIP -d /usr/local bin/protoc
   $ sudo unzip -o $PROTOC_ZIP -d /usr/local 'include\*\'
   ```

   - 安装 Rust：

   ```
   $ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
   ```

2. 在本例中，使用 Server 方式进行模型推理。

   - 对于单卡：

   ```
   $ export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python
   $ CUDA_VISIBLE_DEVICES=0 USE_FLASH_ATTENTION=true \
     text-generation-launcher  --model-id /home/data/llama/llama-13b/ \
   ```

```
  --sharded false --dtype float16 --disable-custom-kernels \
  --trust-remote-code --max-batch-total-tokens 4096 --port 4000
```

- 对于多卡：

```
$ export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python
$ CUDA_VISIBLE_DEVICES=0,2 USE_FLASH_ATTENTION=true text-generation-launcher  --model-
↪  id /home/data/llama/llama-13b/ \
  --sharded true --num-shard 2 --dtype float16 --disable-custom-kernels  \
  --trust-remote-code --max-batch-total-tokens 4096 --port 4000
```

与在 vLLM 中演示的离线推理不同，以 Server 方式运行推理的方式不需要显式编写 Python 文件的过程，vLLM 同样支持以 Server 方式进行推理。在 Server 方式下，您可以直接使用 curl 等发起服务请求：

- 非流式服务请求：

```
$ curl 127.0.0.1:4000/generate \
      -X POST \
      -d '{"inputs":"Q: What is the largest animal?
↪  \nA:","parameters":{"max_new_tokens":64}}' \
      -H 'Content-Type: application/json'
```

- 流式服务请求：

```
$ curl 127.0.0.1:4000/generate_stream \
      -X POST \
      -d '{"inputs":"Q: What is the largest animal?
↪  \nA:","parameters":{"max_new_tokens":20}}' \
      -H 'Content-Type: application/json'
```

总结

以上是基于 IxFormer 进行 TGI Server 推理的主要流程，详细的使用流程请参考 TGI 官方网站。

2.5.3 Xinference

目前天数智芯只支持 Transformers 和 vLLM 后端，使用下面命令安装 Xinference：

```
$ pip install --upgrade-strategy only-if-needed "xinference[transformers,vllm]"
```

2.5.4 Ollama

Ollama 后端仅支持 `llama.cpp`，目前天数智算软件栈未适配 `llama.cpp`，因此不支持 Ollama。

2.6 附录 1：IxFormer 支持的模型列表

IxFormer 提供的 Model Zoo 主要包括 IxFormer 内置模型和 vLLM 模型。其中 **Official** 是 IxFormer 的内置模型，同时，**Official** 中的模型保持了和官方一致的接口和功能。

IxFormer 还在持续发展中，将会有更多的模型被列入支持列表中。

Note

- 请向您的应用工程师获取下表中的模型代码和使用说明。
- 列表中的模型为当前通过验证的模型。

Architecture	Models	vLLM (含代码)	Official (含代码)	TGI (含代码)	Note
LLaMA	Belle LLaMA LLaMA-2	Belle-7B LLaMA-7B (Chat) LLaMA-13B (Chat) LLaMA-65B (Chat) LLaMA2-7B (Chat) LLaMA2-13B (Chat) LLaMA2-70B (Chat)		LLaMA-7B (Chat) LLaMA-13B (Chat) LLaMA-65B (Chat) LLaMA2-7B (Chat) LLaMA2-13 B (Chat) LLaMA2-70 B (Chat) LLaMA-2-7 B-Chat-GP TQ-INT8	
LLaMA	LLaMA-2-AWQ	LLaMA2-7B-AWQ (Chat) LLaMA2-13B -AWQ (Chat) LLaMA2-70B -AWQ (Chat)			vLL M AWQ 目前只支持 INT 4 量化推理
LLaMA	LLaMA-2-SmoothQuant	LLaMA2-7B-Chat-SmoothQuant			vLL M SmoothQuant 目前只支持 LLa MA2
LLaMA	CodeLLaMa-AWQ	CodeLLaMa-13B-AWQ			
ChatGL M	ChatGLM		ChatGLM-6B		Cha tGL M-6 B 接口与官方接口一致，支持 C hat ，P-tuni ng 等功能

Architecture	Models	vLLM (含代码)	Official (含代码)	TGI (含代码)	Note
ChatGLM2	ChatGLM2	ChatGLM2-6B ChatGLM2-6B-32K	ChatGLM2-6B ChatGLM2-6B-32K	ChatGLM2-6B ChatGLM2-6B-32K	ChatGLM2-6B Official 接口与官方接口一致，支持 Chat，P-tuning 等功能，通常情况下，vLLM、TGI、HuggingFace 性能更好
ChatGLM3	ChatGLM3	ChatGLM3-6B ChatGLM3-6B-32K	ChatGLM3-6B	ChatGLM3-6B ChatGLM3-6B-32K	ChatGLM3-6B 接口与官方接口一致，支持 Chat 功能
Baichuan	Baichuan	Baichuan-7B (Chat) Baichuan-13B (Chat) Baichuan2-7B (Chat) Baichuan2-13B (Chat)	Baichuan2-13B (Chat)		Transformers 库中调用的 bitsandbytes 需要用适配的版本
Qwen	Qwen	Qwen-7B (Chat) Qwen-14B (Chat) Qwen-72B (Chat) with AWQ and GPTQ-INT4		Qwen-7B (Chat) Qwen-14B (Chat)	

Architecture	Models	vLLM (含代码)	Official (含代码)	TGI (含代码)	Note
Qwen1. 5	Qwen1.5	Qwen1.5-0. 5B (Chat) Qwen1.5-1. 8B (Chat) Qwen1.5-4B (Chat) Qwen1.5-7B (Chat) Qwen1.5-14 B (Chat) Qwen1.5-72 B (Chat) with AWQ and GPTQ-INT4			
CPM	MiniCPM	MiniCPM-2B			
Yi	Yi	Yi-6B (chat) Yi-34B (chat)			
Mixtra l	Mixtral	Mixtral-8X 7B			
Aquila	Aquila	AquilaChat -7B			
Aquila	AquilaChat2-34B-INT4				
Bloom	Bloom Bloomz	Bloom-7B1			
GPT-2	GPT-2 UTF-8	GPT2-XL GPT3-13B			
GPTBigCode	SantaCoder	SantaCoder			
GPTNeo X	GPT-NeoX	StableLM-Tuned-Alpha -7B			
Intern LM	InternLM	InternLM-7 B			
Opt	Opt Opt-IML	OPT-6.7B			
ChatGLM-130B	ChatGLM-130B		ChatGLM-130B		
CodeShell-7B-Chat	CodeShell-7B-Chat		CodeShell-7B-Chat		

Architecture	Models	vLLM (含代码)	Official (含代码)	TGI (含代码)	Note
CodeShell-7B-Chat	CodeShell-7B-Chat-4Bit		CodeShell-7B-Chat-4bit		
Clip	Clip		Clip		
CPM-Bee	CPM-Bee		CPM-Bee		
Diffusers	Diffusers		Diffusers		支持 1.5、2.1、XL 模型
LLaMA-Adapter	LLaMA-Adapter-V2-Multimodal7B		LLaMA-Adapter-V2-Multimodal7B		
CodeGeeX2	CodeGeeX2			CodeGeeX2-6B	

2.7 附录 2：IxFormer API

IxFormer API 包含：

- ixformer.functions (基础算子)
- ixformer.functions (融合算子)

2.7.1 ixformer.functions (基础算子)

Name	IxFormer API
to	ixformer.functions.to
copy	ixformer.functions.copy
full	ixformer.functions.full
full_like	ixformer.functions.full
empty	ixformer.functions.empty
empty_like	ixformer.functions.empty
zeros	ixformer.functions.zeros
zeros_like	ixformer.functions.zeros
ones	ixformer.functions.ones

Name	IxFormer API
ones_like	ixformer.functions.ones
split	ixformer.functions.split
chunk	ixformer.functions.chunk
linear	ixformer.functions.linear
concat	ixformer.functions.cat
matmul	ixformer.functions.matmul
reshape	ixformer.functions.reshape
view	ixformer.functions.reshape
conv2d	ixformer.functions.conv2d
groupnorm	ixformer.functions.group_norm
softmax	ixformer.functions.softmax
permute	ixformer.functions.permute
transpose	ixformer.functions.transpose
contiguous	ixformer.functions.contiguous

2.7.1.1 ixformer.functions.to

```
to(input: "ixformer.Tensor", device: Union[int, str, Device]) -> Tensor
```

参数说明

- `input`：需要将 `input` 转换到 `device` 的 Tensor
- `device`：如果为 int，那么会将 Tensor 转换到 `cuda:<device>`；如果为 str，支持 `cpu` 和 `cuda:<GPU_ID>`

数据类型

All

Shapes

All

2.7.1.2 ixformer.functions.copy

```
copy(input: "ixformer.Tensor", non_blocking: bool = False, out: "ixformer.Tensor" = None) ->
↪ Tensor
```

参数说明

- input：复制 input
- non_blocking：是否异步拷贝数据
- out：将数据从 input 拷贝到 out 中

数据类型

All

Shapes

All

2.7.1.3 ixformer.functions.full

```
full(shape: List[int],
     fill_value: Union[int, float],
     dtype: "ixformer.DataType" = None,
     device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
     layout: "ixformer.strided" = None, requires_grad: bool = False
    ) -> "ixformer.Tensor"
```

参数说明

- shape：Tensor shape
- fill_value：填充值
- dtype：Tensor 的数据类型
- device：Tensor 的 Device
- layout：数据排布
- requires_grad：是否需要梯度

数据类型

All

Shapes

All

```
full_like(input: "ixformer.Tensor",
          fill_value: Union[int, float],
          dtype: "ixformer.DataType" = None,
          device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
          layout: "ixformer.strided" = None,
          requires_grad: bool = False
         ) -> "ixformer.Tensor"
```

参数说明

- input：创建一个和 input 属性相同的 Tensor
- fill_value：填充值
- dtype：Tensor 的数据类型
- device：Tensor 的 Device
- layout：数据排布
- requires_grad：是否需要梯度

数据类型

All

Shapes

All

2.7.1.4 ixformer.functions.empty

```
empty(*size: int,*,
      dtype: "ixformer.DataType" = None,
      device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
      layout: "ixformer.strided" = None,
      requires_grad: bool = False
     ) -> "ixformer.Tensor"
```

参数说明

- size：Tensor shape
- dtype：Tensor 的数据类型
- device：Tensor 的 Device
- layout：数据排布
- requires_grad：是否需要梯度

数据类型

All

Shapes

All

```
empty_like(input: "ixformer.Tensor", device: Union[int, str, ixformer.Device] = None) ->
↪  "ixformer.Tensor"
```

参数说明

- input：创建一个和 input 属性相同的 Tensor
- device：Tensor 的 Device

数据类型

All

Shapes

All

2.7.1.5 ixformer.functions.zeros

```
zeros(*size: int, *,
      dtype: "ixformer.DataType" = None,
      device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
      layout: "ixformer.strided" = None,
      requires_grad: bool = False
     ) -> "ixformer.Tensor"
```

参数说明

- `size`：Tensor shape
- `dtype`：Tensor 的数据类型
- `device`：Tensor 的 Device
- `layout`：数据排布
- `requires_grad`：是否需要梯度

数据类型

All

Shapes

All

```
zeros_like(input: "ixformer.Tensor",
           dtype: "ixformer.DataType" = None,
           device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
           layout: "ixformer.strided" = None,
           requires_grad: bool = False
          ) -> "ixformer.Tensor"
```

参数说明

- `input`：创建一个和 `input` 属性相同的 Tensor
- `dtype`：Tensor 的数据类型
- `device`：Tensor 的 Device
- `layout`：数据排布
- `requires_grad`：是否需要梯度

数据类型

All

Shapes

All

2.7.1.6 ixformer.functions.ones

```
ones(shape: List[int],
     dtype: "ixformer.DataType" = None,
     device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
     layout: "ixformer.strided" = None,
     requires_grad: bool = False
    ) -> "ixformer.Tensor"
```

参数说明

- shape：Tensor shape
- dtype：Tensor 的数据类型
- device：Tensor 的 Device
- layout：数据排布
- requires_grad：是否需要梯度

数据类型

All

Shapes

All

```
ones_like(input: "ixformer.Tensor",
          dtype: "ixformer.DataType" = None,
          device: Union[int, str, "ixformer.DeviceType", "ixformer.Device"] = None,
          layout: "ixformer.strided" = None,
          requires_grad: bool = False
         ) -> "ixformer.Tensor"
```

参数说明

- input：创建一个和 input 属性相同的 Tensor
- dtype：Tensor 的数据类型
- device：Tensor 的 Device
- layout：数据排布
- requires_grad：是否需要梯度

数据类型

All

Shapes

All

2.7.1.7 ixformer.functions.split

```
split(tensor: "ixformer.Tensor", split_size_or_sections: Union[int, list], dim: int = 0) ->
↪ List[ixformer.Tensor]
```

参数说明

- tensor：输入的 Tensor
- split_size_or_sections：块的大小，目前只支持整形
- dim：在 Tensor 的 dim 维度上进行切分

数据类型

- half
- float

Shapes

All

说明

目前仅支持分割两块数据。如果需要支持更多，请在源代码 (请向您的应用工程师获取源代码) 中增加一条处理语句即可。

2.7.1.8 ixformer.functions.chunk

```
chunk(tensor: "ixformer.Tensor", chunks: int, dim: int = 0) -> List[ixformer.Tensor]
```

参数说明

- tensor：输入的 Tensor
- chunk：需要分为多少块
- dim：在 Tensor 的 dim 维度上进行切分

数据类型

- half
- float

Shapes

All

2.7.1.9 ixformer.functions.linear

```
linear(input: "ixformer.Tensor",
       weight: "ixformer.Tensor",
       bias: "ixformer.Tensor" = None,
       output: "ixformer.Tensor" = None
      ) -> Tensor
```

支持范围

- input, weight, bias, output 的 dtype 为 float16 且为 contiguous tensor
- input.dim()>=2 weight.dim()==2 bias 为 None 或者 bias.dim()=1
- k 要求是偶数
- 支持传入 output，计算结果将会保存在 output 中

Shapes

- input: [*,k]
- weight: [n,k]
- bias: [n]
- output: [*,n]

2.7.1.10 ixformer.functions.cat

```
cat(tensors: List["ixformer.Tensor"], dim: int = 0, out: "ixformer.Tensor" = None) -> Tensor
```

参数说明

- tensor：需要合并的一组 Tensor
- dim：在 Tensor 的 dim 维度上进行切分
- out：如果传入 out，合并的结果将会存放在 out 中，反之会创建一个新的 Tensor

数据类型

All

Shapes

当前仅支持 Tensors 的数量为 0 ~ 6。如果需要支持更多，请在源代码 (请向您的应用工程师获取源代码) 中增加一条处理语句即可。

2.7.1.11 ixformer.functions.matmul

```
def matmul(input: Tensor,
           other: Tensor, *,
           out: Tensor = None,
           transa: bool = False,
```

```
          transb: bool = False,
          alpha: float = 1.0,
          beta: float = 0.0
         ) -> Tensor
```

参数说明

- input：矩阵 A
- other：矩阵 B
- out：矩阵 C
- transa：是否转置 A
- transb：是否转置 B
- alpha：Gemm 中的 Alpha 系数
- beta：Gemm 中的 Beta 系数

数据类型

half

Shapes

C[..., m, n] = A[..., m, k] * B[..., k, n] * alpha + beta * out

2.7.1.12 ixformer.functions.reshape

```
def reshape(input: Tensor, shape: List[int]) -> Tensor
```

参数说明

- input：输入的 Tensor
- shape：新的 shape

数据类型

All

Shapes

All

说明

reshape 会创建一份新的内存空间，将输入的 Tensor 的值拷贝到新创建的 Tensor 中。

```
def view(input: Tensor, shape: List[int]) -> Tensor
```

参数说明

- input：输入的 Tensor
- shape：新的 shape

数据类型

All

Shapes

All

说明

`view` 不会创建一个新的内存空间，`view` 之后的 Tensor 和输入的 Tensor 内存空间是一致的。

2.7.1.13 ixformer.functions.conv2d

```
conv2d(input: "ixformer.Tensor",
       weight: "ixformer.Tensor",
       bias: "ixformer.Tensor" = None,
       output: "ixformer.Tensor" = None,
       stride: int =1,
       padding: int = 0,
       dilation: int= 1,
       groups: int = 1
      )-> Tensor
```

支持范围

- `input`, `weight`, `bias`, `output` 的 dtype 为 float16 且为 contiguous tensor
- `input` 是 NHWC，`weight` 是 OHWI，`output` 是 NHWC 排布
- 支持传入 `output`，计算结果将会保存在 `output` 中

Shapes

- input：[N,H,W,C]
- weight：[OHWI]
- bias：[n]
- output：[NHWC]

2.7.1.14 ixformer.functions.group_norm

```
def group_norm(input: "ixformer.Tensor",
               num_groups: int,
               weight: "ixformer.Tensor",
               bias: "ixformer.Tensor",
               output: "ixformer.Tensor" = None,
               eps: float = 1e-05,
               format: str = "contiguous_format",
               act_type: int = -1
              )->Tensor
```

支持范围

- input, weight, bias, output 的 dtype 为 float16 且为 contiguous tensor
- NCHW 格式，input shape [N, C, *]，output shape [N,C , *]，支持 3 维和 4 维度的输入
- NHWC 格式，input shape [N, *, C]，output shape [N, *, C]，支持 3 维和 4 维度的输入
- C 必须能被 num_groups 整除
- 支持传入 output，计算结果将会保存在 output 中
- format：”contiguous_format” (NCHW) or ”channels_last” (NHWC)
- act_type：0; identity(), 1: silu

2.7.1.15 ixformer.functions.softmax

```
softmax(input: "ixformer.Tensor", dim=None, _stacklevel=3, dtype=None, output=None) ->
↪  ixformer.Tensor
```

参数说明

- dim：目前仅支持 dim=-1
- _stacklevel：暂未使用
- dtype：暂未使用

2.7.1.16 ixformer.functions.permute

```
permute(input: Tensor, dims: List[int]) -> Tensor
```

参数说明

- input：输入
- dims：维度

数据类型

All

Shapes

All

2.7.1.17 ixformer.functions.transpose

```
transpose(input: Tensor, dim0: int, dim1: int) -> Tensor
```

参数说明

- input：输入

- dim0：维度 0
- dim1：维度 1

数据类型

All

Shapes

All

2.7.1.18 ixformer.functions.contiguous

```
contiguous(input: Tensor) -> Tensor
```

参数说明

- input：输入

数据类型

All

Shapes

All

2.7.2 ixformer.functions (融合算子)

Name	IxFormer API
silu_and_mul	ixformer.functions.silu_and_mul
rms_norm	ixformer.functions.rms_norm
vllm_single_query_cached_kv_attention	ixformer.functions.vllm_single_query_cached_kv_attention
act_bias_mm	ixformer.functions.act_bias_mm
glm_split_qkv	ixformer.functions.glm_split_qkv
gen_rotary_emb_weight	ixformer.functions.gen_rotary_emb_weight
rotary_embedding_2d	ixformer.functions.rotary_embedding_2d
attention_masked_softmax	ixformer.functions.attention_masked_softmax
attention_kv_cache_concat	ixformer.functions.attention_kv_cache_concat
vllm_rotary_embedding_neox	ixformer.functions.vllm_rotary_embedding_neox
vllm_cache_ops_reshape_and_cache	ixformer.functions.vllm_cache_ops_reshape_and_cache

Name	IxFormer API
glm_multi_query_split_qkv	ixformer.functions.glm_multi_query_split_qkv
glm_multi_query_repeat_key_value	ixformer.functions.glm_multi_query_repeat_key_value
flash_attn_varlen_func	ixformer.functions.flash_attn_varlen_func
llama_rotary_embedding	ixformer.functions.llama_rotary_embedding

2.7.2.1 ixformer.functions.silu_and_mul

```
silu_and_mul(input: "ixformer.Tensor",
             output: "ixformer.Tensor" = None
            )->"ixformer.Tensor"
```

数据类型与 Shape

- float16
- float32
- input: [*,2d]
- output: [*,d]

PyTorch 等价实现

```
def ref_silu_and_mul(x: torch.Tensor) -> torch.Tensor:
    x1, x2 = x.chunk(chunks=2, dim=-1)
    return F.silu(x1) * x2
```

2.7.2.2 ixformer.functions.rms_norm

```
rms_norm(input: "ixformer.Tensor",
         weight: "ixformer.Tensor",
         output: "ixformer.Tensor" = None
        )->"ixformer.Tensor"
```

数据类型与 Shape

- float16
- input: [*,H]
- weight: [H]
- output: [*,H]

PyTorch 等价实现

```
def rms_norm(x, weight, eps=1e-6):
    output = x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + eps)
    output = output * weight
    return output
```

2.7.2.3 ixformer.functions.vllm_single_query_cached_kv_attention

```
def vllm_single_query_cached_kv_attention(output: "ixformer.Tensor",
                                          query: "ixformer.Tensor",
                                          key_cache: "ixformer.Tensor",
                                          value_cache: "ixformer.Tensor",
                                          head_mapping: "ixformer.Tensor",
                                          scale: float,
                                          block_tables: "ixformer.Tensor",
                                          context_lens: "ixformer.Tensor",
                                          block_size: int,
                                          max_context_len: int,
                                          alibi_slopes: "ixformer.Tensor" = None
                                         )->None
```

数据类型与 Shape

- `output`, `query`: half, [num_seqs, num_heads, head_size]
- `key_cache`: half, [num_blocks, num_heads, head_size//x, block_size,x]
- `value_cache`: half, [num_blocks, num_heads, head_size, block_size]
- `head_mapping`: int [num_seqs]
- `block_tables`: int [num_seqs, max_num_blocks_per_seq]
- `context_lens`: int [num_seqs] max_num_blocks_per_seq = (max_context_len + block_size - 1) // block_size x = 16 // torch.tensor([], dtype=dtype).element_size() = 8
- `alibi_slopes`: float32, [num_heads]

PyTorch 等价实现

```
MAX_SEQ_LEN = 16384

def get_alibi_slopes(total_num_heads: int) -> torch.Tensor:
   closest_power_of_2 = 2 ** math.floor(math.log2(total_num_heads))
   base = torch.tensor(
      2 ** (-(2 ** -(math.log2(closest_power_of_2) - 3))),
      dtype=torch.float32,
   )
   powers = torch.arange(1, 1 + closest_power_of_2, dtype=torch.int32)
   slopes = torch.pow(base, powers)
```

```
    if closest_power_of_2 != total_num_heads:
        extra_base = torch.tensor(
            2 ** (-(2 ** -(math.log2(2 * closest_power_of_2) - 3))),
            dtype=torch.float32,
        )
        num_remaining_heads = min(
            closest_power_of_2, total_num_heads - closest_power_of_2
        )
        extra_powers = torch.arange(
            start=1, end=1 + 2 * num_remaining_heads, step=2, dtype=torch.int32
        )
        slopes = torch.cat([slopes, torch.pow(extra_base, extra_powers)], dim=0)
    return slopes


def get_alibi_mask(num_heads, seqlen, device, dtype):
    causal_mask = torch.triu(
        torch.ones(seqlen, seqlen, dtype=torch.bool, device=device), 1
    )
    x = (
        torch.arange(0, seqlen, device=device, dtype=torch.float32)
        .view(-1, 1)
        .repeat(1, seqlen)
    )
    y = (
        torch.arange(0, seqlen, device=device, dtype=torch.float32)
        .view(1, -1)
        .repeat(seqlen, 1)
    )
    offsets = (y - x).view(1, seqlen, seqlen)
    slopes = (
        get_alibi_slopes(num_heads).to(device).to(torch.float32).view(num_heads, 1, 1)
    )
    mask = offsets * slopes
    mask.masked_fill_(causal_mask, float("-inf"))
    return mask.to(dtype)


def ref_masked_attention(
    query: torch.Tensor,
    key: torch.Tensor,
    value: torch.Tensor,
    scale: float,
    attn_mask: Optional[torch.Tensor] = None,
) -> torch.Tensor:
    query = query * scale
```

```
    attn = torch.einsum("qhd,khd->hqk", query, key)
    if attn_mask is not None:
        attn = attn + attn_mask
    attn = torch.softmax(attn, dim=-1)
    out = torch.einsum("hqk,khd->qhd", attn, value)
    return out


def ref_single_query_cached_kv_attention(
    output: torch.Tensor,
    query: torch.Tensor,
    key_cache: torch.Tensor,
    value_cache: torch.Tensor,
    num_q_per_kv: int, # num_q_per_kv 提供 head_mapping 功能
    scale,
    block_tables: torch.Tensor,
    context_lens: torch.Tensor,
    block_size,
    # max_context_len not used in pytorch
    use_alibi: bool, # 内部构造，提供和 alibi_slopes 一样的功能
) -> None:
    num_query_heads = query.shape[1]
    num_kv_heads = value_cache.shape[1]
    head_size = value_cache.shape[2]
    num_input_tokens = query.shape[0]

    attn_mask = get_alibi_mask(
        num_query_heads, MAX_SEQ_LEN, output.device, output.dtype
    )
    for i in range(num_input_tokens):
        q = query[i].unsqueeze(0)
        block_table = block_tables[i]
        context_len = int(context_lens[i])

        keys = []
        values = []
        for j in range(context_len):
            block_number = int(block_table[j // block_size])
            block_offset = j % block_size

            k = key_cache[block_number, :, :, block_offset, :]
            k = k.reshape(num_kv_heads, head_size)
            keys.append(k)

            v = value_cache[block_number, :, :, block_offset]
            values.append(v)
```

```
        keys = torch.stack(keys, dim=0)
        values = torch.stack(values, dim=0)
        if num_q_per_kv > 1:
            keys = torch.repeat_interleave(keys, num_q_per_kv, dim=1)
            values = torch.repeat_interleave(values, num_q_per_kv, dim=1)
        out = ref_masked_attention(
            q,
            keys,
            values,
            scale,
            attn_mask[:, context_len - 1, :context_len][:, None, :]
            if use_alibi
            else None,
        )
        out = out.view(num_query_heads, head_size)
        output[i].copy_(out, non_blocking=True)
```

2.7.2.4 ixformer.functions.act_bias_mm

```
act_bias_mm(
    mat1: "ixformer.Tensor",
    mat2: "ixformer.Tensor",
    bias: "ixformer.Tensor" = None,
    output: "ixformer.Tensor" = None,
    scale: float = 1,
    act_type: str = "none",
    trans_format: str = "NN",
)
```

说明

对应于 ixInfer 中的 cuinferCustomGemm。

数据类型与 Shape

- `mat1`: float16 [*,m,k]
- `mat2`: float16 [*,n,k] if "TN" else [*,k,n]
- `bias`: float16 [n]
- `output`: float16 [*,m,n]
- `act_type`: "none"、"gelu"、"relu"、"silu"
- `trans_format`: "TN" 或"NN"

PyTorch 等价实现

```
# 模式 1: gemm, 没有 activation 及 bias
output = (mat1 @ mat2) * scale

# 模式 2: gemm + bias
output = (mat1 @ mat2) * scale + bias

# 模式 3: gemm + bias + activation
output = activation((mat1 @ mat2) * scale + bias)
```

2.7.2.5 ixformer.functions.glm_split_qkv

```
def glm_split_qkv(
    ori_qkv: "ixformer.Tensor",
    batch_size: int,
    head_num: int,
    head_dim: int,
    ori_seq_len: int,
    new_seq_len: int,
    new_qkv: List["ixformer.Tensor"] = None
)-> "ixformer.Tensor","ixformer.Tensor","ixformer.Tensor"
```

说明

GLM 中的切分 qkv，与传统的 bert，gpt2 等不相同。实现功能：

1. 将 ori_qkv(batch_size,seq_len,head_num*head_dim*3) 切分为 q,k,v(batch_size,seq_len,head_num*head_dim)
2. 将 q,k,v 进行维度调整 [batch_size,seq_len,head_num,head_dim] -> [batch_size,head_num,seq_len,head_dim]
3. 将 q,k,v 进行 padding。为了适应 flash-attention 的维度要求 [batch_size,head_num,seq_len,head_dim] -> [batch_size,head_num,new_seq_len,head_dim]

数据类型与 Shape

- ori_qkv: float16, [batch_size, ori_seq_len, head_num*head_dim*3]
- new_qkv: [new_q,new_k,new_v]

PyTorch 等价实现

```
def glm_split_qkv_base(qkv, head_num, head_dim, new_q, new_k, new_v):
    batch_size = qkv.shape[0]
    seq_len = qkv.shape[1]
    # GLM 的 qkv 切分方式与 GPT, BERT 等结构不一致 view(batch_size,seq_len,3,head_num*head_dim)
    qkv = qkv.view(batch_size, seq_len, head_num, 3 * head_dim)
    q, k, v = torch.chunk(qkv, 3, dim=-1)
    # batch_size,seq_len,head_num,head_dim -> batch_size,head_num,seq_len,head_dim
    q = q.permute(0, 2, 1, 3).contiguous()
    k = k.permute(0, 2, 1, 3).contiguous()
```

```
    v = v.permute(0, 2, 1, 3).contiguous()
    new_q[:, :, :seq_len, :] = q
    new_k[:, :, :seq_len, :] = k
    new_v[:, :, :seq_len, :] = v
    return new_q, new_k, new_v
```

2.7.2.6 ixformer.functions.gen_rotary_emb_weight

```
gen_rotary_emb_weight(
       cos_emb: "ixformer.Tensor",
       sin_emb: "ixformer.Tensor",
       dim_size: int,
       max_pos_len: int,
       base: float,
)->None
```

说明

ChatGLM-6B RotaryEmbedding 初始化参数，learnable 模式暂未实现。

数据类型与 Shape

- cos_emb: float16, [seq_len,dim_size/2]
- sin_emb: float16, [seq_len,dim_size/2]

PyTorch 等价实现

```
def glm_rotary_base(dim_size, max_pos_len, base, dtype):
    inv_freq = 1.0 / \
       (base ** (torch.arange(0, dim_size, 2).float() / dim_size))
    inv_freq = inv_freq.cuda().float()
    t = torch.arange(max_pos_len, device=inv_freq.device, dtype=inv_freq.dtype)
    freqs = torch.einsum("i,j->ij", t, inv_freq)
    emb = torch.cat((freqs, freqs), dim=-1).to(inv_freq.device)
    cos_cached = emb.cos().to(dtype)
    sin_cached = emb.sin().to(dtype)
    return cos_cached, sin_cached
```

2.7.2.7 ixformer.functions.rotary_embedding_2d

```
def rotary_embedding_2d(
    q: "ixformer.Tensor",
    k: "ixformer.Tensor",
```

```
    cos: "ixformer.Tensor",
    sin: "ixformer.Tensor",
    position_ids: "ixformer.Tensor",
    new_q: "ixformer.Tensor" = None,
    new_k: "ixformer.Tensor" = None,
    shape: List[int] = None,
)
```

说明

ChatGLM 中，position_encoding_2d 对应的 rotary_embedding。本质上是将两个 q,k 的 head_dim 维度切分为 head_dim/2，再分别进行 rotary_embedding。

数据类型与 Shape

- q: (ixformer.Tensor,float16),[batch_size,head_num,seq_len,head_dim]
- k: (ixformer.Tensor,float16),[batch_size,head_num,seq_len,head_dim]
- cos: (ixformer.Tensor,float16),[max_seq_len,head_dim / 2]
- sin: (ixformer.Tensor,float16),[max_seq_len,head_dim / 2]
- new_q: (ixformer.Tensor,float16),[batch_size,head_num,seq_len,head_dim]
- new_k: (ixformer.Tensor,float16),[batch_size,head_num,seq_len,head_dim]
- position_ids: (ixformer.Tensor,int32),[batch_size,2,seq_len]
- shape: 如果指定了 q, k, new_q, new_k 的四个维度，q, k, new_q, new_k 将不会进行维度检查，但是需要保证 q, k, new_q, new_k 的元素个数不小于 batch_size*head_num*seq_len*head_dim

PyTorch 等价实现

```
def rotate_half(x):
    x1, x2 = x[..., : x.shape[-1] // 2], x[..., x.shape[-1] // 2:]
    return torch.cat(
       (-x2, x1), dim=x1.ndim - 1
    )  # dim=-1 triggers a bug in earlier torch versions


def apply_rotary_pos_emb_index(q, k, cos, sin, position_id):
    # position_id: [batch_size,seq_len], q, k: [batch_size,head_num, seq_len, head_dim], cos:
    ↪  [sq, 1, hn] -> [batch_size,1,seq_len,head_dim]
    cos, sin = NNF.embedding(position_id, cos.squeeze(1)).unsqueeze(1), NNF.embedding(
       position_id, sin.squeeze(1)
    ).unsqueeze(1)
    q, k = (q * cos) + (rotate_half(q) * sin), (k * cos) + \
       (rotate_half(k) * sin)
    return q, k


def rotary_emb_2d(query_layer, key_layer, sin, cos, position_ids):
    q1, q2 = torch.chunk(query_layer, 2, dim=-1)
    k1, k2 = torch.chunk(key_layer, 2, dim=-1)
```

```
    position_ids, block_position_ids = (
       position_ids[:, 0, :].contiguous(),
       position_ids[:, 1, :].contiguous(),
    )

    q1, k1 = apply_rotary_pos_emb_index(q1, k1, cos, sin, position_ids)
    q2, k2 = apply_rotary_pos_emb_index(q2, k2, cos, sin, block_position_ids)

    query_layer = torch.concat([q1, q2], dim=-1)
    key_layer = torch.concat([k1, k2], dim=-1)

    return query_layer, key_layer
```

2.7.2.8 ixformer.functions.attention_masked_softmax

```
attention_masked_softmax(
   input: "ixformer.Tensor",
   mask: "ixformer.Tensor",
   output: "ixformer.Tensor" = None
)
# attention 中使用的 masked softmax
```

数据类型与 Shape

- `input`: float16, [batch_size, head_num, tgt_len, src_len]
- `mask`: int32, [mask_batch_size, mask_head_num, mask_tgt_len, mask_src_len] mask 会做 broadcast
- `output`: float16, [batch_size, head_num, tgt_len, src_len]

PyTorch 等价实现

```
def pt_masked_softmax(inputs, mask):
    assert len(mask.shape) == 4
    inputs = torch.masked_fill(inputs, mask.bool(),-10000)
    return F.softmax(inputs, dim=-1)
```

2.7.2.9 ixformer.functions.attention_kv_cache_concat

```
attention_kv_cache_concat(
   past_key: "ixformer.Tensor",
   past_value: "ixformer.Tensor",
   key_layer: "ixformer.Tensor",
```

```
    value_layer: "ixformer.Tensor",
    new_key: "ixformer.Tensor" = None,
    new_value: "ixformer.Tensor" = None,
    shape: List[int] = None,
)
# attention 中使用的 kv-cache concat
```

数据类型与 Shape

- past_key, past_value: float16, [batch_size, head_num, past_seq_len, head_dim]
- key_layer, value_layer: float16, [batch_size, head_num, key_value_seq_len, head_dim]
- new_key, new_value: float16, [batch_size, head_num, past_seq_len+key_value_seq_len, head_dim]
- shape: list, len(shape)==5 batch_size, head_num, past_seq_len, head_dim 如果传入，将按照 shape 的大小进行计算，而对 past_key, past_value, new_key 和 new_value 不做 shape 检查

PyTorch 等价实现

```
def pt_kv_cache_concat(past_key, past_value, key, value):
    new_key = torch.concat([past_key, key], dim=2)
    new_value = torch.concat([past_value, value], dim=2)
    return new_key, new_value
```

2.7.2.10 ixformer.functions.vllm_rotary_embedding_neox

```
vllm_rotary_embedding_neox(
    positions: "ixformer.Tensor",
    query: "ixformer.Tensor",
    key: "ixformer.Tensor",
    head_size: int,
    cos_sin_cache: "ixformer.Tensor",
    is_neox: bool = True,
)
# vLLM 中使用的 rotary_embedding_neox
```

说明

支持 GPT-J 和 GPT-NeoX 两种类型的位置编码。

数据类型与 Shape

- positions: int64 [num_tokens]
- query, key: float16, float32 [num_tokens, num_heads, head_size]
- cos_sin_cache: float16, [max_position, 2, rot_dim // 2]
- head_size: int

PyTorch 等价实现

```
def rotate_half(x: torch.Tensor) -> torch.Tensor:
 x1 = x[..., : x.shape[-1] // 2]
 x2 = x[..., x.shape[-1] // 2 :]
 return torch.cat((-x2, x1), dim=-1)


def rotate_gptj(x: torch.Tensor) -> torch.Tensor:
   x1 = x[..., ::2]
   x2 = x[..., 1::2]
   x = torch.stack((-x2, x1), dim=-1)
   return x.flatten(-2)


def apply_rotary_pos_emb(
   q: torch.Tensor,
   k: torch.Tensor,
   cos: torch.Tensor,
   sin: torch.Tensor,
   is_neox: bool = True,
) -> Tuple[torch.Tensor, torch.Tensor]:
   dtype = q.dtype
   fn = rotate_half if is_neox else rotate_gptj
   q_embed = (q.float() * cos.float()) + (fn(q.float()) * sin.float())
   k_embed = (k.float() * cos.float()) + (fn(k.float()) * sin.float())
   return q_embed.to(dtype), k_embed.to(dtype)


class RefRotaryEmbeddingNeox(nn.Module):

   def __init__(
      self,
      dim: int,
      max_position_embeddings: int = 2048,
      base: int = 10000,
      is_neox: bool = True,
   ) -> None:
      super().__init__()
      self.rotary_dim = dim
      self.max_position_embeddings = max_position_embeddings

      # Create cos and sin embeddings.
      inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2) / dim))
      t = torch.arange(max_position_embeddings).float()
      freqs = torch.einsum("i,j->ij", t, inv_freq.float())
      if is_neox:
         emb = torch.cat((freqs, freqs), dim=-1)
```

```
        else:
            emb = torch.repeat_interleave(freqs, 2, -1)
        cos = emb.cos().to(dtype=inv_freq.dtype)
        sin = emb.sin().to(dtype=inv_freq.dtype)
        self.register_buffer("cos_cached", cos, persistent=False)
        self.register_buffer("sin_cached", sin, persistent=False)
        self.is_neox = is_neox

    def forward(
        self,
        positions: torch.Tensor,  # [num_tokens]
        query: torch.Tensor,  # [num_tokens, num_heads, head_size]
        key: torch.Tensor,  # [num_tokens, num_heads, head_size]
    ) -> Tuple[torch.Tensor, torch.Tensor]:

        query_rot = query[..., : self.rotary_dim]
        query_pass = query[..., self.rotary_dim :]
        key_rot = key[..., : self.rotary_dim]
        key_pass = key[..., self.rotary_dim :]

        query_rot = query_rot.transpose(0, 1)
        key_rot = key_rot.transpose(0, 1)
        cos = F.embedding(positions, self.cos_cached)
        sin = F.embedding(positions, self.sin_cached)
        query_rot, key_rot = apply_rotary_pos_emb(
            query_rot, key_rot, cos, sin, self.is_neox
        )
        query_rot = query_rot.transpose(0, 1).contiguous()
        key_rot = key_rot.transpose(0, 1).contiguous()

        query = torch.cat((query_rot, query_pass), dim=-1)
        key = torch.cat((key_rot, key_pass), dim=-1)

        # Output query/key shape: [num_tokens, num_tokens, head_size]
        return query, key
```

2.7.2.11 ixformer.functions.vllm_cache_ops_reshape_and_cache

```
vllm_cache_ops_reshape_and_cache(
    key: "ixformer.Tensor",
    value: "ixformer.Tensor",
    key_cache: "ixformer.Tensor",
    value_cache: "ixformer.Tensor",
    slot_mapping: "ixformer.Tensor",
```

```
)
# vLLM 中使用的 reshape_and_cache
```

数据类型与 Shape

- key, value: float16, float32 [num_tokens, num_heads, head_size]
- key_cache: float16, float32 [num_blocks, num_heads, head_size/x, block_size, x]
- value_cache: float16, float32 [num_blocks, num_heads, head_size, block_size]
- slot_mapping: int32 [num_tokens]

PyTorch 等价实现

```
def ref_reshape_and_cache(key, value, key_cache, value_cache, slot_mapping,block_size,dtype:
↪ torch.dtype):
    num_tokens = key.shape[0]
    num_heads = key.shape[1]
    head_size = key.shape[2]
    x = 16 // torch.tensor([], dtype=dtype).element_size()
    for i in range(num_tokens):
        reshaped_key = key.reshape(num_tokens, num_heads, head_size // x, x)
        block_idx = torch.div(slot_mapping[i], block_size, rounding_mode="floor")
        block_offset = slot_mapping[i] % block_size
        key_cache[block_idx, :, :, block_offset, :] = reshaped_key[i]
        value_cache[block_idx, :, :, block_offset] = value[i]
```

2.7.2.12 ixformer.functions.glm_multi_query_split_qkv

```
glm_multi_query_split_qkv(
    ori_qkv: "ixformer.Tensor",
    batch_size: int,
    head_num: int,
    kv_head_num: int,
    head_dim: int,
    ori_seq_len: int,
    new_seq_len: int,
    new_qkv: List["ixformer.Tensor"] = None,
)-> "ixformer.Tensor","ixformer.Tensor","ixformer.Tensor"
```

说明

ChatGLM-6B 中的切分 qkv，采用了 multi-query attention。实现功能：

1. 将 ori_qkv(batch_size,seq_len,(head_num+kv_head_dim*2)*head_dim) 切分为 q,k,v

   (batch_size,seq_len,head_num*head_dim, batch_size,seq_len, kv_head_num*head_dim, batch_size, seq_len, kv_head_num*head_dim)

2. 将 q,k,v 进行维度调整 [batch_size,seq_len,head_num,head_dim] -> [batch_size,head_num,seq_len,head_dim]
3. 将 q,k,v 进行 padding。为了适应 flash-attention 的维度要求 [batch_size,head_num,seq_len,head_dim] -> [batch_size,head_num,new_seq_len,head_dim]

数据类型与 Shape

- ori_qkv: float16, [batch_size, ori_seq_len, (head_num+kv_head_num*2)*head_dim]
- new_qkv: [new_q,new_k,new_v]

PyTorch 等价实现

```
def glm_multi_query_qkv_split(qkv, head_num, kv_head_num, head_dim, new_q, new_k, new_v):
    batch_size = qkv.shape[0]
    seq_len = qkv.shape[1]
    assert qkv.size(2) == (head_num + kv_head_num*2) * head_dim
    # batch_size, seq_len, head_num*head_dim
    (query_layer, key_layer, value_layer) = qkv.split(
                [
                   head_num * head_dim,
                   kv_head_num * head_dim,
                   kv_head_num * head_dim,
                ],
                dim=-1,
            )
    query_layer = query_layer.view(batch_size,seq_len,head_num,head_dim)
    key_layer = key_layer.view(batch_size,seq_len,kv_head_num,head_dim)
    value_layer = value_layer.view(batch_size,seq_len,kv_head_num,head_dim)

    query_layer = query_layer.permute(0, 2, 1, 3).contiguous()
    key_layer = key_layer.permute(0, 2, 1, 3).contiguous()
    value_layer = value_layer.permute(0, 2, 1, 3).contiguous()

    new_q[:, :, :seq_len, :] = query_layer
    new_k[:, :, :seq_len, :] = key_layer
    new_v[:, :, :seq_len, :] = value_layer
    return new_q, new_k, new_v
```

2.7.2.13 ixformer.functions.glm_multi_query_repeat_key_value

```
glm_multi_query_repeat_key_value(
    key: "ixformer.Tensor", value: "ixformer.Tensor", head_num: int
)
# ChatGLM2-6B 中对 key, value 在 head_num 维度的复制。
```

数据类型与 Shape

- key, value: float16, [batch_size, kv_head_num, seq_len, head_dim]
- new_key, new_value: float16 [batch_size, head_num, seq_len, head_dim]

PyTorch 等价实现

```
def pt_repeat_key_value(key, value, head_num, new_key, new_value):
    batch_size, kv_head_num, seq_len, head_dim = value.shape
    new_shape = [batch_size, head_num, seq_len, head_dim]
    assert key.size(1) == kv_head_num

    key = key.unsqueeze(2)
    key = key.expand(
                -1, -1, head_num // kv_head_num, -1, -1
            )

    value = value.unsqueeze(2)
    value = value.expand(
                -1, -1, head_num // kv_head_num, -1, -1
            )

    key = key.contiguous().view(*new_shape)
    value = value.contiguous().view(*new_shape)

    new_key[:,:,:seq_len,:] = key
    new_value[:,:,:seq_len,:] = value

    return new_key, new_value
```

2.7.2.14 ixformer.functions.flash_attn_varlen_func

```
def flash_attn_varlen_func(q, k, v, cu_seqlens_q, cu_seqlens_k, max_seqlen_q, max_seqlen_k,
                           dropout_p=0.0, softmax_scale=None, causal=False,
                           return_attn_probs=False):
    """dropout_p should be set to 0.0 during evaluation
    Supports multi-query and grouped-query attention (MQA/GQA) by passing in K, V with fewer
↪   heads
    than Q. Note that the number of heads in Q must be divisible by the number of heads in KV.
    For example, if Q has 6 heads and K, V have 2 heads, head 0, 1, 2 of Q will attention to head
    0 of K, V, and head 3, 4, 5 of Q will attention to head 1 of K, V.
```

数据类型与 Shape

- q: (total_q, nheads, headdim), where total_q = total number of query tokens in the batch.
- k: (total_k, nheads_k, headdim), where total_k = total number of key tokens in the batch.
- v: (total_k, nheads_k, headdim), where total_k = total number of key tokens in the batch.

- cu_seqlens_q: (batch_size + 1,), dtype torch.int32. The cumulative sequence lengths of the sequences in the batch, used to index into q.
- cu_seqlens_k: (batch_size + 1,), dtype torch.int32. The cumulative sequence lengths of the sequences in the batch, used to index into kv.
- max_seqlen_q: int. Maximum query sequence length in the batch.
- max_seqlen_k: int. Maximum key sequence length in the batch.
- dropout_p: float. Dropout probability.
- softmax_scale: float. The scaling of QKˆT before applying softmax. Default to 1 / sqrt(headdim).
- causal: bool. Whether to apply causal attention mask (e.g., for auto-regressive modeling).
- return_attn_probs: bool. Whether to return the attention probabilities. This option is for testing only. The returned probabilities are not guaranteed to be correct (they might not have the right scaling).

返回值

- out: (total, nheads, headdim).
- softmax_lse [optional, if return_attn_probs=True]: (batch_size, nheads, seqlen). The logsumexp of each row of the matrix QKˆT * scaling (e.g., log of the softmax normalization factor).
- S_dmask [optional, if return_attn_probs=True]: (batch_size, nheads, seqlen, seqlen). The output of softmax (possibly with different scaling). It also encodes the dropout pattern (negative means that location was dropped, nonnegative means it was kept).

PyTorch 等价实现

```
def ref_masked_attention(
    query: torch.Tensor,
    key: torch.Tensor,
    value: torch.Tensor,
    scale: float,
    attn_mask=None,
) -> torch.Tensor:
    query = query * scale
    attn = torch.einsum("qhd,khd->hqk", query, key)
    if attn_mask is not None:
        attn = attn + attn_mask
    attn = torch.softmax(attn, dim=-1)
    out = torch.einsum("hqk,khd->qhd", attn, value)
    return out


# [tokens,head_num_kv,head_dim] ->[tokens,head_num,head_dim] ,group attention
def repeat_kv(x: torch.Tensor, n_rep: int) -> torch.Tensor:
    """torch.repeat_interleave(x, dim=2, repeats=n_rep)"""
    tokens, n_kv_heads, head_dim = x.shape
    if n_rep == 1:
        return x
    return (
        x[:, :, None, :]
```

```
        .expand(tokens, n_kv_heads, n_rep, head_dim)
        .reshape(tokens, n_kv_heads * n_rep, head_dim)
    )


def unpad_causal_torch(
    q,
    k,
    v,
    output,
    cu_seqlens_q,
    cu_seqlens_k,
    max_seq_len_q,
    max_seq_len_kv,
    dtype,
    atten_scale,
    is_causal=True,
):

    head_num = q.size(1)
    head_num_kv = k.size(1)
    head_dim = q.size(2)

    assert head_num % head_num_kv == 0

    # tokens,head_num,head_dim
    if head_num != head_num_kv:

        k = repeat_kv(k, head_num // head_num_kv)  # [0,0,0,0,1,1,1,1,2,2,2,2] GROUP
        v = repeat_kv(v, head_num // head_num_kv)

    batch_size = cu_seqlens_q.size(0) - 1

    for i in range(batch_size):
        q_start_index = cu_seqlens_q[i]
        q_end_index = cu_seqlens_q[i + 1]
        cur_q_len = q_end_index - q_start_index
        # 1*seq_len,head_num,head_dim
        cur_q = q[q_start_index:q_end_index]

        k_start_index = cu_seqlens_k[i]
        k_end_index = cu_seqlens_k[i + 1]
        cur_k_len = k_end_index - k_start_index

        cur_k = k[k_start_index:k_end_index]
        cur_v = v[k_start_index:k_end_index]
```

```
        # mask = torch.tril(torch.ones([cur_q_len, cur_k_len], dtype=torch.bool)).cuda()
        # mask = mask.unsqueeze(0).unsqueeze(0)
        if is_causal:
            # Create attention mask.
            attn_mask = torch.triu(
               torch.ones(cur_q_len, cur_k_len, dtype=dtype), diagonal=1
            )
            attn_mask = attn_mask * torch.finfo(dtype).min
            attn_mask = attn_mask.to(dtype=dtype, device="cuda")
        else:
            attn_mask = None

        ref_output = ref_masked_attention(
            cur_q,
            cur_k,
            cur_v,
            atten_scale,
            attn_mask=attn_mask,
        )
        output[q_start_index:q_end_index].copy_(ref_output)

def warpper_torch(
    q,
    k,
    v,
    cu_seqlens_q,
    cu_seqlens_k,
    max_seq_len_q,
    max_seq_len_kv,
    is_causal,
    atten_scale,
):
    output_pt = torch.zeros_like(q)
    unpad_causal_torch(
        q,
        k,
        v,
        output_pt,
        cu_seqlens_q,
        cu_seqlens_k,
        max_seq_len_q,
        max_seq_len_kv,
        torch.float16,
        atten_scale,
        is_causal,
```

```
    )
    return output_pt
```

2.7.2.15 ixformer.functions.llama_rotary_embedding

```
def llama_rotary_embedding(
    q: "ixformer.Tensor",
    k: "ixformer.Tensor",
    sin: "ixformer.Tensor",
    cos: "ixformer.Tensor",
    pos_ids: "ixformer.Tensor",
    new_q: "ixformer.Tensor"=None,
    new_k: "ixformer.Tensor"=None,
)
```

数据类型与 Shape

- q, k: float16, [batch_size, head_num, seq_len, head_dim]
- new_q, new_k: float16, [batch_size, head_num, seq_len, head_dim]
- sin, cos: float16, [...,max_seq_len,head_dim]
- pos_ids: int64, [batch_size,seq_len]
- head_dim 目前只支持 128 和 256

说明

如果 new_q==q, new_k==k，将对 q, k 使用 inplace 操作

PyTorch 等价实现

```
def rotate_half(x):
    """Rotates half the hidden dims of the input."""
    x1 = x[..., : x.shape[-1] // 2]
    x2 = x[..., x.shape[-1] // 2 :]
    return torch.cat((-x2, x1), dim=-1)


def apply_rotary_pos_emb(q, k, cos, sin, position_ids):
    dtype = q.dtype
    q = q.float()
    k = k.float()
    cos = cos.float()
    sin = sin.float()
    # The first two dimensions of cos and sin are always 1, so we can `squeeze` them.
    cos = cos.squeeze(1).squeeze(0)  # [seq_len, dim]
    sin = sin.squeeze(1).squeeze(0)  # [seq_len, dim]
    cos = cos[position_ids].unsqueeze(1)  # [bs, 1, seq_len, dim]
```

```
    sin = sin[position_ids].unsqueeze(1)  # [bs, 1, seq_len, dim]
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed.to(dtype), k_embed.to(dtype)
```

3 商标声明

- 天数智芯、天数智芯 logo、Iluvatar CoreX 等商标、标识、组合商标为上海天数智芯半导体有限公司之注册商标或商标，受法律保护。
- 除了天数智芯的注册商标外，本内容中使用的其他产品名称及标志仅用于识别目的，该等名称及标志可能是归属于其各自公司的商标。我们否认对该等名称及标志的所有权利。
- CentOS 标识为 Red Hat 公司的商标。
- Docker 为 Docker 公司在美国和其他国家的商标或注册商标。
- Linux 为 Linus Torvalds 在美国和其它国家的注册商标。
- NVIDIA 和 CUDA 为 NVIDIA 公司在美国和/或其它国家的商标和/或注册商标。
- PyTorch 为 Facebook 公司的商标。
- TensorFlow 为 Google 公司的商标。
- Ubuntu 为 Canonical 公司的注册商标。