

ITQ3_S:High-Fidelity 3-bit LLM Inference

一、研究背景与核心痛点

1.重尾权重分布

设 w 为待量化的 b 比特权重向量

$$Q_b(w) = \Delta \cdot \left\lfloor \frac{w}{\Delta} + \frac{1}{2} \right\rfloor, \quad \Delta = \frac{w_{\max} - w_{\min}}{2^b - 1}$$

2. 在语义关键方向上累积。误差

二、核心创新方法——FWHT

使用快速沃尔什-哈达玛变换，将非均匀、存在离群点的数据分布转换为近似高斯分布的平滑形态

向量 $v \in \mathbb{R}^n$ （其中 $n = 2^k$ ）的沃尔什-哈达玛变换（WHT）定义为：

$$\hat{v} = H_n v, H_n = \frac{1}{\sqrt{n}} \begin{pmatrix} H_{n/2} & H_{n/2} \\ H_{n/2} & -H_{n/2} \end{pmatrix}, H_1 = [1] \quad (2)$$

将具有大峰值但功率谱不均匀的数据转换为平滑形态，使所有通道的数据方差趋于一致，消除了极端离群点的影响

二、核心创新方法——最优三值量化

最优非均匀量化：采用Lloyd-Max算法预计算量化码本，该算法能寻找最优量化级别，使量化后的均方误差（MSE）最小

For a Gaussian-distributed input $x \sim \mathcal{N}(0, \sigma^2)$, the mean squared error of ternary quantization with threshold α is:

$$\text{MSE}(\alpha) = \int_{-\infty}^{-\alpha} (x + \alpha)^2 \phi(x) dx + \int_{-\alpha}^{\alpha} x^2 \phi(x) dx + \int_{\alpha}^{\infty} (x - \alpha)^2 \phi(x) dx \quad (7)$$

where $\phi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/(2\sigma^2)}$ is the Gaussian density. Setting $d\text{MSE}/d\alpha = 0$:

$$\alpha^* = \sigma\sqrt{2} \cdot \text{erfinv}\left(\frac{2}{3}\right) \approx 0.798 \sigma \quad (8)$$

After the FWHT, the entries of $H\mathbf{w}$ are approximately $\mathcal{N}(0, \sigma^2)$ (Theorem 1), so we can compute α^* directly from the empirical standard deviation of the transformed block, giving a near-optimal scale without expensive second-order Hessian computation.

三、核心创新方法——TurboQuant融合CUDA核

TurboQuant 的核心贡献是将 256 点逆快速沃尔什-哈达玛变换 (Inverse FWHT) 融合到 MMQ 内核的共享内存加载阶段, 从而使去量化权重不会在全局内存中生成。该内核的执行流程如下:

Algorithm 2 ITQ3_S MMQ Dequantization Kernel (`load_tiles_itq3_s`)

Require: Packed quant $\mathbf{q}$, scale d_k , zero-point z_k

Ensure: Reconstructed weight tile in shared memory `smem`

- 1: **Load:** Fetch interleaved 3-bit quant $\mathbf{q}$ from global memory into registers
 - 2: **Unpack:** Bitfield-extract ternary values $\tilde{q}_j \in \{-1, 0, 1\}$ per thread
 - 3: **Dequantize:** $v_j \leftarrow d_k \cdot (\tilde{q}_j - z_k)$
 - 4: **Write** v_j to shared memory `smem_fwht[j]`
 - 5: **Synchronize:** `__syncthreads()`
 - 6: **for** $\text{step} \leftarrow 1, 2, 4, \dots, 128$ **do** $\triangleright \log_2 256 = 8$ butterfly stages
 - 7: $\text{lo} \leftarrow j \bmod (2 \cdot \text{step}); \text{hi} \leftarrow \text{lo} + \text{step}$
 - 8: $u \leftarrow \text{smem_fwht}[\text{lo}]; v \leftarrow \text{smem_fwht}[\text{hi}]$
 - 9: $\text{smem_fwht}[\text{lo}] \leftarrow u + v; \text{smem_fwht}[\text{hi}] \leftarrow u - v$
 - 10: **Synchronize:** `__syncthreads()`
 - 11: **end for**
 - 12: $\text{smem_fwht}[j] \leftarrow 0.0625 \cdot \text{smem_fwht}[j]$ $\triangleright$ Normalize: $1/\sqrt{256} = 0.0625$
 - 13: **Proceed** to matrix multiplication using `smem` as weight tile
-

Hadarmard矩阵

$$H_1 = [1]$$
$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$H_4 = \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$H_N = H_{2^n} = H_2 \otimes H_{2^{n-1}} = \begin{bmatrix} H_{2^{n-1}} & H_{2^{n-1}} \\ H_{2^{n-1}} & -H_{2^{n-1}} \end{bmatrix} = \begin{bmatrix} H_{\frac{N}{2}} & H_{\frac{N}{2}} \\ H_{\frac{N}{2}} & -H_{\frac{N}{2}} \end{bmatrix}$$

<http://2.blog.csdn.net/16011011>

Walsh-Hadamard Transform

$$X = \frac{1}{2^k} H_{2^k} x \text{ 或者 } X = H_{2^k} x$$

$$H_1 = [1] \quad H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_8 = H_2 \otimes H_4 = \begin{bmatrix} H_4 & H_4 \\ H_4 & -H_4 \end{bmatrix} = \begin{bmatrix} H_4 & 0 \\ 0 & H_4 \end{bmatrix} \begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix} = \begin{bmatrix} H_2 & H_2 & 0 & 0 \\ H_2 & -H_2 & 0 & 0 \\ 0 & 0 & H_2 & H_2 \\ 0 & 0 & H_2 & -H_2 \end{bmatrix} \begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix}$$

$$= \begin{bmatrix} H_2 & 0 & 0 & 0 \\ 0 & H_2 & 0 & 0 \\ 0 & 0 & H_2 & 0 \\ 0 & 0 & 0 & H_2 \end{bmatrix} \begin{bmatrix} I_2 & I_2 & 0 & 0 \\ I_2 & -I_2 & 0 & 0 \\ 0 & 0 & I_2 & I_2 \\ 0 & 0 & I_2 & -I_2 \end{bmatrix} \begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix}$$

$$= [G_0][G_1][G_2]$$

$$W(u) = \frac{1}{8} H_8 f(x) = \frac{1}{8} [G_0][G_1][G_2]f(x)$$

令：

$$[f_1(x)] = [G_2][f(x)]$$

$$[f_2(x)] = [G_1][f_1(x)]$$

$$[f_3(x)] = [G_0][f_2(x)]$$

则：

$$W(u) = \frac{1}{8} f_3(x)$$

<https://blog.csdn.net/tylwin>

$$[f_1(x)] = [G_2][f(x)]$$

双正

双正

$$\begin{bmatrix} f_1(0) \\ f_1(1) \\ f_1(2) \\ f_1(3) \\ f_1(4) \\ f_1(5) \\ f_1(6) \\ f_1(7) \end{bmatrix} = [G_2] \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} = \begin{bmatrix} f(0) + f(4) \\ f(1) + f(5) \\ f(2) + f(6) \\ f(3) + f(7) \\ f(0) - f(4) \\ f(1) - f(5) \\ f(2) - f(6) \\ f(3) - f(7) \end{bmatrix}$$

双击此处

$$[f_2(x)] = [G_1][f_1(x)]$$

$$\begin{bmatrix} f_2(0) \\ f_2(1) \\ f_2(2) \\ f_2(3) \\ f_2(4) \\ f_2(5) \\ f_2(6) \\ f_2(7) \end{bmatrix} = [G_1] \begin{bmatrix} f_1(0) \\ f_1(1) \\ f_1(2) \\ f_1(3) \\ f_1(4) \\ f_1(5) \\ f_1(6) \\ f_1(7) \end{bmatrix} = \begin{bmatrix} f_1(0) + f_1(2) \\ f_1(1) + f_1(3) \\ f_1(0) - f_1(2) \\ f_1(1) - f_1(3) \\ f_1(4) + f_1(6) \\ f_1(5) + f_1(7) \\ f_1(4) - f_1(6) \\ f_1(5) - f_1(7) \end{bmatrix}$$

双击此处

$$[f_3(x)] = [G_0][f_2(x)]$$

$$\begin{bmatrix} f_3(0) \\ f_3(1) \\ f_3(2) \\ f_3(3) \\ f_3(4) \\ f_3(5) \\ f_3(6) \\ f_3(7) \end{bmatrix} = [G_0] \begin{bmatrix} f_2(0) \\ f_2(1) \\ f_2(2) \\ f_2(3) \\ f_2(4) \\ f_2(5) \\ f_2(6) \\ f_2(7) \end{bmatrix} = \begin{bmatrix} f_2(0) + f_2(1) \\ f_2(0) - f_2(1) \\ f_2(2) + f_2(3) \\ f_2(2) - f_2(3) \\ f_2(4) + f_2(5) \\ f_2(4) - f_2(5) \\ f_2(6) + f_2(7) \\ f_2(6) - f_2(7) \end{bmatrix}$$

双击此处添加标题

```
def fwht(x: torch.Tensor) -> torch.Tensor:
    x = x.clone()
    n = x.size(0)
    h = 1
    while h < n:
        for i in range(0, n, h * 2):
            for j in range(i, i + h):
                u = x[j]
                v = x[j + h]
                x[j] = u + v
                x[j + h] = u - v
            h *= 2
    return x
```

```
def ifwht(x: torch.Tensor) -> torch.Tensor:
    x = fwht(x)
    return x / x.size(0)
```

离线量化流水线

Algorithm 1 ITQ3-S Offline Quantization

Require: Weight tensor $\mathbf{W} \in \mathbb{R}^{M \times N}$, block size $n = 256$

Ensure: Quantized tensor $\mathbf{Q}$, scales $\mathbf{d}$, zero-points $\mathbf{z}$

- 1: **for** each block $\mathbf{w} \in \mathbb{R}^{256}$ in $\mathbf{W}$ **do**
 - 2: $\mathbf{w}' \leftarrow \text{FWHT}(\mathbf{w})$ ▷ Fast Walsh-Hadamard Transform
 - 3: $d_k \leftarrow \frac{\alpha^*}{\sigma(\mathbf{w}')}$ ▷ Optimal ternary scale (Section 3)
 - 4: $z_k \leftarrow -\text{round}(\mu(\mathbf{w}')/d_k)$ ▷ Zero-point offset
 - 5: $\mathbf{q} \leftarrow \text{Clamp}(\text{round}(\mathbf{w}'/d_k) + z_k, -1, 1)$ ▷ Ternary quantization
 - 6: Store(Pack_{3b}($\mathbf{q}$), d_k , z_k)
 - 7: **end for**
-

Definition 1 (ITQ3-S Encoding). For weight vector $\mathbf{w} \in \mathbb{R}^{256}$, the ITQ3-S encoding function is:

$\mathcal{X}$

$$Encode(\mathbf{w}) = \left(Pack_{3b} \left(Clamp \left(\left\lfloor \frac{H\mathbf{w}}{d_k} + z_k + 0.5 \right\rfloor, -1, 1 \right) \right), d_k, z_k \right)$$

Each ternary value $q \in \{0, 1, 2\}$ (representing $\{-1, 0, +1\}$ with zero-point $z = 1$) is encoded in 3 bits. For a block of 256 values, we interleave two 4-bit nibble streams to form 32-bit words aligned for DP4A:

$$word_i = \bigoplus_{j=0}^7 (q_{8i+j} \bmod 4) \ll 4j \quad (\text{for even } i) \quad (9)$$

The high bit of each 4-bit nibble encodes the interleave selector, allowing the CUDA dequantization kernel to reconstruct full 3-bit values from two consecutive nibbles using a single 32-bit load and bitfield extraction, maximizing L1 cache utilization.