3.faster_transformer

> **Note: FasterTransformer development has transitioned to [TensorRT-LLM](https://github.com/NVIDIA/TensorRT-LLM/tree/release/0.5.0). All developers are encouraged to leverage TensorRT-LLM to get the latest improvements on LLM Inference. The NVIDIA/FasterTransformer repo will stay up, but will not have further development.**

1.简介

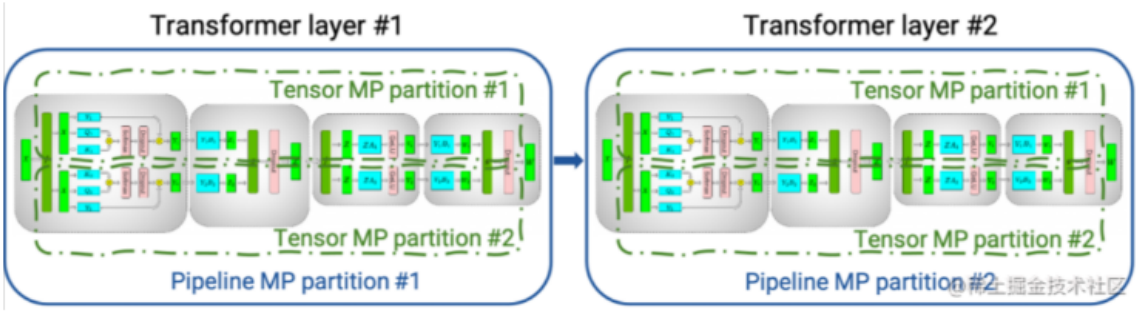
NVIDIA FasterTransformer (FT) 是一个用于实现基于Transformer的神经网络推理的加速引擎。它包含Transformer块的高度优化版本的实现，其中包含编码器和解码器部分。使用此模块，您可以运行编码器-解码器架构模型（如：T5）、仅编码器架构模型（如：BERT）和仅解码器架构模型（如： GPT）的推理。

FT框架是用C++/CUDA编写的，依赖于高度优化的 cuBLAS、cuBLASLt 和 cuSPARSELt 库，这使您可以在 GPU 上进行快速的 Transformer 推理。

与 NVIDIA TensorRT 等其他编译器相比，**FT 的最大特点是它支持以分布式方式进行 Transformer 大模型推理。**

下图显示了如何使用张量并行 (TP) 和流水线并行 (PP) 技术将基于Transformer架构的神经网络拆分到多个 GPU 和节点上。

- 当每个张量被分成多个块时，就会发生张量并行，并且张量的每个块都可以放置在单独的 GPU 上。在计算过程中，每个块在不同的 GPU 上单独并行处理；最后，可以通过组合来自多个 GPU 的结果来计算最终张量。
- 当模型被深度拆分，并将不同的完整层放置到不同的 GPU/节点上时，就会发生流水线并行。



2.FasterTransformer 中的优化技术

与深度学习训练的通用框架相比，FT 使您能够获得更快的推理流水线以及基于 Transformer 的神经网络具有更低的延迟和更高的吞吐量。 FT 对 GPT-3 和其他大型 Transformer 模型进行的一些优化技术包括：

2.1 层融合（Layer fusion）

这是预处理阶段的一组技术，将多层神经网络组合成一个单一的神经网络，将使用一个单一的核（kernel）进行计算。 这种技术减少了数据传输并增加了数学密度，从而加速了推理阶段的计算。 例如， multi-head attention 块中的所有操作都可以合并到一个核（kernel）中。

FasterTransformer 是用 **手写 CUDA kernel** 实现的，不是靠自动图优化（如 TensorRT 的 tactic selection）。

开发者：

1. 分析计算图中哪些 ops 可以合并
2. 手动编写一个 CUDA kernel，把多个 ops 的逻辑写在一起
3. 优化内存访问、使用 shared memory、warp-level primitives 等
4. 针对不同 batch size / sequence length / head 数做特化

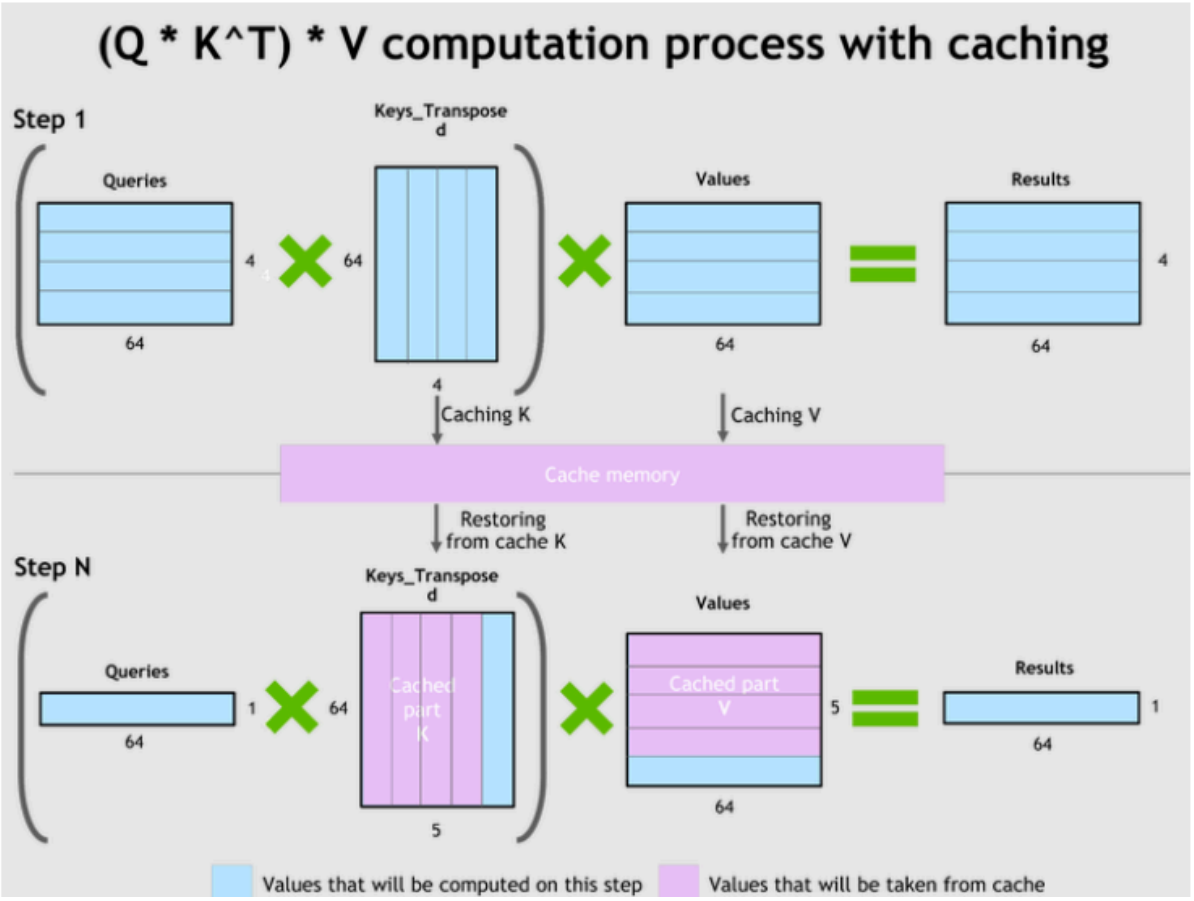
所以，**层融合 = 人工设计的高性能算子融合**，是推理加速的核心手段之一。

> **Note**
>
> 更准确的说法是：
>
> > **将原本由多个独立算子（ops）组成的计算子图（subgraph），融合成一个单一的、高效的执行单元（通常是 CUDA kernel）。**
>
> 这个“子图”可能来自：
>
> - 一个 Attention 层内部（如上面的 MHA）
> - 一个 FFN（前馈网络）层：Linear → GELU → Linear → 融合成一个 kernel
> - 甚至跨子层：比如 Attention 输出 + 残差连接 + LayerNorm，也可以融合（FasterTransformer 确实这么做了！）
>
> 例如，FasterTransformer 的典型融合单元是：
>
> ```
> [Attention + Add + LayerNorm] + [FFN + Add + LayerNorm]
> ```
>
> 整个 Transformer 层被拆成两个大 kernel，而不是 10 多个小 kernel。

2.2 自回归模型的推理优化(激活缓存)

为了防止通过Transformer重新计算每个新 token 生成器的先前的key和value，FT 分配了一个缓冲区来在每一步存储它们。

虽然需要一些额外的内存使用，但 FT 可以节省重新计算的成本。该过程如下图所示，相同的缓存机制用于 NN **（神经网络）**的多个部分。



注意：Query 是当前步新输入的，不需要缓存；只有 Key 和 Value 是“历史信息”，需要缓存供后续步骤使用。

2.3 内存优化

与 BERT 等传统模型不同，大型 Transformer 模型具有多达数万亿个参数，占用数百 GB 存储空间。即使我们以半精度存储模型，GPT-3 175b 也需要 350 GB。因此有必要减少其他部分的内存使用。

例如，在 FasterTransformer 中，**在不同的解码器层重用了激活/输出的内存缓冲（buffer）**。由于 GPT-3 中的层数为 96，因此我们只需要 1/96 的内存量用于激活。

FasterTransformer 做了这样一件事：**“重用激活/输出的内存缓冲区”**

> **只分配两块内存缓冲区（buffer）：**
>
> - `buffer_in`：当前层的输入
> - `buffer_out`：当前层的输出

计算流程如下：

```
初始：input → 存入 buffer_in

for layer in 1 to 96:
    用 buffer_in 作为输入
    计算 layer 的输出 → 写入 buffer_out
    交换 buffer_in 和 buffer_out（指针交换，几乎无开销）
```

类型	是否需要缓存	原因
层间激活（如 a1, a2...）	不需要	只被下一层用一次，之后永远不用

类型	是否需要缓存	原因
Key / Value（在 Attention 中）	需要	**未来所有新 token 都要用到它们**（自回归特性）

- **普通激活值**：用完即弃，可重用内存。
- **KV Cache**：必须保留到生成结束，不能重用。

2.4 使用 MPI 和 NCCL 实现节点间/节点内通信并支持模型并行

FasterTransormer 同时提供张量并行和流水线并行。 对于张量并行，FasterTransformer 遵循了 Megatron 的思想。 对于自注意力块和前馈网络块，FT 按行拆分第一个矩阵的权重，并按列拆分第二个矩阵的权重。 通过优化，FT 可以将每个 Transformer 块的归约（reduction）**操作减少到两次。**

对于流水线并行，FasterTransformer 将整批请求拆分为多个微批，隐藏了通信的空泡（bubble）。 FasterTransformer 会针对不同情况自动调整微批量大小。

> **Tip**
>
> **FasterTransformer 如何隐藏 bubble?**
>
> **核心方法：使用 micro-batching + 交错调度**
>
> **具体机制**：
>
> 1. **将一个大 batch 拆成多个 micro-batches**（比如 8 个）
> 2. 流水线启动后，连续推送 micro-batch：
>   - 时间步 1：micro-batch 1 进入 stage 1
>   - 时间步 2：micro-batch 2 进入 stage 1，同时 mb1 进入 stage 2
>   - 时间步 3：mb3 → stage1, mb2 → stage2, mb1 → stage3
>   - ...
> 3. 当流水线"充满"后，**所有 stage 同时工作**，bubble 被压缩到开头和结尾
>
> > **不是"增加"微批次，而是"合理划分"微批次**，让通信和计算重叠。
>
> **FT 的智能之处：**
>
> - 自动根据 **GPU 数量、模型大小、batch size** 调整 micro-batch 数量
> - 如果 micro-batch 太小 → 通信开销大
> - 如果 micro-batch 太大 → 无法填满流水线，bubble 大
> - FT 会选一个**平衡点**，最大化 GPU 利用率
>
> **效果示意：**
>
> ```
> 时间 →
> GPU0: [mb1][mb2][mb3][mb4]...
> GPU1:     [mb1][mb2][mb3][mb4]...
> GPU2:         [mb1][mb2][mb3][mb4]...
> ```
>
> 中间大部分时间没有空闲 → **bubble 被隐藏了**！

Megatron 的思想：

核心目标：在多个 GPU 之间拆分一个 Transformer 层的计算，使得每个 GPU 只存一部分权重、只算一部分结果，从而突破单卡显存限制。

> Megatron-LM（由 NVIDIA 提出）主要采用 **张量并行（Tensor Parallelism, TP）**，不是把"层"分给不同 GPU（那是流水线并行），而是把单个层内部的矩阵运算拆开。

举例：

(1) 前馈网络（FFN）

标准 FFN：

```
Y = GeLU(X @ W1) @ W2   # W1: d_model → d_ff,  W2: d_ff → d_model
```

Megatron 拆分方式：

- 将 `W1` 按行切分（split along output dim）→ 每个 GPU 算一部分中间激活
- 将 `W2` 按列切分（split along input dim）→ 每个 GPU 持有一部分 `W2`

计算流程：

1. 所有 GPU 同时计算：`Z_i = GeLU(X @ W1_i)`
2. `All-Gather Z_i` → 拼成完整 `Z`（或等价地，下一步用 `All-Reduce`）
3. 每个 GPU 计算局部输出：`Y_i = Z @ W2_i`
4. `All-Reduce Y_i` → 得到完整 `Y`

> 但 Megatron 更聪明：它把步骤 2 和 4 合并，**只用一次 All-Reduce。**

Naive 方法 vs Megatron 方法：

步骤	Naive 方法	Megatron 方法
1. 计算 $Z^{(i)}$	各 GPU 计算 $Z^{(i)}$	各 GPU 计算 $Z^{(i)}$
2. 拼接 Z	`All-Gather` $Z^{(i)}$ → 得到完整 Z（通信量大：$b \times 4d$）	**跳过此步！** 不显式拼接 Z
3. 计算 $Y^{(i)}$	各 GPU 用**完整** Z 与**局部** $W_2^{(i)}$ 计算 $Y^{(i)}$（逻辑不合理：W_2 按列切分，应只作用于对应部分）	各 GPU 用**局部** $Z^{(i)}$ 和**局部** $W_2^{(i)}$ 计算 $Y^{(i)}$
4. 得到最终 Y	无需通信（但已在步骤 2 通信）	`All-Reduce` $Y^{(i)}$ → 得到完整 Y

> **Tip**
>
> **Megatron 用一次 `All-Reduce` 替代了 Naive 方法中的 `All-Gather` + 额外计算**，并且：
>
> - **通信量更小**：
> - Naive 通信 Z：尺寸为 $b \times 4d$
> - Megatron 通信 Y：尺寸为 $b \times d$
> - **计算更合理**：权重切分与激活切分对齐，避免跨 GPU 使用不匹配的数据

- **仅需一次通信**：在输出阶段通过 `All-Reduce` 同时完成通信与求和

结果：更低通信开销 + 更高硬件效率。

(2) 自注意力（Self-Attention）

- **Q/K/V 投影**：`Q = X*W_Q`，类似 FFN，`W_Q` / `W_K` / `W_V` 按行切分。
- **Attention 输出**：`O = Attention(Q, K, V) @ W_O`
  - `W_O` 按列切分（因为它是将多头结果映射回 `d_model`）
  - 每个 GPU 算局部 `O_i`
  - 最后 `All-Reduce` 得到完整 `O`

拆分方式	按行切分（split output dim）	按列切分（split input dim）
目的	让每个 GPU 独立计算一部分中间结果	让最终输出可通过对局部结果归约得到

> 这样，计算可以并行，通信只需在关键点做一次归约。

2.5 MatMul 核自动调整（GEMM 自动调整）

矩阵乘法是基于 Transformer 的神经网络中最主要和繁重的操作。 FT 使用来自 CuBLAS 和 CuTLASS 库的功能来执行这些类型的操作。 重要的是要知道 MatMul 操作可以在“硬件”级别使用不同的底层（low-level）算法以数十种不同的方式执行。

GemmBatchedEx 函数实现了 MatMul 操作，并以cublasGemmAlgo_t作为输入参数。 使用此参数，您可以选择不同的底层算法进行操作。

FasterTransformer 库使用此参数对所有底层算法进行实时基准测试，并为模型的参数和您的输入数据（注意层的大小、注意头的数量、隐藏层的大小）选择最佳的一个。 此外，FT 对网络的某些部分使用硬件加速的底层函数，例如： **expf、** shfl_xor_sync。

2.6 低精度推理

FT 的核（kernels）**支持使用 fp16 和 int8 等低精度输入数据进行推理。** 由于较少的数据传输量和所需的内存，这两种机制都会加速。 同时，int8 和 fp16 计算可以在特殊硬件上执行，例如：Tensor Core（适用于从 Volta 开始的所有 GPU 架构）。

除此之外还有**快速的 C++ BeamSearch 实现**、当模型的权重部分分配到八个 GPU 之间时，**针对 TensorParallelism 8 模式优化的 all-reduce**。

1. **快速的 C++ Beam Search 实现**

背景:在自回归语言模型（如 LLM）推理中，**Beam Search** 是一种常用的解码策略，用于在每一步保留多个候选序列（称为“beam”），以在生成质量和多样性之间取得平衡。

- 原生 Python/TensorFlow/PyTorch 实现通常在 **host（CPU）端调度**，每一步需：
  - 从 GPU 拿回 top-k 候选
  - 在 CPU 上重组序列
  - 再把新输入送回 GPU
- 这会导致 **频繁的 CPU-GPU 数据搬运 + 同步开销**，严重拖慢推理速度。

FT 的优化：FasterTransformer 将 **整个 Beam Search 逻辑用 C++ + CUDA 重写**，并集成到推理 kernel 中：

- **全程在 GPU 上运行**：无需每步回传到 CPU
- **与模型前向计算融合**：避免中间结果落盘或跨设备传输
- **内存预分配 + 循环复用**：减少动态分配开销
- **高度定制化数据结构**：如紧凑的 beam 状态表、score 缓存等

2. **针对 Tensor Parallelism 8 模式优化的 All-Reduce**

背景：在 Tensor Parallelism（TP）中，如你之前了解的 Megatron 方式，模型权重被切分到多个 GPU（比如 8 卡），每层计算后需通过 **All-Reduce** 合并局部结果。

- 标准 NCCL All-Reduce 是通用的，但未必对 **特定通信模式 + 特定卡数** 最优
- 当 TP=8（即 8 个 GPU 做张量并行）时，通信模式高度结构化（例如：每次通信的数据大小、形状、拓扑位置固定）

FT 的优化：FasterTransformer 针对 **TP=8 这一常见配置**，做了深度优化：

- **定制通信 kernel**：绕过通用 NCCL，使用更高效的点对点或环状通信原语
- **与计算 kernel 融合**：例如在 GEMM 计算完成后直接启动通信，减少同步点
- **内存布局对齐**：确保通信 buffer 与计算 output buffer 一致，避免 transpose 或拷贝
- **利用 NVLink 拓扑**：在支持 NVLink 的多 GPU 服务器（如 A100 8 卡）上，显式优化通信路径

为什么特别提 “TP=8”？

- 8 是常见训练/推理部署规模（如单台 DGX A100 有 8×A100）
- FT 预编译或特化了该配置的 kernel，实现 **“开箱即用”的高性能**

> 这体现了 FT 的设计哲学：**不追求通用性，而是在典型场景下做到极致优化。**

技术	优化目标	与你关注点的关联
C++ Beam Search	减少 CPU-GPU 交互、降低解码延迟	提升端到端推理效率，尤其对长文本生成关键
TP=8 专用 All-Reduce	降低张量并行通信开销	直接提升多 GPU 利用率，减少 bubble 和通信瓶颈

这两项优化共同支撑了 FasterTransformer 在 **大规模 LLM 推理场景下的高性能表现**，也体现了其“**软硬协同 + 场景特化**”的核心思想。

3.支持的模型

目前，FT 支持了 Megatron-LM GPT-3、GPT-J、BERT、ViT、Swin Transformer、Longformer、T5 和 XLNet 等模型。您可以在 GitHub 上的FasterTransformer库中查看最新的支持矩阵。

4.存在的问题

英伟达新推出了TensorRT-LLM，相对来说更加易用，后续FasterTransformer将不再为维护了。