

大模型计算 - HW1

September 2025

1 Introduction

本次作业需要在云计算容器中搭建环境, 学习如何进行基本的训练与推理的 profiling, 并计算 Model FLOPs Utilization(MFU), 理解硬件利用率.

此外建议学习 Nsight system 等性能分析工具, 本次作业不作要求.

2 Preliminaries

CUDA CUDA 是 Nvidia 推出的并行计算平台和编程模型, 用于处理 GPU 上的通用计算. 本次作业, 我们可以将 CUDA 分为运行时, 编译器以及工具组.

- 运行时负责设备管理内存分配以及启动 Kernel, 目前 pip install torch 会默认安装 cuda runtime 以及 cublas 等计算库.
- 编译器负责将 CUDA C/ptx 代码编译为 kernel, 需要安装完整 CUDA 或 pip install nvidia-cuda-nvcc
- Tools, 如 nsys, ncu 等 profiling 工具. Nsys 用于系统级的 profiling, 定位性能瓶颈. ncu 用于单个 kernel 的 profiling 和调优. cuda-gdb/compute-sanitizer 可以用于调试和内存检查.

需要注意, 不同版本 cuda 有着最低系统驱动版本要求, 通过 `nvidia-smi` 可以检查驱动版本以及最高支持的 CUDA 版本, 注意显示的 CUDA 版本与实际使用的 CUDA 没有关系.

Kernel CUDA Kernel 是在 GPU 上运行的计算程序. 我们在 Host 侧 (CPU) 上启动 Kernel, 可以设置 Kernel 使用的资源数量, Hopper 之前可以简单概括为:

- grid size, 启动多少 threadblock (or Cta), 一般可以认为会顺序分配到 GPU 的 SM 上. 如果资源足够, 一个 SM 可以运行多个 threadblock. 如果某个 SM 上的 threadblock 计算完成, 调度器将新的 threadblock 拉起运行.
- block size, 每个 threadblock 内有多少逻辑线程. 一个 SM 中实际有 4 个 warp scheduler, 每个 warp 32 个线程. 一个物理线程可以运行多个逻辑线程.
- shared memory, 每个 threadblock 有着所有线程都能访问的存储, 时延相较于 global memory (显存) 小很多. SM 有着固定的 shared memory 总量.

Kernel 可以利用 CUDA Core 实现一般运算, Tensor Core 专门设计用于高效矩阵乘法运算. 考虑到目前大部分深度学习计算为矩阵乘法, 我们希望 Tensor Core 利用率尽可能高.

Pytorch 的底层计算库会根据参与计算 Tensor 的 Device 使用不同的计算方法. 对于 cuda tensor, 其操作通过 cuda kernel 实现.

```
import torch
a = torch.randn((16, 16))
a = a.cuda()
b = a * a # pointwise kernel
```

FLOP 一次浮点数操作. 例如 $a = b + c$, $a = b * c$. 一次乘加操作 ($d = a \times b + c$) 为 2FLOP. 对于 $M \times N \times K$ 的矩阵乘法 $C = AB$ (即 $A \in \mathbb{R}^{M \times K}, B \in \mathbb{R}^{K \times N}, C \in \mathbb{R}^{M \times N}$), 共有 $2M \times N \times K$ FLOPs. 另外 Float point operations per second 简称为 FLOPS. 可以衡量吞吐, 例如 A100 的 BF16 理论 TensorCore 吞吐为 312TFLOPS ($T = 10^{12}$).

MFU MFU 最早在 [PaLM](#) 中通过 Token 吞吐定义. 在 [Megatron Paper 3](#) 中, Model FLOPs 定义为模型理论上一次前向和一次反向计算的 FLOP 数

量. Model FLOPs Utilization (MFU) 指进行有效计算的等效吞吐, 一般在训练场景使用, 且仅计算能够利用 Tensor Core 的矩阵乘利用率.

$$\text{MFU} = \frac{\text{Model FLOPs}}{\text{Hardware Peak FLOPs} \times \text{Iteration Time}}$$

例如模型在 n 张 GPU 上训练一个 Batch, 用时 t 秒, 理论需要 x FLOPs. 每个 GPU 每秒理论峰值为 y FLOPS. 则每个 Device 的等价吞吐为 $\frac{x}{nt}$ FLOPS. MFU 为 $\frac{x}{nyt}$.

对于线性层 $Y = XW^T$. 若一个 batch 有 N tokens, input size 为 H_i , output size 为 H_o . 其前向 FLOPs 为 $2NH_iH_o$. 反向 $\nabla X = \nabla YW, \nabla W = \nabla Y^T X$, FLOPS 为 $2NH_oH_i + 2H_oNH_i = 4NH_iH_o$, 是前向两倍. 由此对于线性层, 我们可以只计算前向 FLOPs, 乘 2 得到反向 FLOPs.

一般来说 Model FLOPs 指的是一个 batch (即一个完整优化步) 的总体 FLOPs, 在实际训练中, 我们会将一个 batch 切分为多个 microbatch, 本次作业中计算 FLOPs 均为 batch, 忽略梯度累积.

对于推理系统, 我们一般更看重 TTFT 和 TPS 指标.

TTFT Time to First Token(TTFT) 用于衡量一个请求发送到第一个 Token 返回的用时.

TPS Tokens Per Second(TPS) 用于衡量一段时间内, 推理引擎输出 token 的速度.

Parallelism 实际训练推理需要在多机器多卡集群上进行. 涉及到不同的并行策略以及调度, 通信问题. 本次作业不涉及 2 卡以上场景.

3 Setup

本次作业需要 SM80 及以上计算能力 GPU. 建议使用 RTX4090.

3.1 容器云

学校有 500 元容器云[算力券](#).

创建有 1 张 4090 的容器，选择 Pytorch 2.7.0 镜像 (下面的流程不用镜像预装的 torch)。创建完成后查看详情中可以看到 ssh 地址、端口和密码。



使用 vscode 连接以及配置参考<https://code.visualstudio.com/docs/remote/ssh>. 通过 `nvidia-smi` 获取基本 GPU 信息

```
root@p-e7dbcd15cdf-ackcs-00gjfkad:/usr/local# nvidia-smi
Fri Sep 26 10:12:39 2025
```

NVIDIA-SMI 550.54.14 Driver Version: 550.54.14 CUDA Version: 12.4										
GPU	Name	Phys. Mem.	Usage	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC	
0	NVIDIA GeForce RTX 4090	24576 MiB	0 MiB	On	00000000:01:00:00	Off	0	0	Off	
		9%	27C	P8	15W / 450W	111B / 24580MiB	9%	Default	N/A	

Processes:						
GPU	GI	CI	PID	Type	Process name	GPU Memory Usage
ID	ID					
No running processes found						

可以注意到驱动版本号为 550.54.14, 最高支持 CUDA 版本为 12.4.

该镜像默认安装 conda, 使用 `conda init` 初始化 `.bashrc`. 重新启动 terminal 能够看到 conda 环境前缀.

conda 新建 conda 环境 `conda create -n torch python=3.12 -y`. 并通过 `conda activate torch` 进入. 用 `which pip` 检查环境路径是否正确

```
(base) root@p-e7dbcd15cdf-ackcs-00gjfkad:~# conda activate torch
(torch) root@p-e7dbcd15cdf-ackcs-00gjfkad:~# which pip
/root/.conda/envs/torch/bin/pip
```

torch 我们可以安装 cuda12.8 版本的 torch, 这是因为 CUDA 从 11.x 开始保证小版本内部的兼容性, 不过部分高版本 feature 不可用.

`pip install torch -i https://pypi.tuna.tsinghua.edu.cn/simple`.

`pip install numpy -i https://pypi.tuna.tsinghua.edu.cn/simple`

安装完成后, 可以检查 cuda 是否支持

```
import torch
print(torch.cuda.is_available())
# True
print(torch._C._GLIBCXX_USE_CXX11_ABI)
# True 是否使用 CXX11 ABI
print(torch.__config__.show())
# 编译信息
```

Flash attention 基础的 Flash Attention2 已经被集成于 [pytorch](#). 不过 page attention 等算法需要更多接口. 我们可以进入 github 的 [release 页面](#)选择合适的 whl 安装.

例如目前环境为 python 3.12, torch 2.8.0, cuda 12.x. CXX11ABI True. 选择

flash_attn-2.8.3+cu12torch2.8cxx11abiTRUE-cp312-cp312-linux_x86_64.whl 下载. 可以本地下载传到容器, 或开启[代理](#)直接下载.

```
export https_proxy=\
    "http://u-UE25Z3:tXGJgV92@10.255.128.102:3128"
export http_proxy=\
    "http://u-UE25Z3:tXGJgV92@10.255.128.102:3128"
pip install https://github.com/.../flash_attn_xxx.whl
```

安装后可以测试一下

```
import torch
from flash_attn import flash_attn_func
B = 1
S = 4096
H = 32
D = 128
q, k, v = [
    torch.randn(
        (B, S, H, D),
```

```

        device='cuda',
        dtype=torch.bfloat16
    ) for _ in range(3)
]

o = flash_attn_func(q, k, v)

# 注意 pytorch 的 API 不同
qt, kt, vt = map(lambda x: x.transpose(1, 2), [q, k, v])
torch_o = torch.nn.functional.scaled_dot_product_attention(
    qt, kt, vt
).transpose(1, 2)

print(torch.allclose(o, torch_o, atol=1e-4, rtol=0.01))

```

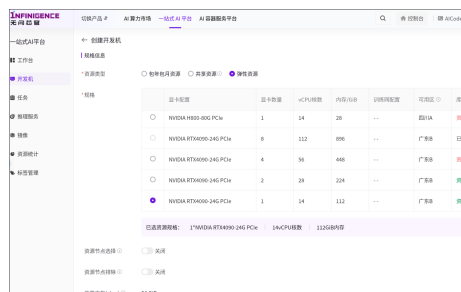
Nsight System(Extra) 容器云的镜像默认安装 cuda 12.8 于 `/usr/local/cuda/`. 可以直接使用 `nsys`.

3.2 无问芯穹

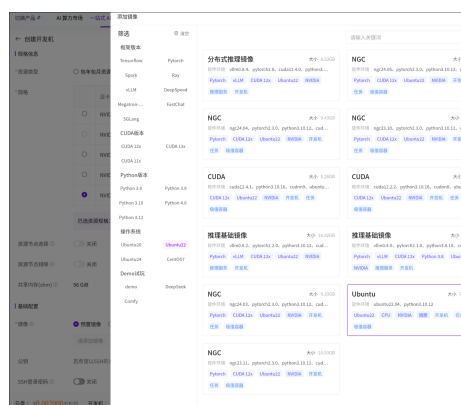
加入 ssh 公钥 点击头像, 选择“访问控制 → SSH 公钥管理“. 加入自己的公钥.

创建开发机 登陆后进入[控制台](#). 选择“算力租赁 → 创建开发机“.

使用“弹性资源“ 选择 RTX4090 单卡.



镜像使用 ubuntu2204.



创建后可以看到 ssh 信息.

miniconda 下载[安装脚本](#). `bash Miniconda3-latest...sh` 安装. 建议初始化 shell. 重新登陆后进入 conda 环境. 创建环境 `conda create -n torch python=3.12`.

torch & flash attention 安装步骤与容器云相同. 安装 torch 会自行安装 cuda runtime. 我们无需安装 CUDA.

Nsight System(Extra) 需要安装 [CUDA](#).

3.3 NanoGPT

[NanoGPT](#) 训练仅依赖于 torch. 直接 `git clone` 到本地.

NanoGPT 默认使用 openwebtext 数据集. 本次作业不关心训练数据, 我们可以修改部分代码使用 mock data 训练.

```
# train.py
def get_batch(_):
    x = torch.randint(
        0, 50000,
        (batch_size, block_size),
        device='cuda',
        dtype=torch.int64
    )
    y = torch.empty_like(x)
```

```
y[:, :-1].copy_(x[:, 1:])  
y[:, -1] = -1 # ignore index -1  
return x, y
```

python train.py 测试运行.

3.4 NanoVLLM

NanoVLLM 依赖于 Flash Attention 和 transformers. git clone 后, pip install -e . 编辑模式安装。

NanoVLLM 支持 Qwen3 系列 dense 模型推理, 可以使用 hfd.sh 下载模型 (需要 apt update && apt install -y aria2), 建议参考 [hf-mirror](#). ./hfd.sh Qwen/Qwen3-0.6B.

可以本地下载上传至容器.

下载并正确设置 Qwen3-0.6B 模型路径后. python bench.py 测试是否正常运行.

3.5 Nsight system GUI(Extra)

nsight system 的 profile 结果可以在本机上通过 GUI 查看. 在<https://developer.nvidia.com/nsight-systems/get-started>下载安装.

4 Profiling

4.1 Timing

CUDA Kernel 相对于 CPU 始终异步执行. 一次 cuda kernel 调用仅仅将该 kernel 放入启动队列, 而不代表完成. 我们可以通过 torch.cuda.synchronize() 强制等待 GPU 上所有 kernel 计算完成. 由此能够如下实现计时

```
import time  
import torch  
  
st = time.time()  
... # torch cuda ops
```



```
torch.cuda.synchronize()
ed = time.time()
print(f"Time: {ed - st}s")
```

然而同步会导致 kernel 队列清空, 同步点之后的代码可能会出现 CPU 瓶颈, 即 GPU 在等待 CPU 发送指令.

另一种方法是通过 CUDA Event 计时

```
import torch

st = torch.Event("cuda", enable_timing=True)
st.record() # or st.record(torch.cuda.current_stream())
... # code 1
ed1 = torch.Event("cuda", enable_timing=True)
ed1.record()
... # code 2
ed2 = torch.Event("cuda", enable_timing=True)
ed2.record()

ed2.synchronize()
print(f"Time: {st.elapsed_time(ed1) / 1e3}s")
print(f"Time: {ed1.elapsed_time(ed2) / 1e3}s")
```

CUDA Event record 不会同步 CPU/GPU, Overhead 更小, 适用于更细粒度的计时.

4.2 Nsight System(Extra)

本节介绍使用 nsight system (nsys) 对程序进行 system-level profile. nsys 指令默认包含在完整 CUDA 安装中 (e.g. /usr/local/cuda/bin/nsys).

A+B problem 对于 vector add 程序

```
# vec_add.py
import torch
```

```
a = torch.randn(4096, device='cuda', dtype=torch.float32)
b = torch.randn(4096, device='cuda', dtype=torch.float32)
c = a + b
torch.cuda.synchronize()
```

我们可以通过 `nsys profile -o result python vec_add.py` 获取 profile 文件.

下载后在本机打开, 可以看到 timeline. 进入 CUDA(HW) 能够看到实际 GPU 上执行的 kernel. 每个 kernel 都在一个 stream 上运行, 本例中为 default stream. 对于每个 kernel, 我们能够看到名称, 运行的开始与结束时间, grid/block size 等信息.

注意到 `randn` 由于在 `device: cuda` 上进行, `torch` 也调用了随机正态初始化 kernel.

NVTX 尽管 `nsight system` 会标注 kernel name. 但仍然难以判断 caller 在 host 代码的具体位置. 此时可以通过 `nvtx` 进行标注

```
# sgemv.py
import torch
import nvtx
a = torch.randn(
    (4096, 4096), device='cuda', dtype=torch.float32
)
b = torch.randn(
    (4096, 4096), device='cuda', dtype=torch.float32
)

for _ in range(3):
    a @ b

with nvtx.annotate("SGEMM"):
    for _ in range(10):
        a @ b
```

```
for _ in range(3):  
    a @ b  
  
torch.cuda.synchronize()
```

此时展开 stream timeline, 就可以看到中间 10 次 gemm 下方标注了 SGEMM.

5 Task

Extra 不用作答.

5.1 MFU

1 对 torch BF16 矩阵乘法计时, 计算 FLOPS. 查找 GPU Spec 中的理论峰值性能, 二者是否相同? 不要求严格 profile kernel. 建议在 30 行 python 内完成.

对于矩阵乘 $N \times N \times N$, 画出 FLOPS 关于 N 的折线图, 标注出理论 FLOPS。

2 对于 [GPT2 Layer](#). 计算出一个 Batch 训练的 Model FLOPs, 只考虑 Tensor Core 计算 (矩阵运算), 忽略 Norm, Softmax 等操作. 注意是单个 Layer. 此外, 本作业中要求计算**单向注意力** (Causal Attention) 的 FLOPs.

公式中各个变量的定义如下

B Batch size

S Sequence length per sample

H Hidden size

n Heads count

d Head size

I Intermediate size (FFN)

3 对于 [Llama Layer](#). 计算出一个 Batch 训练的 Model FLOPs.
RoPE 是否需要 Tensor Core?

4 对于 [Qwen3-8B Layer](#). 计算出一个 Batch 训练的 Model FLOPs. 之前的变量是否足以表示? 如果不能需要加入什么?

可以在 [modeling_qwen3.py](#) 看到 qwen3 的 transformers 实现.

5 对于 [Qwen3-8B Layer](#).

(1) 计算 Batch size 为 1 时, prefilling 的 Model FLOPs.

(2) 计算 Batch size 为 1 时, 有长度为 C 的 kv cache 的 decoding FLOPs.

本次作业中:

1. prefilling 指没有 kv cache 的 $[B, S]$ 输入的前向计算.
2. decoding 指有 kv cache, 新输入 $S = 1$ 的前向计算.

Extra Deepseek-V3 MOE Layer Training 的 Model FLOPs 如何计算?

5.2 Training

Config:

- GPT-medium
- batch size 128
- sequence length 1024
- micro batch size 8.

请注意 batch size 指一个 iteration 用于更新的数据量, 与 nanoGPT 中 batch size 不同.

1 修改 nanoGPT 代码

- (1) 使用 mock data,
- (2) 修改参数, 与 [GPT-medium](#) 对齐. bias 不作要求, 可以不加 bias.
- (3) 根据 GPU 规格 (peak BF16 FLOPS), 修改 MFU 公式, 仅包含矩阵乘 FLOPs. 除了 Layer 内部, 还有哪里有矩阵乘?

2

- (1) 对比开启与关闭 torch compile 的 MFU.
- (2) 在保证 batch size 不变的情况下, 修改 micro batch size 为 1-2-4-8. MFU 是多少? micro batch size 是否可以更大?
- (3) 为什么会观察到 (1)(2) 中的 MFU 差异? 查阅资料. 各用 30 个字左右解释.

4 将 model layer (attn+mlp) 的线性层修改为如下代码

```
def int8_quantization(x: torch.Tensor):
    scale = x.abs().max() / 127
    qx = (x / scale).round().to(torch.int8)
    return qx, scale

class INT8LinearFunc(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, w, bias):
        x_shape = list(x.shape)
        x = x.reshape(-1, x.size(-1))
        qx, sx = int8_quantization(x)
        qw, sw = int8_quantization(w)
        ctx.save_for_backward(qx, sx, w, bias)
        ctx.x_shape = x_shape
        y = torch._int_mm(qx, qw.T).to(x.dtype) * (sx * sw)
        if bias is not None:
            y += bias[...]
        return y.reshape(x_shape[:-1] + [-1])

    @staticmethod
    def backward(ctx, dy: torch.Tensor):
        dy = dy.reshape(-1, dy.size(-1))
        x_shape = ctx.x_shape
        qx, sx, w, bias = ctx.saved_tensors
        if bias is not None:
```

```

        db = dy.sum(dim=0)
    else:
        db = None
    qwt, swt = int8_quantization(w.T)
    qdy, sdy = int8_quantization(dy)
    dx = torch._int_mm(
        qdy, qwt.T
    ).to(dy.dtype).reshape(x_shape) * (swt * sdy)

    del qwt, swt
    qxt = qx.transpose(0, 1).contiguous()

    dw = torch._int_mm(
        qdy.T.contiguous(), qxt.T
    ).to(w.dtype) * (sx * sdy)

    return dx, dw, db

class INT8Linear(torch.nn.Linear):
    def forward(self, input):
        return INT8LinearFunc.apply(
            input, self.weight, self.bias
        )

```

单个矩阵乘法 kernel (`torch._int_mm`) 相较于 BF16 是否有加速? 训练是否有 End-to-end 加速? 此时训练中 GPU 有效吞吐为多少 TFLOPS?

Extra1 保持总的 batchsize 不变, 使用 2 张 GPU, DDP 训练. 此时 MFU 是多少? 是否达到 2 倍加速?

Extra2 使用 nsys profile 开启/关闭 torch compile, 单卡/多卡训练, INT8Linear or Not, 检查 timeline. 训练瓶颈在哪? Kernel Fusion 能够在哪些方面带来

收益?

5.3 Inference

1 推理 nanoVLLM, 运行自带的 `bench.py` 脚本, 获取 TPS. 将 `force_eager` 设为 `True`, 获取 TPS.

2 在 `bench.py` 的基础上拓展, 计算出每个 sample 推理的 Model FLOPs, 并由此计算出默认 benchmark 的 MFU. 能否达到训练的 MFU?

3 修改 `bench.py`, 固定每个 sample 的 query/output length.

(1) 从 1 逐步增大 query length, 保持固定的 output length. 画出此时 MFU 关于 query length 的曲线图.

(2) 从 1 逐步增大 output length, 保持固定的 query length. 画出此时 MFU 关于 output length 的曲线图.

建议用 1024 作为最大 input/output length. 实验方便, 结论清晰即可.

Extra 使用 `nsight system profile nanoVLLM. force eager or not. CUDA Graph` 对于推理有什么作用?

6 Submission

请提交单个 PDF 报告. 对于每个 task, 给出必要的说明. 图片/代码等请插入 PDF 中. 对于计算, 请给出简要思路. 对于实操, 可以给出代码片段. 比如 nanoGPT 修改为 GPT2-medium 模型规格, 具体修改了哪些参数?

截止日期 10 月 31 日.