

DiT 图像生成挑战 - 优化报告

队名: NullPointerException

队员: 刘以煦、邵浩林、刘君昊

日期: 2025 年 7 月 29 日

1. 单卡 Baseline 结果 (10%)

从加载模型至 sample 生成, 记录运行时间为: 381.0304 s

```
DiT-SUSTCSC > dit_inference.out
1  >> Running sample_baseline.py with job_id = 5221
2  Using device: cuda
3  CUDA device name: Tesla V100-SXM2-32GB
4  Sample image saved to: /work/sustcsc_11/DiT-SUSTCSC/output/job_5221/sample_5221.png
5  🕒 sampling time: 381.0304 seconds
```

baseline 跑通后, 能生成对应标签类别的图片, 如下图:



2. 单卡 Profile 结果及性能瓶颈分析 (20%)

使用 perfetto 分析耗时前十的热点函数, 结果如下图:

Query result (10 rows) - 1,096.4ms SELECT slice.name, COUNT(1) AS call_count, SUM(slice.dur) AS total_dur, AVG(slice.dur) AS avg_dur FROM slice GROUP BY slice.name ORDER BY total_dur DESC				Copy
name	call_count	total_dur	avg_dur	
PyTorch Profiler (0)	1	363326932542	363326932542	< < 1-10 of 10 > >
sample.py(120): <module>	1	363326821720	363326821720	
sample.py(92): main	1	363326819926	363326819926	
diffusion/gaussian_diffusion.py(419): p_sample_loop	1	362068805480	362068805480	
diffusion/gaussian_diffusion.py(376): p_sample	250	361988198031	1447952792.124	
diffusion/respace.py(89): p_mean_variance	250	361920540856	1447682163.424	
diffusion/gaussian_diffusion.py(254): p_mean_variance	250	361917072093	1447668288.372	
diffusion/gaussian_diffusion.py(510): p_sample_loop_progressive	250	359684787896	1438739151.584	
aten:copy_	20264	351530428148	17347533.959139362	
diffusion/gaussian_diffusion.py(861): _extract_into_tensor	2000	351449387586	175724693.793	

Query history (1 queries)

```
SELECT
    slice.name,
    COUNT(1)      AS call_count,
    SUM(slice.dur) AS total_dur,
    AVG(slice.dur) AS avg_dur
FROM slice
GROUP BY slice.name
ORDER BY total_dur DESC
LIMIT 10;
```

- **p_sample_loop + p_sample** : 采样过程是模型运行的绝对瓶颈
 - 减少 num_sampling_steps (会影响生成图像质量)
 - 使用 DDIM / DPM-Solver 等更快采样器
 - 加速 transformer 的 attention 部分
- **aten::copy_** : 张量数据拷贝, 通常在 CPU ↔ GPU 数据转移 VAE decode 或模型 output 阶段调用
 - 用 AMP (自动混合精度) 减少中间张量大小 → 减少拷贝时间

3. 单卡优化方案与结果 (20%)

● 优化方案

考虑优化成本和优化效益, 依次进行以下优化:

- 使用 AMP 混合精度包裹模型推理 (简单高效) ·

TF32 虽然在计算上提供了加速, 但它仍然是 FP32 的变种, 而 AMP 采用了 FP16。相比于 TF32, FP16 占用内存小。FP16 在支持 Tensor Cores 的 GPU (如 A100、V100) 上能够更好地利用硬件资源, 从而加速计算过程。特别是在矩阵乘法 和 卷积 等计算密集型任务中, 使用 FP16 而不是 TF32 可以带来更显著的加速。且 AMP 会对部分算子进行算子融合, 减少中间数据的存储和传输, 进一步提升计算效率。

- 使用 TorchScript 编译模型, 还需将 forward_with_cfg 先封装成 nn.Module
 - 模块结构清晰: nn.Module 继承 + forward() 实现
 - 模型中无动态 Python 控制流
 - 所有核心模块 (PatchEmbed, TimestepEmbedder, DiTBlock 等) 都用 PyTorch 构建

● 优化结果

从加载模型至 sample 生成, 记录运行时间为: 116.5998 s

```
DiT-SUSTCSC > dit_inference.out
1  >> Running sample_01.py with job_id = 5621
2  Using device: cuda
3  CUDA device name: Tesla V100-SXM2-32GB
4  ✓ 成功加载已编译的 TorchScript 模型: /work/sustcsc_11/DiT-SUSTCSC/pretrained_models/DiT-XL-2_512x512_scripted.pt
5  Image saved: /work/sustcsc_11/DiT-SUSTCSC/output/job_5621/job_5621.png
6  ⚡ sampling time: 116.5998 seconds
7
```

生成对应标签类别的图片，如下图：



使用 perfetto 分析耗时前十的热点函数，结果如下图：

Query result (10 rows) - 673.4ms SELECT slice.name, COUNT(1) AS call_count, SUM(slice.dur) AS total_dur, AVG(slice.dur) AS avg_dur FROM slice GROUP BY slice.name ORDER BY total_dur DESC ...				Copy	⋮
name	call_count	total_dur	avg_dur	1 < 1-10 of 10 > >	
PyTorch Profiler (0)	1	110879016720	110879016720		
sample_01.py(165): <module>	1	110878916192	110878916192		
sample_01.py(116): main	1	110878914241	110878914241		
diffusion/gaussian_diffusion.py(419): p_sample_loop	1	109828114205	109828114205		
diffusion/gaussian_diffusion.py(376): p_sample	250	109748674772	438994699.088		
diffusion/repace.py(89): p_mean_variance	250	109683353821	438733415.284		
diffusion/gaussian_diffusion.py(254): p_mean_variance	250	109680193025	438720772.1		
diffusion/gaussian_diffusion.py(510): p_sample_loop_progressive	250	107834666504	431338666.016		
aten::to_copy	85662	86689817931	1011998.5282972613		
aten::copy_	100173	85413231065	852657.2136703503		

与 baseline 相比，耗时前十的热点函数均有明显加速。

4. 多卡方案设计 (25%)

● 各并行方式分析：

• 数据并行

- 思想：每个 GPU 拷贝一份完整模型，处理不同子 batch，梯度平均或采样结果合并

- 实现简单：PyTorch DataParallel 或 DDP

- 显存开销：每张卡需容纳完整模型 + 局部数据 ($\text{batch_size} / \#\text{GPU}$)

- 通信代价低（仅参数或最终输出同步）

- DiT 适配性：极佳，生成任务中样本间独立

- 问题：模型越大、卡越少，显存负担越重 (V100 的 32GB 足够)

• 流水线并行

- 思想：将模型拆成若干阶段，按顺序运行在不同 GPU 上，数据按批传输

- 适合模型：具有明显“阶段结构”的模型（如 GPT、BERT）

- **问题**: DiT 是 Transformer 均匀堆叠，层之间无显著切割点；批量推理时流水难以充分饱和（小 batch 无法 pipeline 化）；生成阶段不可流式：每张图从 noise 采样到图像，过程非分阶段，带来额外同步负担

- **张量并行**

- **思想**: 单层内部的计算被拆分，如 attention 的 QKV 投影拆到不同卡执行

- **适合模型**: 超大规模模型（10B 以上），单卡放不下时

- **问题**: DiT-XL (2.1B) 本身可以完整放入单个 V100 (32G)；张量并行需要深度集成库（如 Megatron-LM、DeepSpeed、Colossal-AI）；且推理时通信多、效率反而下降；模型过大时需要跨卡切分每个线性层。过度设计，反而复杂化部署，收益小，代价大

- **序列并行**

- **思想**: 将 token 维度切分（如对序列长度维度切片），每张卡处理部分 token

- **适合模型**: NLP 中处理长文本（序列长达几千 token）

- **问题**: DiT 输入是图像 patch (16x16 patch -> 1024 tokens for 512x512)，序列长度较短，切分后粒度太细，而且需要频繁通信（attention 中每层都需交换序列）。序列并行适合超长序列 NLP，不适用于 DiT 图像任务

- **并行方式选择：**

选择**数据并行**。原因如下：

- DiT 图像生成任务 样本间天然独立，非常适合 batch 切分

- 模型大小适中，可完整加载至单张 V100

- 通信开销小，结构简单，无需重构代码

- 4 卡处理 batch=8，每卡 batch=2 恰好均衡

- 推理任务无梯度回传，无需反向通信同步

- **多卡方案设计：**

- 在单卡优化的基础上，使用 PyTorch 的 Distributed Data Parallel (DDP) 推理，使用 NVIDIA 的 NCCL 库）来聚合梯度

- 每张卡根据 rank 分配标签（每张卡 2 个标签）
- 作业提交脚本中，使用 torchrun 启动四个进程（每张卡一个进程）
- 使用 torch.compile 编译模型，实现算子融合（考虑到矩阵乘法耗时）、图优化和并行化

5. 多卡优化结果 (25%)

生成对应标签类别的图片，如下图：



使用 perfetto 分析耗时前十的热点函数，结果如下图：

Query result (20 rows) - 731.8ms SELECT slice.name, COUNT(1) AS call_count, SUM(slice.dur) AS total_dur, AVG(slice.dur) AS avg_dur FROM slice GROUP BY slice.name ORDER BY total_dur DESC			
name	call_count	total_dur	avg_dur
torch/nn/modules/module.py(1755): _call_impl	638	52518994528	82318173.2413793
PyTorch Profiler (0)	1	43260564664	43260564664
threading.py(973): _bootstrap	1	43260487927	43260487927
threading.py(1016): _bootstrap_inner	1	43260487100	43260487100
tqdm/_monitor.py(60): run	1	43260483978	43260483978
sample_02_profiler.py(207): <module>	1	43260478989	43260478989
sample_02_profiler.py(143): main	1	43260476469	43260476469
diffusion/gaussian_diffusion.py(419): p_sample_loop	1	39643637265	39643637265
diffusion/gaussian_diffusion.py(376): p_sample	250	39562805923	158251223.692
diffusion/respace.py(89): p_mean_variance	250	39495004645	157980018.58
diffusion/gaussian_diffusion.py(254): p_mean_variance	250	39491557813	157966231.252
diffusion/gaussian_diffusion.py(510): p_sample_loop_progressive	250	37809122653	151236490.612
threading.py(589): wait	4	33286302266	8321575566.5
threading.py(288): wait	4	33286247862	8321561965.5
<built-in method acquire of _thread.lock object at 0x149243763140>	16	33286128969	2080383060.5625
diffusion/respace.py(124): __call__	250	25745921422	102983685.688
nn.Module: OptimizedModule_0	250	25698774767	102795099.068
torch/_dynamo/eval_frame.py(619): _fn	250	25695890514	102783562.056
nn.Module: RecursiveScriptModule_0	250	25681871889	102727487.556
forward	250	18209139131	72836556.524

不使用 profiler 时，记录运行时间为：44.4200 s，已达到赛题要求。

对应热点函数耗时比较：

	baseline	单卡优化	多卡	
main	363326.82	110878.914	43260.4765	
p_sample_loop	362068.805	109828.114	39643.6372	
p_sample	1447.68216	438.994699	158.251223	
p_mean_variance	1447.66829	438.733415	157.980018	
p_sample_loop_progressive	1438.73915	431.338666	151.23649	
aten::copy_	17.347533	0.852657	0.145068	单位: ms

6. 总结：

本次比赛的优化方向很明确，在对各种并行方式及 DiT 模型特点有了基本了解后，可以确定采用数据并行优化，速度大幅加快。

比赛遇到的第一个问题是如何使用 profiler。一开始我使用 Torch 自带的 profiler，用 chrome://tracing/ 打开。但似乎因为文件过大打不开。故后续使用 perfetto。

除了并行加速外，经由 profile 分析，我主要针对矩阵乘法和 Transformer 的 attention 方面加速。放一个 G 老师的图：

函数名称	对应的 Transformer 部分
p_sample_loop	自注意力 (Self-Attention) 层
p_sample	多头注意力 (Multi-Head Attention)
aten::copy_	参数传递和梯度更新
p_mean_variance	输出规范化 (Layer Normalization)
p_sample_loop_progressive	层与层之间的逐步计算

开始我想利用 Flash Attention 加速，但是 V100 不支持（捣鼓了好久），后续有尝试 xformers，效果不显著。

综合优化效益，我除了采用数据并行外，使用了 AMP 混合精度包裹模型推理部分，有效利用 Tensor Cores 进行硬件加速，同时减少了内存消耗。使用 TorchScript 和 torch.compile 编译模型。

本次比赛我初步掌握了如何使用集群，如何配置环境，借助赛题背景初步了解了大模型，也为我后续的 CS 学习打下了基础。