

## 1 简介

我最近看到了一个 Qwen 3.8 27B 的三值量化，名为 Bonsai 2，只需要 5.95 GB。然而，我只有 8G 显存，本地要用作 agent 的话又至少需要 131K 上下文。所以，我不得不尝试魔改各种东西，让它能塞下。

## 2 权重的无损压缩

我首先观察了 Bonsai 的权重数据。它主要的部分在于 401 个 PTQ1\_0 格式的张量，这是 Bonsai 自定义的格式，具体如下：

```
struct ptq1_0 {
    uint8_t qs[24]; // 存储 120 个 trit
    uint8_t qh[2];  // 存储 8 个 trit
    ggml_half d;    // 缩放
};
```

Trit 是一个三值单元，范围是 0 和  $\pm 1$ 。从信息论的角度， $\log_2 3 = 1.585$ ，所以一个 trit 需要 1.585 个比特；qs 用 1.6 个比特储存，已经无法压缩了。qh 则是 2 比特一个，而且 d 可能还有额外的冗余。因此，压缩是存在理论可能的。

理论存在，实践开始。我们需要先统计分布。我注意到：权重的分布非常接近均匀分布，而且没有可利用的结构，因此这里无法压缩。然而，d 作为一个 16 位浮点数，自然分为一个符号位、5 位指数和 10 位尾数。这里的尾数部分没法压缩，但是 5 位指数在差分后，信息熵仅有 0.41。符号位更是恒为正，因此完全无需储存。

这样的话，一个 28 字节、224 比特的单元的信息量其实只有

$$128 \log_2 3 + 0.41 + 10 = 213.28$$

字节数需要是整数，那么我们自然需要用 216 比特，也就是 27 字节来储存。然而，信息论给出的是理论下限，不代表实际上可以做到；就算能够设计合适的编码，也可能不适合 GPU 运算。这是因为，GPU 的每个块之间必须独立解码，而通常的 Huffman 或者 arithmetic coding 这种变长编码必须从头开始，无法随机访问。

所以，我采取了八个 PTQ1\_0 权重一组的形态：

```
struct ptq1_1 {
    uint8_t qs[192]; // 存储 960 个 trit
    uint8_t qh[13];  // 存储 64 个 trit
    uint10_t mantissa[8]; // 存储 8 个浮点尾数
    uint8_t exponents;
};
```

前面只需要 205 字节就可以打包下 1024 个 trit，原本需要 208 字节。之后的浮点数则一共只需要 11 字节，原本需要 16。这样就总体省下了 8 字节，模型将会缩小大约  $\frac{1}{28}$ 。这样打包出来的更大的块称作一个 **superblock**，原始的 PTQ1\_0 格式则是单个 **block**。

5.95 GB 的模型现在大概 5.7 GB。有什么用？完全没用，没人会去管 3.5% 的空间缩小。但无所谓了，因为山就在那里。

总之，其他的部分都好理解，问题是如何以 8 字节储存指数。其实，说 8 字节是不准确的，一共是  $8 + \log_2(3)$  字节，因为 qh[13] 原本可以储存 65 个 trit，这里却只有 64 个，所以有一个免费的 trit 空间可以用。

那么这一共有 768 种情况。在一个张量中，连续八个 ptq1\_0 块的指数分布实际上几乎不会超过 768 种，仅有大约万分之一的部分会落在这之外。因此，我们只需要令解码出 767 的时候，去一张外界的表里查询就可以了，这张表大约只有 20 KB 左右。

这就是完整的压缩算法，共计可以节省 200 MB 内存。代价则是我们需要重写一套算子，不过有 AI 大人在，一切都会好起来的。

### 3 MMVQ

在做这个东西之前，我只知道 GPU 的理论知识：什么是 SIMT，什么是 warp 和 thread block（后者在 CUDA 里似乎叫做 CTA）。然而，要写算子就需要知道更多了。

最基础的算子是矩阵-向量乘法 (GEMV)，在 llama.cpp 里叫做 mmvq。我们要计算的是向量  $d$ ，使得

$$d_r = \sum_{k=0}^{K-1} W_{r,k} \times x_k$$

不过，LLM 框架会为多用户推理服务设计。如果要同时为多个用户推理，那么我们可能会希望用同一个矩阵  $W$  处理多个  $x$ 。所以 GEMV 计算的其实是

$$d_{r,c} = \sum_{k=0}^{K-1} W_{r,k} \times x_{c,k}$$

注意这和矩阵乘法的下标  $x_{k,c}$  是相反的。

接下来，我们需要理解如何并行地计算它，也就是首先要设计一个线程到底计算哪部分。首先，我们自然可以拆分  $k$ ，这是矩阵的列长，在 Qwen 里有 5120, 6144, 17280 这三种选择，都很大，值得拆分。不重叠的  $k$  范围之间的计算是独立的，只需要最后加起来即可。同理， $r$  作为行长也是可以切分的，同样完全独立，甚至连相加都不需要。 $c$  则很小，大可不必继续分了。

那么，接下来是参数环节。到底每份多大合适呢？这里选择的是每个线程领取 128 份  $k$ ，而  $r$  则是当  $c \leq 4$  的时候取 4，否则取 2。这里  $r$  的选择和原有格式 (PTQ1\_0) 相同。我们将列长  $K$  除以 128 的结果记作 bpr (block per row)。那么，线程的序号 tid 自然可以改写为  $rg * bpr + kbx$ ，其中  $rg$  (row group) 即为行的偏移量，而  $kbx$  指示的自然就是  $k$  块自身的下标。

不过，CUDA kernel 是按照 CTA 分派的，而一个 CTA 中有 128 个线程，它们访问同一块 CTA 内部的共享内存，类似 CPU 中的 L1 cache。所以，我们需要先计算一个 CTA 可以装下多少行，以最大化利用它。这样，每个线程就可以处理完  $r$  行之后，再处理下一组  $r$  行，以此类推，直至占满整个共享内存。这样可以减少从 HBM 加载  $W_{r,k}$  的次数，而且在不同行之间  $x_{c,k}$  也可以完全复用。

那么，我们需要知道每个线程到底会占用多少。在我们的算子里，每一个线程都会将它的部分和完整地写入共享内存，总计  $r \times c \times k$  个。共享内存一共只能储存 4096 个 fp32，那么  $r$  的最大值即为  $\frac{4096}{c \times k}$ 。这个部分已经是 PTQ1\_0 为我们调好的。

接下来，我们就可以尝试编写算子主体了。

#### 3.1 数据格式

我们需要先找出每个线程对应的  $W$  在哪里。我们将八个 PTQ1\_0 块打包成了一个 PTQ1\_1，那么序号 tid 所在的 superblock sb 自然是  $kbx \gg 3$ ，而 superblock 内部，它访问的 block 序号则是  $sub = kbx \& 7$ 。

$W$  的输入放在指针  $vx$  中，而  $x$  则是  $vy$ 。那么

```
for (int i = 0; i < ROWS; ++i) {
    // 计算当前线程所在的行
    int r = std::min(rg * ROWS + i, n_rows_cta - 1);

    // 计算 PTQ1_1 超块的偏移量
    const int64_t w_off = (int64_t) (row0 + r) * stride_row_x + sb;
    auto b = (const block_ptq1_1 *) vx + w_off;

    const uint8_t *patt = ptq1_1_pattern(b, sb_base + w_off, codebook, esc_patt, esc_idx,
```

```

n_esc);
    qs[i] = b->qs[sub];
    tl[i] = ptq1_1_tail8(b, sub);
    d[i] = ptq1_1_scale(b, patt, sub);
}

```

这里 `qs[i]` 是指向 `uint8_t[24]` 的指针，`tl[i]` 是 `uint2`，而 `d` 则是 `float[i]`。

不过，`uint2` 并不是 `uint2_t`；它是两个 `uint`。容易发现，这跟 `PTQ1_0` 的格式完全一致。这样解码的话，就可以直接使用 `PTQ1_0` 的 `GEMV` 算法了。

至于  $x$ ，它是 `block_q8_1` 格式的：

```

typedef struct {
    union {
        struct {
            ggml_half d;
            ggml_half s;
        } data;
        ggml_half2 ds;
    };
    int8_t qs[32];
} block_q8_1;

```

这里的 `qs` 和 `d` 和之前的 `PTQ1_1` 格式相同，都是数据和缩放，不过 `d` 是整数。而  $s = d \sum_i qs_i$  则是所有数据之和。

我们编写的算子中， $x$  的排列方式称作 `PT`，并不是一个普通的数组。以  $k = 5120$  为例，应当有 40 个 `block`，每个 `block` 中有 128 个元素，因此会有 4 对  $(d, s)$ 。它们会这样重排：

```

plane 0 : [ block 0 ][ block 1 ] ... [ block 39 ]   元素 0.. 15
plane 1 : [ block 0 ][ block 1 ] ... [ block 39 ]   元素 16..31
...
plane 7 : [ block 0 ][ block 1 ] ... [ block 39 ]   元素 112..127
plane 8 : [ block 0 ][ block 1 ] ... [ block 39 ]   所有 (d, s)

```

重排之后，每一个 `block` 的长度变成了 16 字节。Plane 0-7 中，存储的都是 `qs` 的值，而 4 个  $(d, s)$  则储存在 plane 8 中。

假设一个原始 `block` 序号为  $i$ ，那它的第  $e$  个元素在 `PT` 格式下所在的字节就是

$$o = 16 \left( \left\lfloor \frac{e}{16} \right\rfloor \times \frac{K}{128} + i \right) + e \bmod 16$$

其中  $\frac{K}{128}$  自然是每个 `plane` 中 `block` 的数量。

## 3.2 反量化

我们已经计算出了数据块的位置，但还需要将它解包、重新展开才能计算。同时，我们还需要找到数据在块内部的位置才能乘起来。

我们按照 `PTQ1_0` 已经调优过的格式解码，会得到存储在 24 个字节中的 120 个 `trit`，以及存储在 2 字节中的 8 个 `trit`。这个格式中，`trit` 的存储不是连续的，并非第一个字节的 5 个 `trit` 就对应前五个权重。具体编码是这样的：对于前 80 个 `trit`，第  $e$  个权重位于第  $e \& 15$  个字节的第  $e \gg 4$  个 `trit`。所以，第一个字节存储的实际上是 0, 16, 32, 48, 64 这五个元素。对于后 40 个 `trit`，第  $e + 80$  个权重位于  $e \& 7$  字节处，是第  $e \gg 3$  个 `trit`。这个打包方式和前 80 个是一样的，只是小了一半而已。最后八个 2-bit 打包的是单独处理的。

要这样分开，我的理解是因为  $x$  的一个 `block` 是 16 个元素，而 120 则不是 16 的倍数。所以在 80 的位置分界似乎是合理的。不过总而言之，这就是 `Prism ML` 他们设计的，反正能跑，照着做就可以了，因为可以复用 `ptq1_0_*` 系列函数，不用自己从头开始写。

(未完待续，这些算子都是 AI 大人帮我写的，我自己光是看懂就要花很多精力了)