AlphafoId3在CPU上的推理优化探索经验分享

上海交通大学 桂仲腾

2025.8.23

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录

1. 从 Jax 迁移到 PyTorch

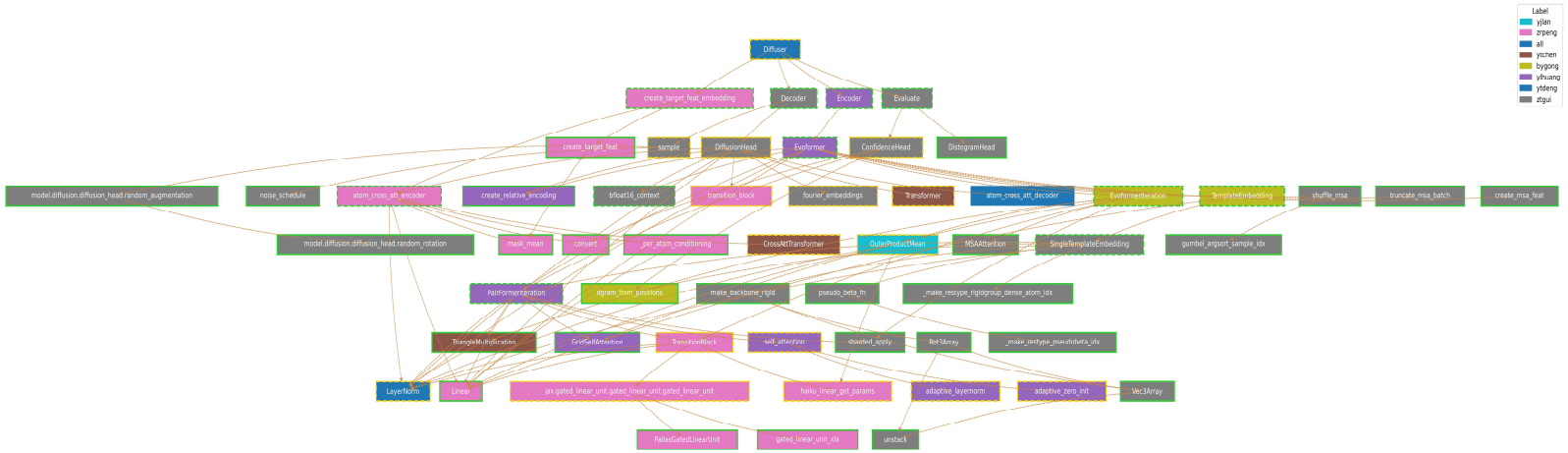
动机

- Jax 版本迁移至 CPU 后运行效率较低
- Jax 版本大量使用 JIT 导致难以进行 profiling

尝试

- 自顶向下迁移
- 使用 dlpack 转换 jax.ndarray 与 torch.Tensor

结果：jax.vmap 等涉及并行/批量处理的部分无法轻量级转换，转向使用现成 xfold 项目



2. 对 xfold 进行 profiling

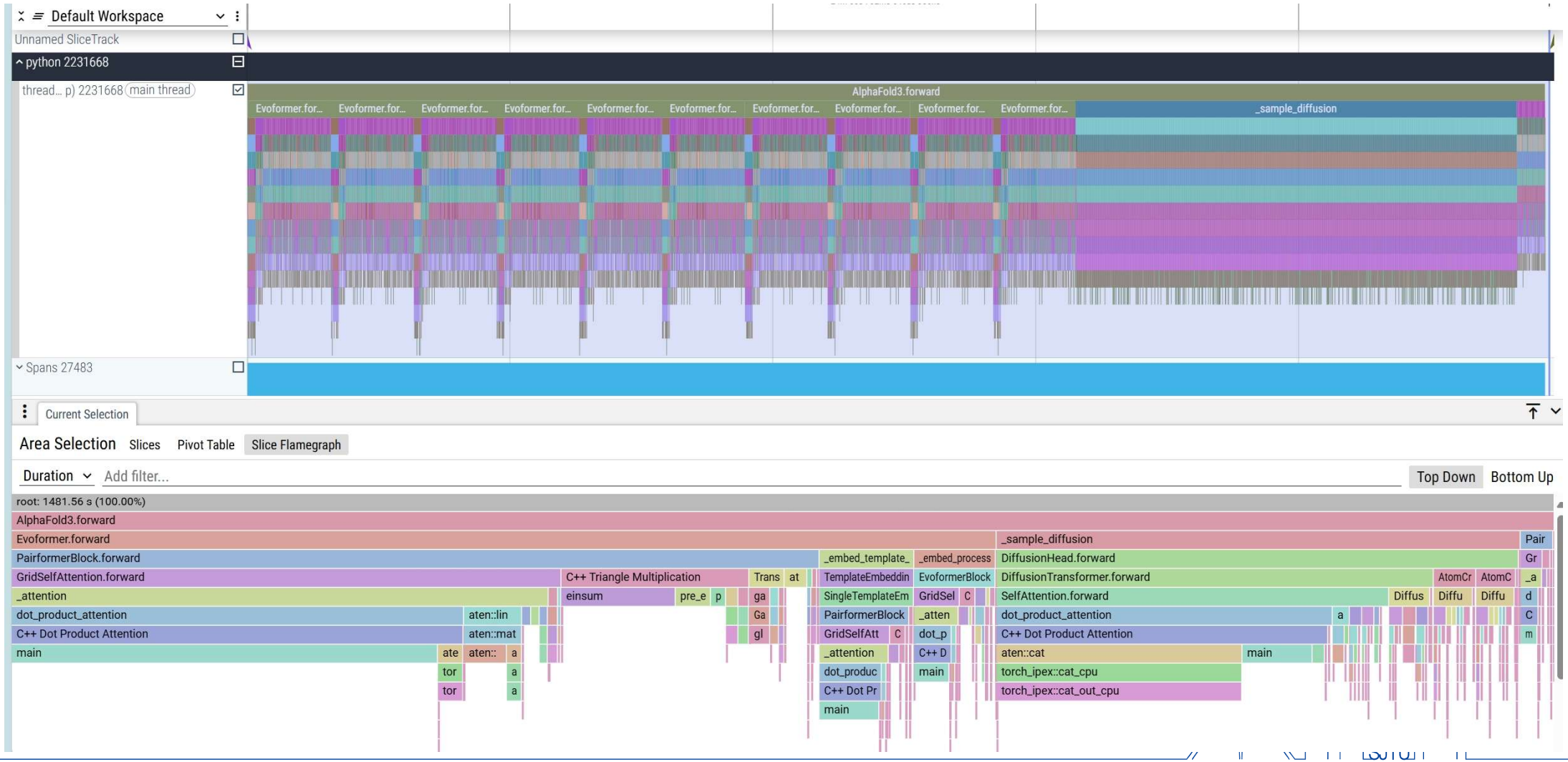
- 使用 torch.profiler 进行数据采集
- 使用 Perfetto 进行数据可视化

```python
def profile(name: Optional[str] = None):
    def decorator(func):
        if DO_PROFILE:

            @wraps(func)
            def wrapper(*args, **kwargs):
                with torch.profiler.record_function(func.__name__ if name is None else name):
                    return func(*args, **kwargs)

            return wrapper
        else:
            return func

    return decorator
```

小技巧

- 使用装饰器语法糖简化函数/模块染色
- 过滤耗时极短的事件，大幅缩减 profile 文件大小

2. 对 xfold 进行 profiling



3. 基于 profiling 进行优化

torch.to 耗时占比高

发现问题来源于 torch.amp.autocast，其存储数据采用的是 fp32，

每次计算时动态将数据转为 bf16，浪费大量时间

手动在开头将所有 fp32 转为 bf16，避免运行中重复转换

```
-   with torch.amp.autocast(device_type="cuda", dtype=torch.
    bfloat16):
+           featurised_example = {
+               k: v.to(torch.bfloat16) if v.dtype == torch.float32 else
                    v for k, v in featurised_example.items()
+           }
+           self._model.to(dtype=torch.bfloat16)
```

```
36  -   broadcast_factor = 1.0
37  +   broadcast_factor = torch.Tensor([1.0]).to(value.dtype)
```

3. 基于 profiling 进行优化

Diffusion 部分的 `_conditioning` 函数，随数据规模增大耗时占比大幅提升

↓

发现此处进行的大量计算均是对每次迭代中的常量进行重复计算

↓

第一次计算完毕后将结果进行缓存，之后直接复用

3. 基于 profiling 进行优化


```
class DiffusionHead(nn.Module):
    def __init__(self):
        self.fourier_embeddings = FourierEmbeddings(dim=256)




    @profile()
    def _conditioning(
        self,
        batch,
        embeddings: dict[str, torch.Tensor],
        noise_level: torch.Tensor,
        use_conditioning: bool,
    ) -> tuple[torch.Tensor, torch.Tensor]:


        single_embedding = use_conditioning * embeddings['single']
        pair_embedding = use_conditioning * embeddings['pair']

        rel_features = featurization.create_relative_encoding(
            batch.token_features, max_relative_idx=32,
            max_relative_chain=2
        ).to(dtype=pair_embedding.dtype)
        features_2d = torch.concatenate([pair_embedding, rel_features],
        dim=-1)

        pair_cond = self.pair_cond_initial_projection(
            self.pair_cond_initial_norm(features_2d)
        )

        pair_cond += self.pair_transition_0(pair_cond)
        pair_cond += self.pair_transition_1(pair_cond)

        target_feat = embeddings['target_feat']
        features_1d = torch.concatenate(
            [single_embedding, target_feat], dim=-1)
        single_cond = self.single_cond_initial_projection(
            self.single_cond_initial_norm(features_1d))




        noise_embedding = self.fourier_embeddings(
```

```
92  class DiffusionHead(nn.Module):
93      def __init__(self):
144         self.fourier_embeddings = FourierEmbeddings(dim=256)
145
146+         self.first_run = True
147+         self.single_cond = None
148+         self.pair_cond = None
149+
150     @profile()
151     def _conditioning(
152         self,
153+        batch,  # constant
154+        embeddings: dict[str, torch.Tensor],  # constant
155+        noise_level: torch.Tensor,  # variable
156+        use_conditioning: bool,  # True
157     ) -> tuple[torch.Tensor, torch.Tensor]:
158+        if self.first_run:
159+            self.first_run = False
160             single_embedding = use_conditioning * embeddings['single']
161             pair_embedding = use_conditioning * embeddings['pair']
162
163             rel_features = featurization.create_relative_encoding(
164                 batch.token_features, max_relative_idx=32,
                    max_relative_chain=2
165             ).to(dtype=pair_embedding.dtype)
166             features_2d = torch.concatenate([pair_embedding,
                rel_features], dim=-1)
167
168             pair_cond = self.pair_cond_initial_projection(
169                 self.pair_cond_initial_norm(features_2d)
170             )
171
172             pair_cond += self.pair_transition_0(pair_cond)
173             pair_cond += self.pair_transition_1(pair_cond)
174
175             target_feat = embeddings['target_feat']
176             features_1d = torch.concatenate(
177                 [single_embedding, target_feat], dim=-1)
178             single_cond = self.single_cond_initial_projection(
179                 self.single_cond_initial_norm(features_1d))
180
181+            self.single_cond = single_cond
182+            self.pair_cond = pair_cond
183+
184+        single_cond = self.single_cond.clone()  # will be changed
185+        pair_cond = self.pair_cond
186         noise_embedding = self.fourier_embeddings(
```

3. 基于 profiling 进行优化

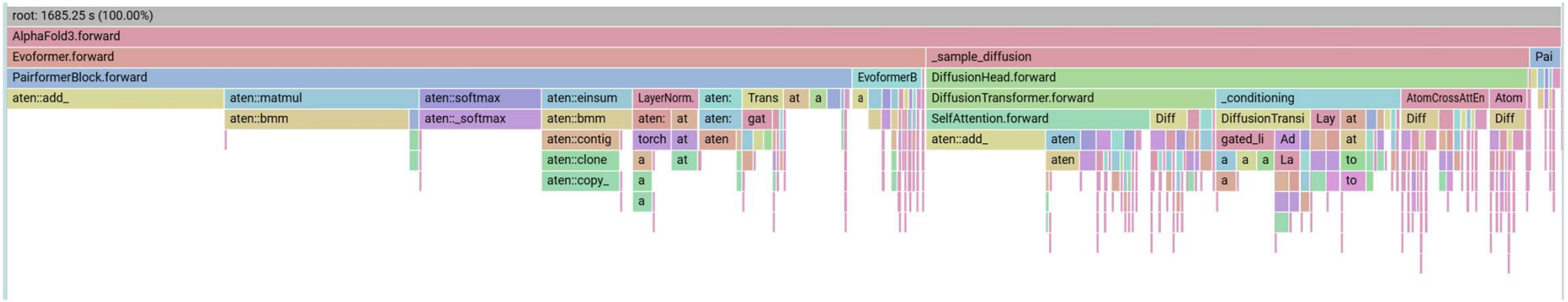
耗时聚焦在

GridSelfAttention、

TriangleAttention、

SelfAttention 等模块

使用 C++ 重写算子



4. 使用 LIBXSMM 重写算子

优点

- GEMM 性能高，支持最新的 AMX 指令集
- 内部基于 JIT 实现
  - 不依赖特定编译器/编译选项，GCC 编译出来也能和 ICPX 跑的一样快!
  - “一次编译，到处运行”，不强求硬件平台支持高版本指令集

4. 使用 LIBXSMM 重写算子

使用子库 **Intel(R) Tensor Processing Primitives extension for PyTorch*** 开发 af3_kernels

- 与 PyTorch 高度融合，集成了 pybind11，上手即用
- 原语层面封装，结构与风格统一
- 将裸指针包装为任意形状数组，提升代码可读性
- 低成本分模块计时，便于性能调优，并可与 `torch.profiler` 兼容
- 有现成的对 Alphafold2 中 `GridSelfAttention` 模块的重写样例与对应的 PyTorch 代码，便于模仿学习

4. 使用 LIBXSMM 重写算子

Gated Linear Unit 算子示例

```
    {
        RECORD_SCOPE(glu_gemm, {input, weight, out});
        auto gemm      = SCOPEIT((tpp::BrgemmTPP<T, T>(M_BLK, N_BLK, K, 0, 0, K, 2 * N, N_BLK, 0.0f, 0, 1, vnni_repacked, 0, 0)), BRGEMM);
        auto silu      = SCOPEIT(tpp::SiLUFwdTPP<T>(M_BLK, N_BLK), SILU);
        auto mul       = SCOPEIT(tpp::MulTPP<T>(M_BLK, N_BLK, N_BLK, N), EW_MUL);
        auto gemm_left = SCOPEIT((tpp::BrgemmTPP<T, T>(leftover, N_BLK, K, 0, 0, K, 2 * N, N_BLK, 0.0f, 0, 1, vnni_repacked, 0, 0)), MARGIN);
        auto silu_left = SCOPEIT(tpp::SiLUFwdTPP<T>(leftover, N_BLK), MARGIN);
        auto mul_left  = SCOPEIT(tpp::MulTPP<T>(leftover, N_BLK, N_BLK, N), MARGIN);
#pragma omp parallel for collapse(2)
        for(int i = 0; i ≤ Mb; i++) {
            for(int j = 0; j < Nb; j++) {
                LIBXSMM_ALIGNED(T tmp[M_BLK][N_BLK], 64);
                LIBXSMM_ALIGNED(T tmp2[M_BLK][N_BLK], 64);
                if(i ≠ Mb) {
                    gemm.config();
                    gemm(&input_a[i][0][0], &weight_a[0][j][0], &tmp[0][0], 1, true);
                    silu(&tmp[0][0], &tmp[0][0], &tmp2[0][0]);
                    gemm(&input_a[i][0][0], &weight_a[0][j + Nb][0], &tmp2[0][0], 1, true);
                    mul(&tmp[0][0], &tmp2[0][0], &out_a[i][0][j][0]);
                    gemm.release();
                } else if(leftover) {
                    gemm_left.config();
                    gemm_left(&input_a[Mb][0][0], &weight_a[0][j][0], &tmp[0][0], 1, true);
                    silu_left(&tmp[0][0], &tmp[0][0], &tmp2[0][0]);
                    gemm_left(&input_a[Mb][0][0], &weight_a[0][j + Nb][0], &tmp2[0][0], 1, true);
                    mul_left(&tmp[0][0], &tmp2[0][0], &out_a[Mb][0][j][0]);
                    gemm_left.release();
                }
            }
        }
    }
```

4. 使用 LIBXSMM 重写算子

算子性能调优

- 填充 Tensor 至分块大小，并对齐到 64 Byte
- 预先对权重进行 VNNI Repack
- 动态调整分块大小（通常以能放入 L2 Cache 为佳）
- 设置 Flush to zero 标志，避免非标浮点数大幅拖慢性能
- 使用线代等价变换，节省不必要的矩阵转置操作

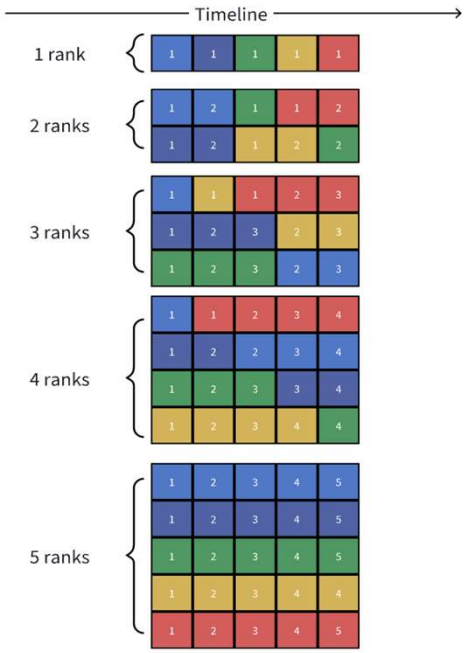
	均不转置	A转置	B转置	均转置
均禁用vnni	1.8T	1.4T	1.8T	1.4T
A vnni	1.8T	1.4T	1.8T	1.4T
B vnni	2.1T	NaN	NaN	NaN
均vnni	2.2T	0.15T	2.2T	0.16T

	GridSelfAttention	TriangleMultiplication	Gated Linear Unit
Sequence length	1536	1536	1536
Block size	128	128	128
Original speed	2699.24ms	529.58ms	126.71ms
Optimized Speed	186.10ms	99.67ms	28.09ms
Speedup	**14.50x**	**5.31x**	**4.51x**

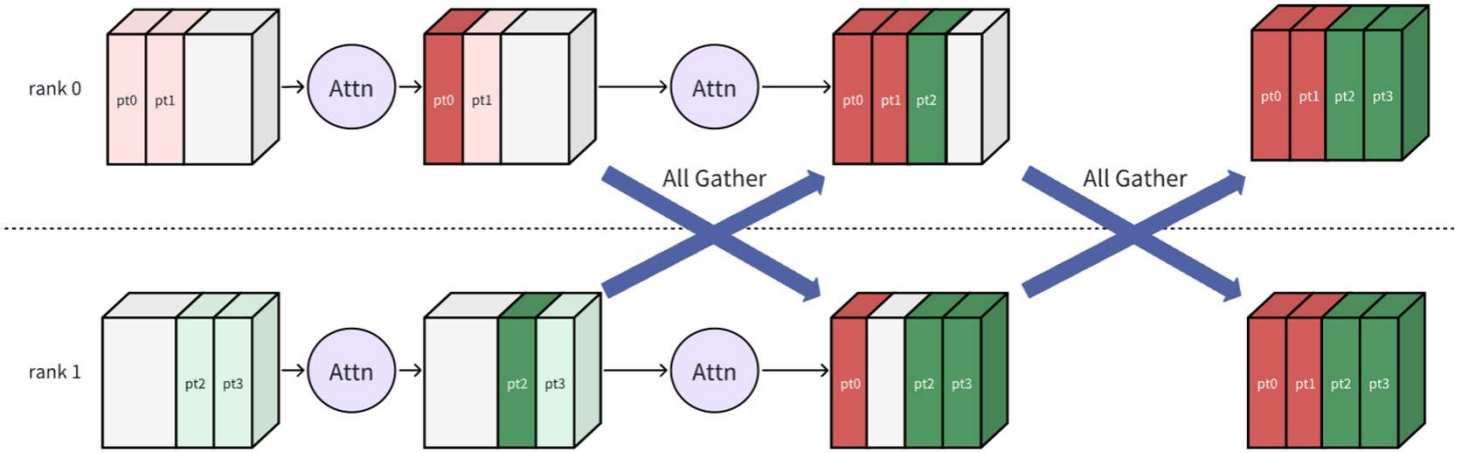
算子加速比

5. 多机推理优化

- 采用 **oneCCL** 作为通讯后端
- Diffusion 部分数据并行，设计任务调度系统在保持负载均衡的同时最小化通讯量与通讯次数
- GridSelfAttention 部分切分 seq_len，使计算与通讯可以互相掩盖

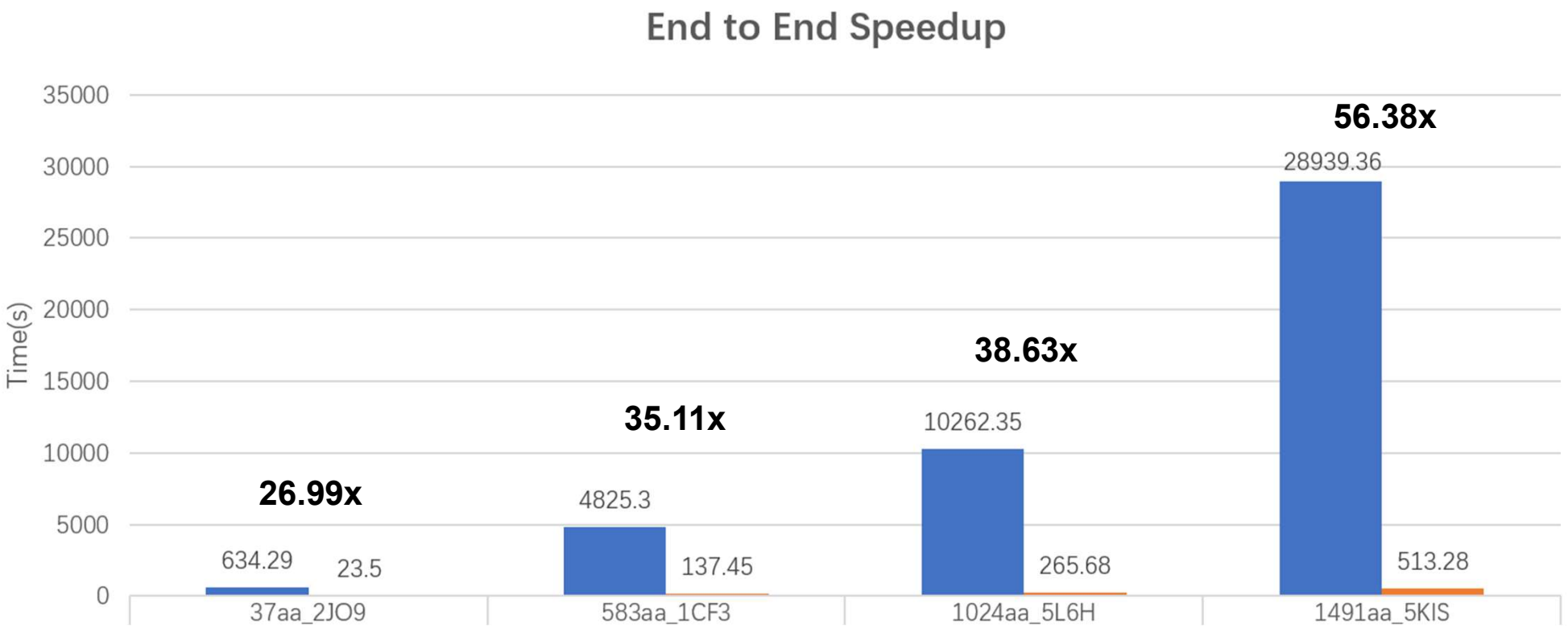


Distributed Diffusion



Distributed GridSelfAttention

5. 多机推理优化


End to End Speedup
Time(s)
35000
30000
25000
20000
15000
10000
5000
0
26.99x
634.29
23.5
35.11x
4825.3
137.45
38.63x
10262.35
265.68
56.38x
28939.36
513.28
37aa_2JO9
583aa_1CF3
1024aa_5L6H
1491aa_5KIS

6. 一些踩过的坑

- 尝试使用 Intel Extension for PyTorch，但算子性能没有显著提升
- 尝试使用 torch.compile，但一直无法完全缓存编译结果，会在运行时触发重编译
- 尝试了多种多机并行 TriangleMultiplication 的方案，但在目前数据规模下速度不如单机

Thanks!

项目开源地址：

https://github.com/HPC-SJTU/xfold

https://github.com/HPC-SJTU/af3_kernels