

1 简介

我只有这一张显卡，但就是想跑 Qwen 3.8 27B。所以，这就是我搞出来的能用的方法了。

首先，注意到 RTX 4060 的显存是 8188 MiB。在初始化 CUDA 时，需要占用 115 MiB，同时显卡驱动还会占用大概 380 左右，可用空间是 7600 MiB 上下。我们需要在这个空间里放下模型本体和 131072 token 大小的 KV 缓存，同时保持足够的推理速度。

Windows 的 WDDM 会凭空占用 1-2 GB 的显存，用 WSL 也绕不过去。所以这一切都得在原生 Linux 上进行。

1.1 模型压缩

模型本体是 Prism ML 发布的 Ternary-Bonsai 三值量化。

我的另一篇文章三值量化模型的无损压缩讲述了如何通过重新打包量化格式，降低 3.5% 显存占用。这可以节省大概 200 MB 左右。

而且，如果不需要使用中文和英文之外的其他语言，还可以通过缩减词表的方式来减少额外的 60 MB。

Qwen 模型使用 BPE tokenizer，本质上是一张有向图，类似 ("un, "believable" -> "unbelievable") 这种。我们可以首先标出不需要的所有字符，例如谚文、天城文之类的，然后将含有它们的所有节点全部去掉。显然，这一定不会破坏有向图的连接。不过，中文和使用拉丁字母的文字已经占了大概 70%，所以这样去掉得不够多。可以利用词频统计来去掉拉丁字母区域里的大概一万多个单词，只能移除叶子，否则把 un 当成法语的不定冠词删掉就会让英语也坏掉了。

删除本身并不复杂，不需要去改 tokenizer。实际上，BPE 的所有信息都储存在 .gguf 的元数据区域里了，所以只要改变这个元数据，同时对应地删除列就好。已经有开源项目写了脚本把这些都搞好了，不过我忘记具体是什么项目了。

这个模型的 token embedding 和输出层的大小都是 248320×5120 ，加起来大概只有 27B 参数总量的 5% 左右，而且还只有输出层在显存里，token embedding 是在 CPU 中算出来的。所以即使删除了大约 60000 个词，我们大概也只能减少 60 MB，也就是 1%。不过这差不多值 5K token 呢，不要白不要。

1.2 KV cache 压缩

和模型不同，KV cache 很难无损压缩。指数位确实有重复的地方，但是压缩率最大只有 6%，而且还必须随着推理进行而解码，不像模型在运行前压缩完一次就不用再压缩了。所以这玩意挺难搞的，我就没做。

那么，最有效的手段还是量化，毕竟模型本体能只有 5200 MiB 左右也还是靠量化。单纯地用 llama.cpp 自带的最小量化 q4_0 会出问题，模型只会一直输出同一个字符，完全用不了。好在 Google 似乎最近搞出了一种新的技术，叫作 turbo quant，能在几乎无损的情况下把模型量化到每 32 个元素 14 字节的程度。目前 llama 主线是没有实现的，不过也有其他 fork 搞好了，只需要搬过来就行了。

1.3 修复 llama.cpp 的 bug

Qwen 有 48 个线性注意力层和 16 个传统的注意力层。然而，llama 会给线性注意力分配过多的空间，48 层加起来一共有 600 MiB。这确实是个 bug，最后是 AI 大人帮忙修的。我对大模型和推理框架确实没啥了解，只能依靠大肥鱼了。

此外，llama.cpp 会在计算注意力的时候将所有东西反量化到一个跟张量一样大的 fp16 数组里，只有 q4_0, q8_0 两种路径标注为 native，从而得以幸免。不过标为 native 并不需要编写新的算子，只需要套已有的模板和宏就可以了。我们可以为 turbo3 也专门编写一个。这样就能再省下来 300 MB 左右。

1.4 预测解码

预测解码是很常见的加速技术，也有各种不一样的实现方式。最常见的是 MTP，依靠模型外挂的额外训练权重预测下一个 token，并以此加速。效果很不错，可以加速 1.7 倍左右，但是问题在于需要的内

存有 1.4 GB。虽然 MTP 头的权重本身只有 200 MB 上下，但是它需要一些专门的 KV cache，而线性注意力层也需要更多的储存状态的空间。所以，MTP 大概是用不了的。

不过 llama 还自带了一些其他的预测解码，主要是基于 n-gram 模型的。它们的预测效果很难通过理论描述，主要是调参的事情了。这也是大肥鱼帮忙做的。我测试的主要是重构已有的代码，发现这组参数效果最好，大概可以加速 2.1 倍左右：

```
--spec-type ngram-map-k,ngram-suffix \  
--spec-ngram-suffix-n-max 48 \  
--spec-ngram-suffix-max-factor 3 \  
--spec-ngram-suffix-max-depth 32
```

不过 ngram-suffix 来自 llama 未合并的 PR。不管主线的维护者是怎么想的，反正我要用，所以就把它合并到本地了。这是后缀解码，也来自一篇论文，不过我暂时没时间去看。这个 PR 也算是未完成状态，原论文说的是基于 agent 以往的很多轮对话来解码，而这里我们只考虑了单次对话。原文似乎还有一种树状搜索的算法，不过我们这里并没有。但 2.1 倍的加速已经很不错了，瓶颈现在是 prefill，到后面就算开了 FlashAttention 大概也只有 300 t/s。