QuIP#: Even Better LLM Quantization with Hadamard Incoherence and Lattice Codebooks

Albert Tseng[*1] Jerry Chee[*1] Qingyao Sun[2] Volodymyr Kuleshov[1] Christopher De Sa[1]

Abstract

Post-training quantization (PTQ) reduces the memory footprint of LLMs by quantizing their weights to low-precision. In this work, we introduce QuIP#, a weight-only PTQ method that achieves state-of-the-art results in extreme compression regimes ($\leq$ 4 bits per weight) using three novel techniques. First, QuIP# improves QuIP's (Chee et al., 2023) incoherence processing by using the randomized Hadamard transform, which is faster and has better theoretical properties. Second, QuIP# uses vector quantization to take advantage of the ball-shaped sub-Gaussian distribution that incoherent weights possess: specifically, we introduce a set of hardware-efficient codebooks based on the highly symmetric E_8 lattice, which achieves the optimal 8-dimension unit ball packing. Third, QuIP# uses fine-tuning to improve fidelity to the original model. Our experiments show that QuIP# outperforms existing PTQ methods, enables new behaviors in PTQ scaling, and supports fast inference. Our code can be found at `https://github.com/Cornell-RelaxML/quip-sharp`.

1. Introduction

Large language models (LLMs) have driven rapid advances across diverse fields such as natural language processing (Touvron et al., 2023b), scientific modeling (Nguyen et al., 2023), and program synthesis (Rozière et al., 2024). However, the massive size of these models poses significant challenges to their deployment. For example, the largest model in the Llama 2 family has 70B parameters, and requires 140GB of GPU memory in native 16-bit precision (Touvron et al., 2023b). This massive memory footprint motivates research into lossless LLM compression methods.

[*]Equal contribution [1]Department of Computer Science, Cornell University [2]Department of Operations Research and Information Engineering, Cornell University. Correspondence to: Albert Tseng <albert@cs.cornell.edu>, Jerry Chee <jerrychee@cs.cornell.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

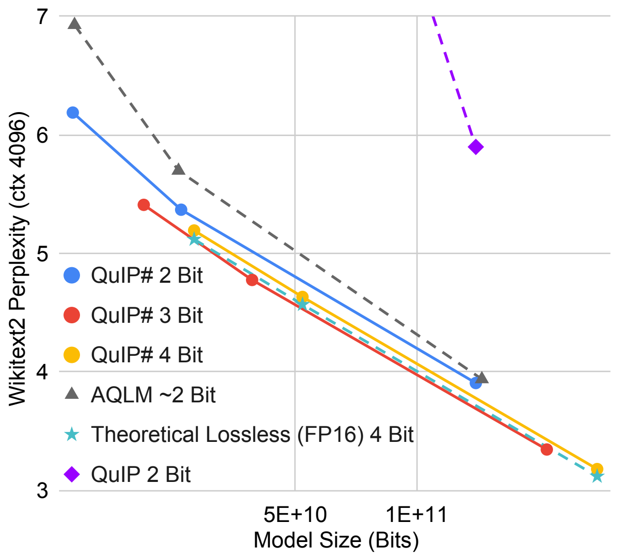

Figure 1. QuIP# offers unprecedented quantization quality at extreme compression ratios. QuIP# 3-bit models also scale better than theoretically lossless 4-bit models, a previously unseen result.

Post-training quantization (PTQ) linearly reduces the memory footprint of models by storing trained weights with less precision. For example, Llama 2 70B only requires $<$ 20GB of memory when quantized to 2 bits. This not only lets large models fit on smaller devices, but also enables faster throughput in memory bound settings such as autoregressive decoding. However, existing quantization methods either do not scale to extreme compression ratios (Shao et al., 2024) or have expensive decoding schemes (Egiazarian et al., 2024), motivating the development of both good *and* fast PTQ methods.

In this work, we introduce QuIP#, a *weight-only* PTQ method that achieves a new state-of-the-art in model quantization. QuIP# improves over existing work via three techniques: incoherence processing, lattice codebooks, and fine-tuning. Incoherence processing is a principled form of outlier suppression that produces approximately Gaussian distributed weight matrices (Chee et al., 2023). QuIP#

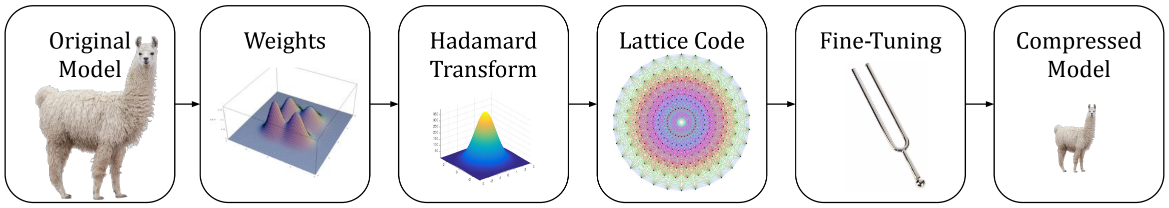

Figure 2. QuIP# performs incoherence processing with a Randomized Hadamard Transform and uses lattice codebooks to achieve state-of-the-art quantized models.

performs incoherence processing with the computationally-efficient randomized Hadamard transform (Halko et al., 2011) (Section 3). To quantize incoherent matrices, QuIP# uses the BlockLDLQ block adaptive rounding algorithm with compressible codebooks based on the E_8 lattice, which achieves the highest density 8 dimensional unit-ball packing (Viazovska, 2017) (Section 4). The E_8 lattice is highly structured and symmetric, allowing our codebooks to be hardware-friendly and admit fast inference. Finally, QuIP# includes an inter-layer fine-tuning algorithm that further improves quantization quality (Section 5).

QuIP# significantly outperforms existing PTQ methods including OmniQuant (Shao et al., 2024), QuIP (Chee et al., 2023) (a previous, separate work), and AQLM (Egiazarian et al., 2024). To the best of our knowledge, QuIP# is also the first PTQ method where 3-bit models scale better than 4-bit models. This directly refutes Dettmers & Zettlemoyer (2023)'s claim that 4-bit models are "optimal" and indicates that as the field of PTQ develops, 2-bit models are likely to scale better than 3-bit models in the near future. Moreover, QuIP# was designed from the ground up to be fast. Algorithm 2 describes fast inference with a QuIP#-quantized linear layer. Our "proof of concept" CUDA implementation of QuIP# achieves over 50% of peak memory bandwidth on a NVIDIA RTX 4090, validating our design choices.

In summary, we introduce QuIP#, a post-training quantization method that achieves state-of-the-art results by

1. Performing incoherence processing with the Randomized Hadamard Transform, which has better incoherence properties and faster runtime than the Kronecker factorization in QuIP.
2. Rounding incoherence-processed weight matrices with block adaptive rounding and codebooks based on the E_8 lattice, which achieves the highest 8-dimension unit ball packing density (kissing number).
3. Introducing an inter-layer fine-tuning algorithm that further improves quantization quality.

Algorithm 1 QuIP# without Fine-Tuning (QuIP#-NoFT)

```
input Weight $W \in \mathbb{R}^{m \times n}$, hessians $H \in \mathbb{R}^{n \times n}$, $g$-dim. $k$-bit codebook $C$
  $\hat{W}, \hat{H}, S_U, S_V \leftarrow$ IP-RHT$(W, H)$ (Alg. 3)
  $\hat{W} \leftarrow$ BlockLDLQ$(\hat{W}, \hat{H}, C)$ (Sec. 4.1)
output $\hat{W}, S_U, S_V$
```

Algorithm 2 QuIP# Inference (for a Linear Layer)

```
input $\hat{W}, S_U, S_V$ from Alg. 1, $g$-dim. $k$-bit codebook $C$, input $x \in \mathbb{R}^n$.
  $y \leftarrow \texttt{Had}(S_V \odot x)$ where Had performs
      an orthogonal Hadamard transform (Sec. 3)
  $y \leftarrow \texttt{decompress\_multiply}(\hat{W}, C, y)$
  $y \leftarrow \texttt{Had}(S_U \odot y)$
output $y$
```

2. Background / Related Work

2.1. Compressing LLMs

A large body of work has focused on compressing LLMs, as doing so can directly benefit LLM inference at scale. Methods such as pruning, quantization aware training (QAT), and post-training quantization (PTQ) all focus on different areas of this problem and are not strictly orthogonal to each other. Pruning *removes* weights from models while preserving model quality and inference performance (Chee et al., 2022; Sun et al., 2023). QAT focuses on training models that are more "quantizable" but usually requires training models from scratch (Nagel et al., 2022; Xi et al., 2023). PTQ, which QuIP# falls under, instead quantizes *pre-trained* models. PTQ requires less compute than QAT and achieves competitive performance (Chee et al., 2023; Frantar et al., 2023; Shao et al., 2024; Egiazarian et al., 2024). For the rest of this paper, we focus on PTQ.

2.2. Quantization and Adaptive Rounding

In QuIP#, we follow existing state-of-the-art PTQ methods and round weights to minimize the per-layer proxy loss, as formalized by Nagel et al. (2020):

$$\ell(\hat{W}) = E_x\left[\|(\hat{W} - W)x\|^2\right] \quad (1)$$

$$= \operatorname{tr}\left((\hat{W} - W)H(\hat{W} - W)^T\right). \quad (2)$$

Here, $W \in \mathbb{R}^{m\times n}$ is the original weight matrix in a linear layer, $\hat{W} \in \mathbb{R}^{m\times n}$ are the quantized weights, $x \in \mathbb{R}^n$ is an input vector drawn uniformly at random from a calibration set, and $H = E_x[xx^T]$ is a proxy Hessian. This intra-layer formulation makes quantization tractable for LLMs. One way to minimize ℓ is to use adaptive rounding methods that iteratively round weight matrices by considering the current rounding error for that specific matrix. For example, the LDLQ[1] rounding algorithm iteratively rounds rows of model weights using linear feedback from quantization error of already rounded rows. LDLQ is optimal within the class of adaptive rounding methods with linear feedback and offers provably better error rates than nearest or stochastic rounding (Chee et al., 2023).

2.3. Incoherence Processing

Multiple works have observed that outliers in model activations and weights can hinder quantization quality, motivating methods that "suppress" outliers during quantization. For example, AWQ (Lin et al., 2023) scales model weights by information from activations and OmniQuant (Shao et al., 2024) uses simple learnable model-preserving transformations. However, these heuristic-based approaches tend to fail at lower bitrates.

Instead, in QuIP, Chee et al. (2023) proposed that *incoherence* is important for LLM quantization. Informally, incoherent matrices have concentrated entry magnitudes—ruling out outliers. In LLMs, incoherent weight and Hessian matrices mean that both the thing being rounded (weights) and important rounding directions (Hessians) are not too large in any coordinate. This enables quantization with *provably* bounded error.

Definition 2.1 (Chee et al. (2023)). A Hessian $H \in \mathbb{R}^{n\times n}$ is μ-incoherent if its eigendecomposition $H = Q\Lambda Q^T$ has

$$\max_{i,j} |Q_{ij}| = \max_{i,j} |e_i^T Q e_j| \leq \mu/\sqrt{n}.$$

A weight matrix $W \in \mathbb{R}^{m\times n}$ is μ-incoherent if

$$\max_{i,j} |W_{ij}| = \max_{i,j} |e_i^T W e_j| \leq \mu\|W\|_F/\sqrt{mn}.$$

To exploit incoherence, Chee et al. (2023) introduced *incoherence processing* as a part of their quantization method QuIP. QuIP's incoherence processing works by conjugating W and H by structured random orthogonal matrices. Specifically, QuIP constructs orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ via a Kronecker product by drawing uniform random orthogonal matrices U_1, U_2 (of sizes about $\sqrt{n}$), V_1, and V_2 (of sizes about $\sqrt{m}$) and setting $U = U_1 \otimes U_2$ and $V = V_1 \otimes V_2$. If we assign $\tilde{H} \leftarrow VHV^T$ and $\tilde{W} \leftarrow UWV^T$, $\tilde{H}$ and $\tilde{W}$ become $\tilde{O}(1)$-incoherent with high probability (see their Lemma 5). Note that this transformation preserves the proxy objective, as $\operatorname{tr}\left((UWV^T)(VHV^T)(VW^TU^T)\right) = \operatorname{tr}\left(WHW^T\right)$. After quantizing the transformed weight matrix $\tilde{W}$ using $\tilde{H}$, during inference, QuIP-quantized models transform model activations x with V and U^T to compute

$$U^T(\text{quantized}(\tilde{W})(Vx)) \approx U^T(\tilde{W}(Vx)) = Wx.$$

These structured orthogonal multiplies by a Kronecker product lead to a runtime overhead of $\Theta(n\sqrt{n} + m\sqrt{m})$, which is small relative to the $\Theta(mn)$ cost of the multiply by W.

Incoherence processing can be seen as a principled alternative to more complicated and heuristic methods for outlier suppression. Methods such as grouping and keeping outliers in FP16 require extra storage and can negatively impact performance. For example, using a 16 bit scale per group of 64 weights requires an extra 0.25 bits per weight. This increase is significant in extreme compression regimes, whereas incoherence processing has minimal inference overhead and allows more bits to be spent on actually quantizing model weights. Alternatively, keeping outliers in high precision requires storing unstructured high precision matrices, which are slow to multiply by.

2.4. Vector Quantization

Prior PTQ works have focused on quantizing each scalar weight W_{ij} individually, amounting to scalar quantization (SQ) (Chee et al., 2023; Lin et al., 2023; Shao et al., 2024). However, SQ is subotimal as it ignores the shape of the source distribution. Vector quantization (VQ) instead quantizes a group of d weights together as a d dimensional vector. In k-bit VQ, a vector is quantized to one of 2^{kd} vectors $\in \mathbb{R}^d$ that form a $2^{kd} \times d$ codebook C. By shaping C to the source distribution of W, VQ can achieve lower distortion than SQ, with higher d enabling better shaping (Kostina & Verdú, 2011).

However, VQ has exponential cost in both the bitrate and vector dimension. As such, VQ can be expensive and can have limited distortion gains over SQ due to practical constraints on d. For example, for fast inference on GPUs, C must fit in L1 cache even after bank conflicts (32$\times$ duplication). This means that kd can be at most ≈ 10 for an unstructured C. In QuIP$\#$, we mitigate these issues by using a highly structured 2-bit codebook based on the 8D E_8 lattice,

[1]OPTQ (Frantar et al., 2023) and QuIP independently introduced alternative formulations of this rounding method, and QuIP showed them to be equivalent. LDLQ is the name given by QuIP.

Algorithm 3 Incoherence Processing with RHT (IP-RHT)

```
input $W \in \mathbb{R}^{m\times n}, H \in \mathbb{R}^{n\times n}$
    Sample sign vectors $S_V \sim \mathcal{U}\{\pm 1\}^n, S_U \sim \mathcal{U}\{\pm 1\}^m$
    $\hat{W} \leftarrow \texttt{Had}(diag(S_U)\texttt{Had}(diag(S_V)W^T)^T)$ where
        Had is the Hadamard transform (sec. 3)
    $\hat{H} \leftarrow \texttt{Had}(diag(S_V)\texttt{Had}(diag(S_V)H)^T)$
output $\hat{W}, \hat{H}, S_U, S_V$
```

E8P. E8P achieves $kd = 16$ but can be compressed $256\times$, allowing it to fit in GPU cache.

2.5. Fine-Tuning vs. Quantization Aware Training

Fine-tuning (FT) for LLM PTQ was introduced in AQLM (Egiazarian et al., 2024) as a tractable way to capture inter-layer interactions. As presented in AQLM and here, fine-tuning is essentially a hybrid method between pure PTQ and full QAT that requires significantly less data and compute than full QAT. With QuIP#, fine-tuning generally matches the performance of QAT, with the caveat that QAT for LLMs is a relatively underexplored area. For example, with some extrapolation, LLM-QAT (Liu et al., 2023) 4 bit (4-16-16) performs around the same as QuIP# 4 bit with or without FT. However, QuIP# can quantize a 70B parameter model in a few hours on a single 8 GPU node while LLM-QAT needs 960 GPU-hours to generate training data alone. Since fine-tuning for PTQ is a very recent development, both the methods presented here and in AQLM are almost certainly not optimal. However, they serve to show that FT is a relatively cheap way to achieve QAT-quality models, making such an approach practical and promising.

3. Incoherence Processing with the Randomized Hadamard Transform

In this section, we propose a way of improving the incoherence processing of QuIP by replacing the 2-factor Kronecker product by a Randomized Hadamard Transformation (RHT) (Halko et al., 2011). This change yields three advantages: (1) the theoretical bound on the incoherence parameter μ is improved; (2) the asymptotic cost of multiplying by the structured random orthogonal matrix is improved from $\Theta(n\sqrt{n})$ to $\Theta(n \log n)$; (3) the cost to multiply is further reduced by a constant factor, since a Hadamard matrix multiply can be performed without any floating-point multiplies as its entries are in $\{-1, +1\}$. Additionally, we show in Section 6.4 that this change by itself improves the perplexity of quantized LLMs.

Recall from section 2.3 that one way to efficiently perform incoherence processing is to conjugate W and H by structured random orthogonal matrices. QuIP# uses the RHT, which performs $x \to VSx$ where $V \in \mathbb{R}^{n\times n}$ is a Hadamard matrix, S is a random sign vector $\{\pm 1\}^n$, and $x \in \mathbb{R}^n$. The RHT can be computed in $O(n \log n)$ time with the Fast Walsh-Hadamard Transform (Fino & Algazi, 1976) when n is a power of 2. We will temporarily assume that all dimensions are powers of 2. Later in the section we will explain 2 methods for incoherence processing when the dimension is not a power of 2.

Lemma 3.1. *Let H be any positive semidefinite matrix on $\mathbb{R}^{n\times n}$ and W any weight matrix on $\mathbb{R}^{m\times n}$. Let $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ be orthogonal scaled Hadamard matrices. Let $S_U \in \mathbb{R}^{m\times m}$ and $S_V \in \mathbb{R}^{n\times n}$ be random diagonal matrices with independent diagonal elements drawn uniformly from $\{-1, +1\}$. Then for any $\delta > 0$, $VS_VHS_VV^T$ is μ_H-incoherent with probability at least $1-\delta$, and $US_UWS_VV^T$ is μ_W-incoherent with probability at least $1-\delta$, where*

$$\mu_H = \sqrt{2\log\left(\frac{2n^2}{\delta}\right)} \quad \text{and} \quad \mu_W = 2\log\left(\frac{4mn}{\delta}\right).$$

In QuIP (Chee et al., 2023), the 2-factor Kronecker approach achieves $\mu_W^{Kron} = A^2 \log(4Cmn/\delta)^2$, where A and C are global constants independent of n and the number of factors. QuIP#'s RHT achieves superior incoherence via a log dependence on the matrix size rather that the Kronecker method's log-squared dependence. All of QuIP's theory analyzing the proxy loss in Eq. (1) still holds with the RHT, with the improved incoherence rates propagating through.

Now, what about dimensions n that are not powers of 2? In most cases, we can factorize $n = pq$ where that p is the largest power of 2 such that there exists a known Hadamard matrix of size q. This allows us to construct $V \in \mathbb{R}^{n\times n} = H_p \otimes H_q$ where H_p and H_q are size p and q Hadamard matrices, respectively. Then we can compute VSx in $O(q^2 p \log p)$ time, which is faster than the $O(n(p+q))$ time of QuIP's 2-factor Kronecker approach when $p \gg q$. For example, Llama 2 70B has intermediate dimension $28672 = 1024 * 28$; $1024 \gg 28$. Algorithm 3 describes how to perform incoherence processing with the RHT. Doing so requires storing two sign vectors $S_U \in \{\pm 1\}^m$ and $S_V \in \{\pm 1\}^n$. Since $n, m \gg 1000$ for LLMs, S_U and S_V add less than 0.01 bits per weight (see Section F.1 for more details).

While the Hadamard conjecture states that $\exists H_k \forall k, 4 \mid k$, finding such Hadamard matrices is still an open problem (Hedayat & Wallis, 1978). In cases when there does not exist a factorization $n = pq$ where $\exists H_p, H_q$, we present a Randomized Fast Fourier Transform (RFFT) incoherence processing algorithm with similar runtime and concentration properties as the RHT. At a high level, the RFFT performs incoherence processing with the Fast Fourier Transform (FFT) (Cochran et al., 1967) and a random complex phase. The RFFT only requires n to be even, which is much weaker than the RHT's restrictions on n. The RFFT is also useful when there *does exist* a decomposition $n = pq$ but $p \not\gg q$, resulting in reduced speedups over an $\Theta(n\sqrt{n})$ algorithm. The FFT itself is also well supported on a wide variety of hardware, meaning that it may be easier to implement a fast RFFT when adapting QuIP# to new hardware. In practice, we find that the RFFT performs slightly worse than the RHT but still achieves strong results (Table 1). We describe the RFFT in detail in Section A.2 in the Appendix.

Table 1. RHT vs. RFFT incoherence processing using 2 Bit QuIP# (no FT). WikiText2 perplexity (↓), context length 4096.

INCOHERENCE	2-7B	2-13B	2-70B
HADAMARD	8.22	6.06	4.16
FOURIER	8.30	6.08	4.17

4. BlockLDLQ and Lattice Codebooks

It follows from the central limit theorem that RHT-transformed weights follow a roughly ball-shaped Gaussian distribution. However, rounding weights one at a time, as QuIP does with its LDLQ, ignores this shaping—producing a set of representable weight vectors that is shaped like a hypercube rather than a ball. Vector quantization (VQ) lets us shape codebooks to better match the source distribution. VQ codebooks quantize multiple weights to a single codebook entry, and we design the overall shape of our codebook to better match the roughly ball shape of the RHT transformed weights. In Section 4.1, we introduce BlockLDLQ, which adaptively rounds blocks of weights with VQ. Within BlockLDLQ's VQ step, QuIP# uses the 2 bit E8P codebook (Section 4.2). E8P is based on the E_8 lattice, which achieves the highest density unit ball packing in $\mathbb{R}^8$ (Viazovska, 2017). E8P achieves good shaping while enabling fast inference by only needing to look up from a 256×8 codebook.

4.1. Adaptive Rounding for Vector Quantization

Chee et al. (2023) formulated a class of adaptive rounding algorithms with linear feedback. These methods round columns one at a time with linear feedback a_k from the already rounded columns. Specifically, columns of a weight matrix $W \in \mathbb{R}^{m\times n}$ are iteratively rounded for $k = 1, 2, \ldots, n$: $\hat{W}_k = \mathcal{Q}(W_k + (W_{:(k-1)} - \hat{W}_{:(k-1)})a_k)$, where W_k is the k-th column of W, $W_{:(k-1)}$ is the first $k-1$ columns of W, $\mathcal{Q}$ performs nearest or stochastic rounding, and $a_k \in \mathbb{R}^{k-1}$. The resulting $\hat{W}$ satisfies $\hat{W} = \mathcal{Q}(W + (W - \hat{W})U)$, where $U \in \mathbb{R}^{n\times n}$ is a upper triangular matrix whose columns are a_k and $\mathcal{Q}$ acts elementwise.

The LDLQ algorithm sets U to be $L^T - I$ where $H = L^T DL$ is the LDL decomposition of the proxy Hessian H. From QuIP, we know that LDLQ is optimal within adaptive rounding methods with linear feedback when rounding to the integers. However, LDLQ does not work with vector quantization, which rounds multiple columns together. Here, we extend LDLQ to support vector quantization. Given a block size g that evenly divides n, our block LDLQ is based on a novel g-block LDL decomposition $H = \mathbf{L}^T\mathbf{D}\mathbf{L}$, where $\mathbf{L}$ is a unit block lower triangular matrix (among the n^2/g^2 $g \times g$ blocks of $L \in \mathbf{R}^{n\times n}$, the n/g diagonal blocks are all I and all blocks above the diagonal are 0), and $\mathbf{D}$ is a block diagonal matrix.[2] As before, we set $\mathbf{U} = \mathbf{L}^T - I$, and round W in a block-wise fashion via

$$\hat{W}_k = \mathbf{Q}(W_k + (W_{:(k-1)} - \hat{W}_{:(k-1)})\mathbf{A}_k),$$

where $\mathbf{A}_k \in \mathbb{R}^{n\times g}$ contains the $k-g+1$ through k-th columns of $\mathbf{U}$ (the kth block), W_k similarly denotes the kth block of W, and $\mathbf{Q}$ denotes a vector quantizer. As in the original QuIP paper, we can bound the error of this method.

Theorem 4.1. *Suppose that we round $W \in \mathbb{R}^{m\times n}$ using g-block LDLQ with Hessian H, producing $\hat{W}$. Suppose that H is μ-incoherent, and that we use a (possibly stochastic) vector quantizer $\mathbf{Q}$ that satisfies $\mathbf{E}[(\mathbf{Q}(x) - x)(\mathbf{Q}(x) - x)^T] \preceq \sigma^2 I$ for any $x \in \mathbb{R}^g$. Then*

$$\mathbf{E}[\operatorname{tr}((\hat{W} - W)H(\hat{W} - W)^T)] \leq \tfrac{gm\mu^2\sigma^2}{n} \operatorname{tr}(H^{1/2})^2.$$

Observe that under the same conditions, just quantizing all blocks independently would yield $\mathbf{E}[\operatorname{tr}((\hat{W} - W)H(\hat{W} - W)^T)] \leq gm\sigma^2 \operatorname{tr}(H)$: this "improvement" from the trace of H to the square of the trace of its square root divided by n is the same factor achieved in the scalar case in QuIP.[3]

4.2. The E8P ("E8 Padded") Codebook

BlockLDLQ relies on an internal vector quantization (VQ) step $\mathbf{Q}$ that rounds a d-dimension (g in the previous section) vector to a codebook C. To effectively apply VQ, C should be shaped like the source distribution and have high packing density. One way to improve shaping is by increasing d. However, recall from Section 2.4 that to quantize a vector $v \in \mathbb{R}^d$ to k bits with VQ, C must have size $2^{kd} \times d$. Since the codebook size is exponential in both the vector dimension and bitrate, VQ quickly becomes intractable at high dimensions or bitrates.

In QuIP#, we introduce the novel 2-bit 8 dimensional *E8P codebook*, which contains 2^{16} entries but only requires lookups into a 2^8-entry table, with the remaining 8 bits being used to store signs and shifts. E8P requires only 1KiB of space and therefore fits in the L1 cache of any modern GPU, even after duplicating for bank conflicts (32×). E8P

[2]It is straightforward to produce the g-block LDL decomposition from the Cholesky decomposition of H.

[3]The original QuIP paper also included multiple other technical guarantees, including a bound that considers more rigorously the "real" case of finite-sized codebooks. While these results could also be generalized to the block-LDLQ case, we view this as not providing much insight relevant to QuIP# beyond Theorem 4.1, so (if desired) they are left as an exercise for the reader.

mitigates the scaling issues of VQ by taking advantage of the structure and symmetries of the E_8 lattice on which it is based. The E_8 lattice is composed of all-integer or all-half-integer vectors in $\mathbb{R}^8$ whose sum is an even number, that is

$$E_8 = \left(\mathbb{Z}^8 \cup \left(\mathbb{Z}^8 + \tfrac{1}{2}\right)\right) \cap \left\{x \mid \mathbf{1}^T x \text{ is even}\right\}.$$

The construction of the E8P codebook starts with an equivalent way to write E_8 via the $\hat{D}_8$ lattice, where $\hat{D}_8 = \left\{x \in \mathbb{Z}^8 + \frac{1}{2} \mid \mathbf{1}^T x \text{ is even}\right\}$ is the set of half-integer vectors with even parity: here, $E_8 = \hat{D}_8 \cup (\hat{D}_8 + \frac{1}{2})$. It follows that $(\hat{D}_8 - \frac{1}{4}) \cup (\hat{D}_8 + \frac{1}{4}) = E_8 + \frac{1}{4}$ is just a shifted copy of E_8 (keeping the same optimal packing density).

$\hat{D}_8$ has nice symmetry properties: flipping any (nonzero) even number of signs of an element in $\hat{D}_8$, yields another distinct element in $\hat{D}_8$. This means that if $|\hat{D}_8|$ denotes the set of elementwise absolute values of entries in $\hat{D}_8$, then each element of $\hat{D}_8$ can be expressed (uniquely) as the elementwise product of an entry $s \in |\hat{D}_8|$ and a sign vector of appropriate parity. So, if we start from some "source codebook" of absolute entries $S \subset |\hat{D}_8|$, we can use the 128 possible odd- or even-parity sign flips to generate a subset of $\hat{D}_8$. Each entry in S is either an odd or even number of flips away from an entry in $\hat{D}_8$, but not both. Thus, given $s \in S$ and 7 out of the 8 sign flips, we can infer the last one from the parity of the 7 sign flips and s. This lets us use the following pattern to store a 16-bit codeword in $E_8 + \frac{1}{4}$: 8 bits for the entry in S, 7 bits for sign flips, and 1 bit to $\pm\frac{1}{4}$. This lets us decode a size 2^{16} codebook by looking up into only a size 2^8 codebook (S) and performing some operations. All that remains is how to choose S: we set S to be the 227 elements of $|\hat{D}_8|$ with norm $\leq \sqrt{10}$ plus 29 "padding" elements from $|\hat{D}_8|$ with norm $\sqrt{12}$ (see Section C.1). We call this ball-shaped 2^{16}-entry lattice codebook "E8P."

Figure 3 plots the elementwise MSE of quantizing a standard multivariate Gaussian to various k bit codebooks. Each k-bit codebook consists of a d-dimensional base lattice intersected with a ball to reach 2^{kd} points. The E_8-based codebooks achieve lower MSEs than all other presented codebooks, including those based on the D_4 lattice (the even-parity vectors in $\mathbb{Z}^4$), which achieves the kissing number in $\mathbb{R}^4$. This figure illustrates the importance of dimension for vector quantization. Increasing the vector dimension decreases the error for the half integer grid, as the resulting codebook is closer in shape to the source distribution. Finally, while K-means on the source distribution would achieve lower MSE (Lloyd, 1982), there are a number of practical reasons why a K-means based codebook would be less practical, including worse end-to-end empirical performance. We discuss this more in Section C.3.

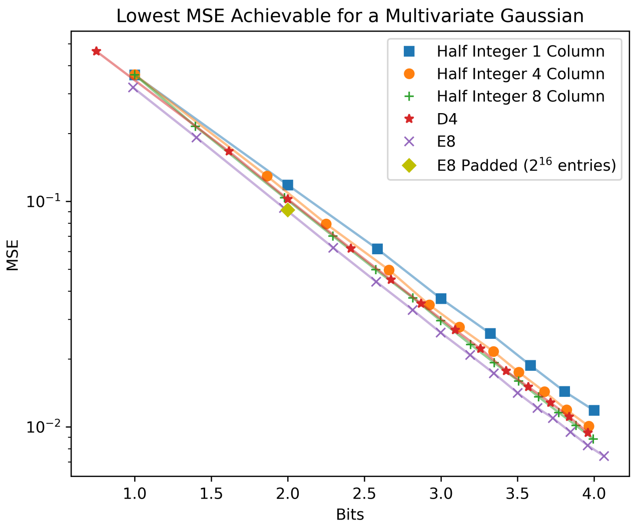

Figure 3. Minimum achievable elementwise MSE of quantizing a Gaussian to various codebooks. E_8-based codebooks outperform other presented codebooks due to the underlying packing density and high dimensionality of E_8.

4.3. Scaling E_8 to Higher Bitrates

The E_8 lattice works well for low bitrates (e.g. 2 bits), but quickly becomes intractable at higher bitrates due to codebook size. In QuIP#, we use residual vector quantization (RVQ) (Juang & Gray, 1982) to get the benefits of lattice codebooks at higher bitrates. RVQ quantizes a vector x to p bits with a set q of q_i-bit codebooks (denoted RVQ(x, p, q) where $p = \sum_{0 \leq i < |q|} q_i$) by repeatedly quantizing the quantization residual. That is, RVQ$(x, p, q) = \sum_{0 \leq i < |q|} \delta_i$ where $\delta_i = Q_{q_i}\left(\left(x - \sum_{0 \leq j < i} \delta_j\right)/s_i\right) \cdot s_i$, we let $Q_{q_i}(\cdot)$ denote quantizing to a q_i bit codebook, and $s_i \in \mathbb{R}$. Using RVQ, we can quantize to 4 bits by rounding with the 2 bit E8P codebook twice. We can also quantize to 3 bits by using the 2 bit E8P codebook and a 1-bit E_8 codebook (elements of E_8 with norm ≤ 2 and 15 elements of E_8 with norm 4). One could also use more advanced multi-codebook quantization approaches other than RVQ, but we found that RVQ was sufficient to achieve strong quantization performance.

5. Fine-Tuning During Quantization

Recent works have suggested that inter-layer interactions are important for lossless extreme quantization (Shao et al., 2024; Egiazarian et al., 2024). Here, we employ a simple fine-tuning algorithm that attempts to recover the original unquantized model during quantization. Our fine tuning method runs on a small development set and can be performed in around 50 GPU-hours for a 70B parameter model.

First, we fine-tune within each transformer block by fine-tuning unquantized layers to compensate for *already-quantized* layers before quantization. This mitigates the activation error caused by an individual linear layer *during quan-*

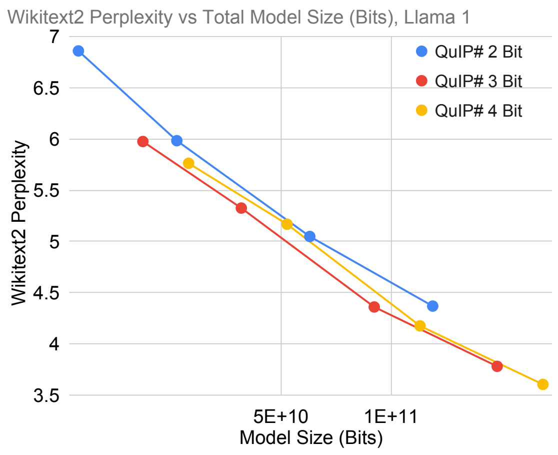

Figure 4. QuIP# scaling, Llama 1. Like Llama 2, QuIP# 3 bit scales better than QuIP# 4 bit for Llama 1 models and QuIP# 2 bit scales similarly to higher bitrates.

tization, and can be parallelized across transformer blocks. The idea of fine-tuning within a transformer block was previously proposed in Egiazarian et al. (2024); our methodology differs in how we fine tune (before quantization) and the set of tunable parameters. Second, after all linear layers in the model are quantized, the remaining unquantized parameters are fine-tuned to minimize activation error over the *entire model*. By optimizing the sign vectors as real vectors instead of binary vectors in both steps, we allow the incoherence processing step to shape the weight matrix *to the codebook*. While this means we must store the sign vectors in FP16 instead of as bitvectors, the size of LLM matrices means that the sign vectors still add less than 0.01 bits per weight. We describe these steps in more detail in Section D.

6. Experiments

Our main experiments show the performance of QuIP# on the Llama 1 (Touvron et al., 2023a) and 2 (Touvron et al., 2023b) family of models. These models range in size from 7 billion to 70 billion parameters and offer good performance, making them suitable for understanding how quantization methods perform and scale. Additional results for other models are available in the Appendix.

In Section 6.1, we compare QuIP# with recently published weight-only PTQ methods. AWQ scales weights by activation magnitudes *before* quantizing to reduce outliers (Lin et al., 2023). OmniQuant learns model-preserving layerwise transformations that reduce outliers per transformer block (Shao et al., 2024). AQLM uses vector quantization with learnable unstructured 8D codebooks (Egiazarian et al., 2024)[4]. We report AQLM's "1×16" numbers, which amounts to using a single codebook with 2^{16} entries $\in \mathbb{R}^8$ *per linear layer*. These codebooks each take up 1MiB of space, making them too large to fit in the L1 cache of any current GPU and thus preventing fast inference (see Table 6). Finally, we include QuIP (Chee et al., 2023) as a baseline for the improvements in QuIP#.

We report WxA16 numbers for AWQ and OmniQuant from the OmniQuant paper and AQLM numbers from AQLM. We note that there are currently 2 methods for evaluating perplexity: using the Llama 1 context length of 2048 or using the model's native context length (e.g. 4096 for Llama 2). OmniQuant and AWQ use 2048 for Llama 2 while AQLM uses 4096; we report both sets of numbers. We also note that AQLM paper reports QuIP# numbers from an outdated version of QuIP#; the numbers here represent the latest QuIP# numbers. Finally, we **bold** numbers in our tables when they are clearly better, such as a smaller model matching or outperforming a larger model or a similar sized model significantly outperforming another model.

6.1. QuIP# on Llama Models

Table 2 shows a comparison of QuIP# with OmniQuant, AWQ, and QuIP# without fine tuning and E8P, with context length 2048. QuIP# offers a paradigm shift in quantization quality over OmniQuant and AWQ. Notably, while AWQ falls apart at even 2.15 bits (Shao et al., 2024) and OmniQuant produces unusable models at 2 bits, QuIP# produces high quality models that are close to OmniQuant 3 bit models. Table 2 also shows the importance of incoherence processing. QuIP# without fine-tuning or lattice codebooks significantly outperforms OmniQuant and AWQ, which both rely on heuristics to reduce model outliers during quantization.

Table 4 shows a comparison of QuIP# with AQLM with context length 4096. At 2 and 3 bits, QuIP# either significantly outperforms similar-sized AQLM models or achieves similar performance with a smaller model[5]. At 4 bits, both methods perform similarly. This is not surprising as state-of-the-art 4 bit models are all very close to FP16 performance. Furthermore, the QuIP# 3 and 4 bit results presented in this paper use residual vector quantization; one could potentially achieve better numbers with more advanced multi-codebook quantization approaches.

Table 3 shows zeroshot results for QuIP#, AQLM, and OmniQuant. Both AQLM and QuIP# signficantly outperform OmniQuant, which correlates with the perpelxity results. AQLM and QuIP# both perform very close to FP16 at higher bitrates and for larger models, but QuIP# tends to outperform AQLM at lower bitrates and model sizes. We note that zeroshot tasks have an element of randomness and even FP16 numbers can disagree by up to 0.5%.

[4]We report results from the Jan 11, 2024 ArXiv version.

[5]In our experience, at extreme quantization levels, even 0.1 bits can make a significant difference in quantization quality.

Table 2. Llama 1 & 2 Wikitext2 and C4 perplexity (↓), context length 2048.

		Wikitext 2							C4						
Method	Bits	1-7	1-13	1-30	1-65	2-7	2-13	2-70	1-7	1-13	1-30	1-65	2-7	2-13	2-70
FP16	16	5.68	5.09	4.10	3.53	5.47	4.88	3.32	7.08	6.61	5.98	5.62	6.97	6.47	5.52
AWQ	4	6.08	5.34	4.39	3.76	6.15	5.12	-	7.52	6.86	6.17	5.77	7.68	6.74	-
OmniQ	4	5.86	5.21	4.25	3.71	5.74	5.02	3.47	7.34	6.76	6.11	5.73	7.35	6.65	5.65
QuIP# no FT & no E_8	4	5.83	5.20	4.23	3.63	5.66	5.00	3.42	7.25	6.70	6.06	5.68	7.17	6.59	5.59
QuIP#	4	**5.76**	**5.17**	**4.18**	**3.60**	**5.56**	**4.95**	**3.38**	**7.18**	**6.67**	**6.03**	**5.66**	**7.07**	**6.54**	**5.56**
AWQ	3	11.9	7.45	10.0	5.21	24.0	10.5	-	13.3	9.13	12.7	7.11	23.9	13.1	-
OmniQ	3	6.49	5.68	4.74	4.04	6.58	5.58	3.92	8.19	7.32	6.57	6.07	8.65	7.44	6.06
QuIP# no FT & no E_8	3	6.29	5.52	4.54	3.91	6.19	5.34	3.71	7.82	6.98	6.29	5.86	7.85	6.98	5.78
QuIP#	3	**5.98**	**5.31**	**4.36**	**3.78**	**5.79**	**5.10**	**3.56**	**7.39**	**6.83**	**6.17**	**5.77**	**7.32**	**6.72**	**5.67**
OmniQ	2	15.5	13.2	8.71	7.58	37.4	17.2	7.81	24.9	18.3	13.9	10.8	90.6	26.8	12.3
QuIP# no FT & no E_8	2	9.95	7.18	5.80	5.02	12.3	7.60	4.87	11.7	8.67	7.55	6.83	14.8	9.57	6.82
QuIP#	2	**6.86**	**5.97**	**5.02**	**4.36**	**6.66**	**5.74**	**4.16**	**8.36**	**7.48**	**6.71**	**6.19**	**8.35**	**7.45**	**6.12**

Table 3. Zeroshot Accuracy (`acc` in LM Eval, not `acc_norm`), Llama 2.

	2-70					2-13					2-7				
Method	Bits	ArcC	ArcE	PiQA	Wino	Bits	ArcC	ArcE	PiQA	Wino	Bits	ArcC	ArcE	PiQA	Wino
FP16	16	51.1	77.7	81.1	77.0	16	45.6	73.3	73.5	69.6	16	40.0	69.3	78.5	67.3
OmniQ	4	49.8	77.9	80.7	75.8	4	43.1	70.2	78.4	67.8	4	37.9	67.8	77.1	67.0
QuIP	4	47.0	74.3	80.3	76.0	4	44.9	73.3	**79.0**	69.7	4	-	-	-	-
AQLM	4.07	51.0	78.1	81.4	76.9	3.94	43.9	72.2	78.6	70.4	4.04	40.3	68.9	77.7	67.3
QuIP#	4	50.6	**78.1**	**81.4**	**77.1**	4	**45.5**	**73.9**	78.9	69.9	4	**40.5**	**69.1**	**78.4**	**67.6**
OmniQ	3	47.6	75.7	79.7	73.5	3	42.0	69.0	77.7	65.9	3	35.3	62.6	73.6	63.6
QuIP	3	46.3	73.2	80.0	74.6	3	41.5	70.4	76.9	**69.9**	3	-	-	-	-
AQLM	3.01	50.0	77.6	81.3	**77.2**	3.03	43.6	**73.5**	77.8	67.6	3.04	38.7	67.8	76.6	**68.4**
QuIP#	3	**50.9**	**77.7**	**81.4**	76.4	3	**44.0**	72.5	**78.4**	69.1	3	**39.2**	**68.4**	**77.3**	66.5
OmniQ	2	28.7	55.4	68.8	53.2	2	23.0	44.4	62.6	52.6	2	21.6	35.2	57.5	51.5
QuIP	2	34.0	62.2	74.8	67.5	2	23.5	45.2	62.0	52.8	2	19.4	26.0	54.6	51.8
AQLM	2.07	47.9	77.7	80.4	75.9	1.97	38.5	67.0	75.1	**69.5**	2.02	33.6	62.8	73.5	64.6
QuIP#	2	**48.7**	77.3	80.3	**75.9**	2	**39.5**	**69.3**	**77.3**	67.7	2	**34.6**	**64.6**	**75.1**	**64.9**

6.2. QuIP# Bit Scaling

Figures 1 (first page) and 4 show how QuIP# scales on the Llama family of models and Wikitext2. On both Llama 1 and 2, QuIP# 3 bit outperforms QuIP# 4 bit and QuIP# 2 bit offers similar scaling to 3 and 4 bit models. Furthermore, on Llama 2, QuIP# 3 bit outperforms a theoretical lossless 4 bit model (FP16 at 4 bits). To the best of our knowledge, this is the first time a 3 bit PTQ method has outperformed a theoretical lossless 4 bit model and also the first time a 2 bit PTQ method has offered similar scaling to higher bitrates.

6.3. Efficient Inference with QuIP#

One of the key benefits of PTQ is to increase the maximum possible inference throughput on a given device. Since small-batch autoregressive decoding is usually memory bound, a smaller model requires less data to be read and can therefore be served faster. However, achieving an actual speedup requires a quantization method with low decoding overhead, or inference will be bottlenecked by decoding. For example, the AQLM models in the experiment tables use a different $2^{16} \times 8$ codebook for every linear layer. Each entry in these codebooks takes 2 bytes, meaning that each codebook is 1MiB. During inference, weights are read from these codebook in an essentialy random access pattern, meaning that the entire codebook must fit in L1 cache to enable fast inference (even L2 cache is too slow). However, 1MiB is larger than any current GPU's L1 cache (the H100 has 256KB), so AQLM inference suffers from high cache miss rates and is actually *slower than FP16* on modern GPUs (Table 6).

In contrast, QuIP# was designed around fast inference. The RHT can be computed in essentially $O(n \log n)$ time and E8P only requires 1KiB and can be decoded from with very few (< 5) instructions per weight. Table 5 shows QuIP#'s generation speed as measured with the FlashAttention library's (Dao et al., 2022; Dao, 2023) implementation of Llama. QuIP# is able to achieve over 50% of peak memory bandwidth with a 2 bit model even with minimal kernel fusion in the RHT, validating our design choices. We note that since these "fast inference design choices" essentially amount to restrictions on what can be done during quantization, it should be entirely possible to achieve even better quantization quality at the expense of inference speed.

Table 4. Wikitext2 and C4 perplexity (↓), context length 4096.

	2-7			2-13			2-70		
METHOD	BITS	W2	C4	BITS	W2	C4	BITS	W2	C4
FP16	16	5.12	6.63	16	4.57	6.05	16	3.12	4.97
QUIP#	4	**5.19**	6.75	4	4.63	6.13	4	3.18	5.02
↳ NO FT	4	5.22	6.79	4	4.65	6.15	4	3.18	5.02
↳ NO E_8	4	5.29	6.86	4	4.68	6.20	4	3.22	5.05
QUIP	4	-	-	4	4.76	6.29	4	3.58	5.38
AQLM	4.04	5.21	6.74	3.94	4.64	6.14	4.07	3.17	5.01
QUIP#	3	**5.41**	**7.04**	3	**4.78**	**6.35**	3	**3.35**	**5.15**
↳ NO FT	3	5.60	7.34	3	4.90	6.50	3	3.41	5.20
↳ NO E_8	3	5.77	7.61	3	4.99	6.65	3	3.48	5.28
QUIP	3	-	-	3	5.12	6.79	3	3.87	5.67
AQLM	3.04	5.46	7.10	3.03	4.83	6.37	3.01	3.36	5.17
QUIP#	2	**6.19**	**8.16**	2	**5.35**	**7.20**	2	**3.91**	**5.71**
↳ NO FT	2	8.22	11.0	2	6.06	8.07	2	4.16	6.01
↳ NO E_8	2	11.2	14.5	2	7.04	9.37	2	4.58	6.51
QUIP	2	-	-	2	13.5	16.2	2	5.90	8.17
AQLM	2.02	6.93	8.84	1.97	5.70	7.59	2.07	3.94	5.72

Table 5. QuIP# generation throughput on a NVIDIA RTX 4090 using the *FlashAttention* library's Llama implementation. QuIP# achieves $> 50\%$ peak memory bandwidth (1TB/s) during generation and admits fast inference.

MODEL	2 BIT TOK/S	2 BIT % MEM BW	4 BIT TOK/S	4 BIT % MEM BW
2-7B	170.50	29.60%	117.73	40.87%
2-13B	104.83	33.80%	71.09	45.84%
1-30B	51.60	38.39%	32.50	48.36%
2-70B	32.74	56.84%	OOM	OOM

Finally, if we look at the speed-quality tradeoffs of different quantization methods, we also find that QuIP# enables new frontiers of PTQ performance. Compared to QuIP's published throughput numbers (measured on an A6000), QuIP# on an A6000 achieves roughly twice the inference throughput at the same bitrate, making QuIP# strictly better. Compared to existing "fast inference" quantization methods such as SpQR (Dettmers et al., 2023) and SqueezeLLM (Kim et al., 2023), we again find that QuIP# offers significantly higher throughput ($> 40\%$) at the same or better quantization quality.

Table 6. QuIP# vs AQLM and FP16 generation throughput on a NVIDIA RTX 4090 using the *HuggingFace* library's Llama implementation. Unlike AQLM, whose codebook is too large to fit in L1 cache, QuIP# achieves significant speedups over FP16.

METHOD	2-7B	2-70B
FP16	33.1 TOK/S	OOM
AQLM 2 BIT	20.6	8.27
QUIP# 2 BIT	106.3	25.9

6.4. Ablations

Table 4 also contains an ablation on the various components of QuIP#. The "no FT" row shows QuIP# without fine-tuning and the "no E_8" row shows QuIP# without fine-tuning and lattice codebooks. For the latter, we round to the 1-dimensional half-integer grid. We also include QuIP numbers as reported by AQLM. At all bitrates, each component of QuIP# brings additional performance gains. The difference between QuIP and QuIP# without fine-tuning and lattice codebooks also shows the difference between QuIP's Kronecker factorization and QuIP#'s RHT. The RHT offers stronger incoherence properties than the Kronecker factorization (Section 3), which improves performance.

7. Conclusion

We present QuIP#, a weight-only post training compression method that achieves state-of-the-art results on LLMs at 2, 3, and 4 bits per weight. QuIP# uses the Randomized Hadamard Transform as an efficient and principled form of outlier suppression, and introduces the E_8 lattice-based E8P codebook to better quantize RHT transformed weights. The E8P codebook is highly symmetric and admits fast inference, allowing a "proof of concept" QuIP# CUDA implementation to achieve over 50% peak memory bandwidth on modern GPUs. QuIP# also implements inter-layer fine tuning, further improving quantization. To the best of our knowledge, QuIP# is the first PTQ method to achieve superior scaling at 3 bits over 4 bits and similar scaling at 2 bits to higher bitrates. Our results indicate that, in the near future, 2 bit models are likely to scale better than 3 bit ones.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

Acknowledgements

We thank, in no particular order, David Hou for helping with the QuIP# CUDA implementation, Tiancheng Yuan for lending his RTX 4090 and helping with acquiring QuIP# timing numbers, Tri Dao for a fast CUDA implementation of the Hadamard transform and general help with QuIP#, and Together AI for compute resources.

References

Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Étienne Goffinet, Hesslow, D., Launay, J., Malartic, Q., Mazzotta, D., Noune, B., Pannier, B., and Penedo, G. The falcon series of open language models, 2023.

Chee, J., Renz, M., Damle, A., and Sa, C. D. Model preserving compression for neural networks. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=gt-l9Hu2ndd.

Chee, J., Cai, Y., Kuleshov, V., and Sa, C. D. QuIP: 2-bit quantization of large language models with guarantees. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=xrk9g5vcXR.

Cochran, W., Cooley, J., Favin, D., Helms, H., Kaenel, R., Lang, W., Maling, G., Nelson, D., Rader, C., and Welch, P. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967. doi: 10.1109/PROC.1967.5957.

Computer, T. Redpajama: An open source recipe to reproduce llama training dataset, 2023. URL https://github.com/togethercomputer/RedPajama-Data.

Dao, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Dettmers, T. and Zettlemoyer, L. The case for 4-bit precision: k-bit inference scaling laws. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 7750–7774. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/dettmers23a.html.

Dettmers, T., Svirschevski, R., Egiazarian, V., Kuznedelev, D., Frantar, E., Ashkboos, S., Borzunov, A., Hoefler, T., and Alistarh, D. Spqr: A sparse-quantized representation for near-lossless llm weight compression, 2023.

Egiazarian, V., Panferov, A., Kuznedelev, D., Frantar, E., Babenko, A., and Alistarh, D. Extreme compression of large language models via additive quantization, 2024.

Fino and Algazi. Unified matrix treatment of the fast walsh-hadamard transform. *IEEE Transactions on Computers*, C-25(11):1142–1146, 1976. doi: 10.1109/TC.1976.1674569.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

Gao, L., Tow, J., Abbasi, B., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., Le Noac'h, A., Li, H., McDonell, K., Muennighoff, N., Ociepa, C., Phang, J., Reynolds, L., Schoelkopf, H., Skowron, A., Sutawika, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. A framework for few-shot language model evaluation, 12 2023. URL https://zenodo.org/records/10256836.

Halko, N., Martinsson, P.-G., and Tropp, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

Hedayat, A. and Wallis, W. D. Hadamard Matrices and Their Applications. *The Annals of Statistics*, 6(6):1184 – 1238, 1978. doi: 10.1214/aos/1176344370. URL https://doi.org/10.1214/aos/1176344370.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts, 2024.

Juang, B.-H. and Gray, A. Multiple stage vector quantization for speech coding. In *ICASSP '82. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 7, pp. 597–600, 1982. doi: 10.1109/ICASSP.1982.1171604.

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M., and Keutzer, K. Squeezellm: Dense-and-sparse quantization. *arXiv*, 2023.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Kostina, V. and Verdú, S. Fixed-length lossy compression in the finite blocklength regime: Gaussian source. *2011 IEEE Information Theory Workshop, ITW 2011*, 10 2011. doi: 10.1109/ITW.2011.6089501.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration, 2023.

Liu, Z., Oguz, B., Zhao, C., Chang, E., Stock, P., Mehdad, Y., Shi, Y., Krishnamoorthi, R., and Chandra, V. Llm-qat: Data-free quantization aware training for large language models, 2023.

Lloyd, S. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. doi: 10.1109/TIT.1982.1056489.

Nagel, M., Amjad, R. A., Van Baalen, M., Louizos, C., and Blankevoort, T. Up or down? Adaptive rounding for post-training quantization. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7197–7206. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/nagel20a.html.

Nagel, M., Fournarakis, M., Bondarenko, Y., and Blankevoort, T. Overcoming oscillations in quantization-aware training. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 16318–16330. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/nagel22a.html.

Nguyen, E., Poli, M., Faizi, M., Thomas, A., Birch-Sykes, C., Wornow, M., Patel, A., Rabideau, C., Massaroli, S., Bengio, Y., Ermon, S., Baccus, S. A., and Ré, C. Hyenadna: Long-range genomic sequence modeling at single nucleotide resolution. 2023.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2024.

Shao, W., Chen, M., Zhang, Z., Xu, P., Zhao, L., Li, Z., Zhang, K., Gao, P., Qiao, Y., and Luo, P. Omniquant: Omnidirectionally calibrated quantization for large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=8Wuvhh0LYW.

Sloane, N. Hadamard Matrices — neilsloane.com. http://neilsloane.com/hadamard/. [Accessed 02-02-2024].

Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023. URL https://openreview.net/forum?id=tz9JV2PRSv.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023b.

Viazovska, M. The sphere packing problem in dimension 8. *Annals of Mathematics*, 185(3), May 2017. ISSN 0003-486X. doi: 10.4007/annals.2017.185.3.7. URL http://dx.doi.org/10.4007/annals.2017.185.3.7.

Xi, H., Li, C., Chen, J., and Zhu, J. Training transformers with 4-bit integers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=H9hWlfMT6O.

A. Concentration Inequalities for the Randomized Hadamard Transform and Fast Fourier Transform

A.1. Incoherence Processing with the Randomized Hadamard Transform

Lemma A.1. *For any non-negative real number n,*

$$\frac{1}{B\left(n,1/2\right)} \int_{-1}^{+1} (1-x^2)^{n-1} \cdot \exp(tx)\, dx \le \exp\left(\frac{t^2}{4n+2}\right).$$

Proof. We start with the following "standard" integral. For non-negative integer m and real $n > 0$,

$$\int_{-1}^{+1} x^{2m}(1-x^2)^{n-1}\, dx = B\left(m+\frac{1}{2}, n\right) = \frac{\Gamma\left(m+\frac{1}{2}\right)\Gamma\left(n\right)}{\Gamma\left(m+n+\frac{1}{2}\right)}.$$

This means that

$$\begin{aligned}\frac{1}{B\left(\frac{1}{2},n\right)} \int_{-1}^{+1} x^{2m}(1-x^2)^{n-1}\, dx &= \frac{B\left(m+\frac{1}{2},n\right)}{B\left(\frac{1}{2},n\right)} \\ &= \frac{\Gamma\left(m+\frac{1}{2}\right)\Gamma\left(n\right)}{\Gamma\left(m+n+\frac{1}{2}\right)} \cdot \frac{\Gamma\left(n+\frac{1}{2}\right)}{\Gamma\left(\frac{1}{2}\right)\Gamma\left(n\right)} \\ &= \frac{\Gamma\left(m+\frac{1}{2}\right)\Gamma\left(n+\frac{1}{2}\right)}{\sqrt{\pi}\cdot\Gamma\left(m+n+\frac{1}{2}\right)}.\end{aligned}$$

Applying the Legendre duplication formula, for integer m,

$$\Gamma\left(m+\frac{1}{2}\right) = \frac{(2m)!\,\sqrt{\pi}}{4^m m!},$$

then

$$\begin{aligned}\frac{1}{B\left(\frac{1}{2},n\right)} \int_{-1}^{+1} x^{2m}(1-x^2)^{n-1}\, dx &= \frac{(2m)!\,\sqrt{\pi}}{4^m m!} \cdot \frac{(2n)!\,\sqrt{\pi}}{4^n n!} \cdot \frac{1}{\sqrt{\pi}} \cdot \frac{4^{m+n}(m+n)!}{(2m+2n)!\,\sqrt{\pi}} \\ &= \frac{(2m)!\,(2n)!\,(m+n)!}{m!\,n!\,(2m+2n)!}.\end{aligned}$$

In particular, this means that

$$\begin{aligned}
\frac{1}{B\left(\frac{1}{2},n\right)}\int_{-1}^{+1}\exp(tx)(1-x^2)^{n-1}\,dx &= \sum_{m=0}^{\infty}\frac{t^{2m}}{(2m)!}\cdot\frac{1}{B\left(\frac{1}{2},n\right)}\int_{-1}^{+1}x^{2m}(1-x^2)^{n-1}\,dx \\
&= \sum_{m=0}^{\infty}\frac{t^{2m}}{(2m)!}\cdot\frac{(2m)!\,(2n)!\,(m+n)!}{m!\,n!\,(2m+2n)!} \\
&= \sum_{m=0}^{\infty}\frac{t^{2m}}{m!}\cdot\frac{(2n)!\,(m+n)!}{n!\,(2m+2n)!} \\
&= \sum_{m=0}^{\infty}\frac{t^{2m}}{m!}\cdot\prod_{k=1}^{m}\frac{k+n}{(2k+2n)(2k+2n-1)} \\
&= \sum_{m=0}^{\infty}\frac{t^{2m}}{m!}\cdot\frac{1}{2^m}\prod_{k=1}^{m}\frac{1}{2k+2n-1} \\
&\leq \sum_{m=0}^{\infty}\frac{t^{2m}}{m!}\cdot\frac{1}{2^m}\left(\frac{1}{2n+1}\right)^m \\
&= \sum_{m=0}^{\infty}\frac{1}{m!}\left(\frac{t^2}{4n+2}\right)^m \\
&= \exp\left(\frac{t^2}{4n+2}\right).
\end{aligned}$$

This proves the lemma

$$\frac{1}{B\left(n,1/2\right)}\int_{-1}^{+1}(1-x^2)^{n-1}\cdot\exp(tx)\,dx \leq \exp\left(\frac{t^2}{4n+2}\right).$$

□

Lemma A.2. *Call $U \in \mathbb{R}^{nd\times nd}$ an (n,d)-block orthohadamard matrix if it has the following properties: (1) U is a orthogonal matrix, and (2) each aligned $d\times d$ block of U is $1/\sqrt{n}$ times an orthogonal matrix. This generalizes the notion of Hadamard matrices. Let $S \in \mathbb{R}^{nd\times nd}$ be a random block diagonal matrix, where each $d\times d$ block of the diagonal is sampled independently and uniformly from the set of (possibly special) orthogonal matrices. Then we call multiplication by US a* randomized orthohadamard transform*, and observe that it has the following nice property. Let $x \in \mathbb{R}^{nd}$ be any fixed vector, and let $b \in \mathbb{R}^{nd}$ be a fixed vector that is* sparse *in the sense that it is supported only on a single d-sized aligned block (i.e. all but one of the n blocks are zero). Then*

$$\mathbf{P}\left(\left|b^TUSx\right|\geq a\right)\leq 2\exp\left(-\frac{a^2nd}{2\left\|b\right\|^2\left\|x\right\|^2}\right).$$

Proof. If we let the ith block of x be $x_i \in \mathbb{R}^d$ and let the ith block of $S^TU^Tb^T$ be v_i, then the v_i will be independent and uniformly distributed on the sphere in d dimensional space of radius $\|b\|/\sqrt{n}$, and so $v_i^Tx_i = \|b\|\,\|x_i\|\,n^{-1/2}z_i$, where the z_i are all independent and distributed according to an entry of a random point on the unit sphere in d dimensional space. Observe that this means that

$$\mathbf{P}(z_i) \propto (1-z_i^2)^{\frac{d-1}{2}-1}.$$

So,

$$\begin{aligned}
\mathbf{E}\left[\exp\left(tb^TUSx\right)\right] &= \mathbf{E}\left[\exp\left(t\sum_{i=1}^{n}\|b\|\,\|x_i\|\,n^{-1/2}z_i\right)\right]\\
&= \prod_{i=1}^{n}\mathbf{E}\left[\exp\left(t\,\|b\|\,\|x_i\|\,n^{-1/2}z_i\right)\right]\\
&\le \prod_{i=1}^{n}\mathbf{E}\left[\exp\left(\frac{1}{4\cdot\frac{d-1}{2}+2}\right)\left(t\,\|b\|\,\|x_i\|\,n^{-1/2}\right)^2\right]\\
&= \prod_{i=1}^{n}\mathbf{E}\left[\exp\left(\frac{t^2\,\|b\|^2\,\|x_i\|^2}{2nd}\right)\right]\\
&= \mathbf{E}\left[\exp\left(\frac{t^2\,\|b\|^2\,\|x\|^2}{2nd}\right)\right],
\end{aligned}$$

where the the last line follows from Lemma A.1. It follows from the standard application of Markov's inequality that for any $a > 0$,

$$\mathbf{P}\left(\left|b^TUSx\right| \ge a\right) \le 2\exp\left(-\frac{a^2nd}{2\,\|b\|^2\,\|x\|^2}\right).$$

This is what we wanted to show. ☐

Lemma A.3. *Let $H \in \mathbb{R}^{n\times n}$ be an orthogonal scaled Hadamard matrix or $F \in \mathbb{R}^{n\times n}$ be an orthogonal FFT matrix (the FFT understood as operating on a real vector space). Let $S \in \mathbb{R}^{n\times n}$ be a random diagonal matrix with diagonal elements supported on $\mathbb{R}^n$, and let $P \in \mathbb{R}^{n\times n}$ be a random 2-block-diagonal matrix with 2×2 diagonal blocks supported on $\mathrm{SO}(2)$ (we can also think of this as acting like a diagonal complex matrix with each diagonal element a random complex number of absolute value 1). Let $U \in \mathbb{R}^{n\times n}$ be any fixed orthogonal matrix. Then, for any $\epsilon > 0$,*

$$\mathrm{Prob}\left(\max_{i,j}\left|e_i^THSUe_j\right| \ge \sqrt{\frac{2}{nd}\log\left(\frac{2n^2}{\epsilon}\right)}\right) \le \epsilon$$

and

$$\mathrm{Prob}\left(\max_{i,j}\left|e_i^TFPUe_j\right| \ge \sqrt{\frac{2}{nd}\log\left(\frac{2n^2}{\epsilon}\right)}\right) \le \epsilon.$$

That is, with probability at least $1-\epsilon$, multiplying by either HS or FP makes the resulting orthogonal matrix μ-incoherent, where

$$\mu_H = \sqrt{2\log\left(\frac{2n^2}{\epsilon}\right)}.$$

Proof. Setting $b = e_i$ and $x = Ue_j$ in Lemma A.2,

$$\mathbf{P}\left(\left|e_i^THSUe_j\right| \ge a\right) \le 2\exp\left(-\frac{a^2nd}{2}\right).$$

By the union bound,

$$\mathbf{P}\left(\max_{i,j}\left|e_i^THSUe_j\right| \ge a\right) \le 2n^2\exp\left(-\frac{a^2nd}{2}\right).$$

Setting

$$a^2 = \frac{2}{nd}\log\left(\frac{2n^2}{\epsilon}\right)$$

proves the lemma. The FFT case is identical. ☐

Algorithm 4 Incoherence Processing with RFFT (IP-RFFT)

```
input $W \in \mathbb{R}^{m \times n}, H \in \mathbb{R}^{n \times n}$
  Sample phase vectors
      $\theta_V \sim \mathcal{U}[0, 2\pi]^{n/2}$, $\theta_U \sim \mathcal{U}[0, 2\pi]^{m/2}$
      $S_V = \cos(\theta_V) + i \sin(\theta_V)$, $S_U = \cos(\theta_U) + i \sin(\theta_U)$
  $\hat{W} \leftarrow \texttt{FFT}(diag(S_U)\texttt{FFT}(diag(S_V)W^T)^T)$ where
      FFT is the Fast Fourier transform (Sec. A.2)
  $\hat{H} \leftarrow \texttt{FFT}(diag(S_V)\texttt{FFT}(diag(S_V)H)^T)$
output $\hat{W}, \hat{H}, S_U, S_V$
```

Lemma A.4. *Let $H_L \in \mathbb{R}^{m \times m}$ be an orthogonal scaled Hadamard matrix or $F \in \mathbb{R}^{m \times m}$ be an orthogonal FFT matrix (the FFT understood as operating on a real vector space). Let $S_L \in \mathbb{R}^{m \times m}$ be a random diagonal matrix with diagonal elements supported on $\mathbb{R}^m$, and let $P \in \mathbb{R}^{m \times m}$ be a random 2-block-diagonal matrix with 2×2 diagonal blocks supported on* $\mathrm{SO}(2)$. *Let $H_R \in \mathbb{R}^{n \times n}$, F_R, S_R, and P_R be defined analogously over n-dimensional space. Let $W \in \mathbb{R}^{m \times n}$ be any fixed matrix. Then, for any $\epsilon > 0$,*

$$\mathbf{P}\left(\max_{i,j} \left|e_i^T H_L S_L W S_R^T H_R^T e_j\right| \geq \|W\|_F \sqrt{\frac{4}{mn}} \log\left(\frac{4mn}{\epsilon}\right)\right) \leq \epsilon.$$

and

$$\mathbf{P}\left(\max_{i,j} \left|e_i^T F_L P_L W P_R^T F_R^T e_j\right| \geq \|W\|_F \sqrt{\frac{4}{mn}} \log\left(\frac{4mn}{\epsilon}\right)\right) \leq \epsilon.$$

That is, with probability at least $1 - \epsilon$, multiplying on both sides by a randomized Hadamard transform or a randomized FFT yields a weight matrix that is μ_W-incoherent, where

$$\mu_W = 2 \log\left(\frac{4mn}{\epsilon}\right).$$

Proof. From Lemma A.2,

$$\mathbf{P}\left(\left|b^T U S x\right| \geq \|b\| \, \|x\| \sqrt{\frac{2}{n} \log\left(\frac{4mn}{\epsilon}\right)}\right) \leq \frac{\epsilon}{2mn}.$$

By applying this once on each side to the rows and columns respectively, and union bounding over the mn entries, we get

$$\mathbf{P}\left(\left|e_i^T H_L S_L W S_R^T H_R^T e_j\right| \geq \|W\|_F \sqrt{\frac{4}{mn}} \log\left(\frac{4mn}{\epsilon}\right)\right) \leq \epsilon.$$

The proof in the FFT case is identical. □

Lemma 3.1. *Let H be any positive semidefinite matrix on $\mathbb{R}^{n \times n}$ and W any weight matrix on $\mathbb{R}^{m \times n}$. Let $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ be orthogonal scaled Hadamard matrices. Let $S_U \in \mathbb{R}^{m \times m}$ and $S_V \in \mathbb{R}^{n \times n}$ be random diagonal matrices with independent diagonal elements drawn uniformly from $\{-1, +1\}$. Then for any $\delta > 0$, $V S_V H S_V V^T$ is μ_H-incoherent with probability at least $1 - \delta$, and $U S_U W S_V V^T$ is μ_W-incoherent with probability at least $1 - \delta$, where*

$$\mu_H = \sqrt{2 \log\left(\frac{2n^2}{\delta}\right)} \quad \textit{and} \quad \mu_W = 2 \log\left(\frac{4mn}{\delta}\right).$$

Proof. The incoherence of H follows from the application of Lemma A.3. The incoherence of W follows from the application of Lemma A.4. □

A.2. Incoherence Processing with the Randomized Fast Fourier Transform (RFFT)

Here we described the Randomized Fast Fourier Transform (RFFT), $x \to VSx$ where $V \in \mathbb{C}^{n/2 \times n/2}$ is the discrete Fourier transform matrix, $S \in \mathbb{C}^{n/2}$ is a random complex phase vector, and $x \in \mathbb{R}^n$. The discrete Fourier transform can be computed in $O(n \log n)$ time via the fast Fourier transform. Here it is understood that the FFT operates over the reals, in that a vector $x \in \mathbb{R}^n$ is mapped to a complex representation $\mathbb{C}^{n/2}$, the RFFT is performed, and the resulting vector mapped back to $\mathbb{R}^n$. Here the mapping simply represents reshaping real-valued x into dimension $(n/2, 2)$, and interpreting the corresponding 2-tuples as a complex number.

Incoherence processing via the RFFT achieves similar theoretical guarantees as the RHT, see Lemmas A.3 and A.4. Ultimately the choice of the orthogonal transformation is up to the user. A Fourier transform works almost as well as a Hamard transform in practice (Table 1), so if a fast Hadamard implementation is not available, the FFT is a good option.

B. Block LDLQ

Lemma B.1. *Let $H \in \mathbb{R}^{nd \times nd}$ be a positive definite matrix with d-block LDL decomposition $H = L^T DL$. Then*

$$\operatorname{tr}(D) \le \operatorname{tr}\left(H^{1/2}\right) \cdot \left\| H^{1/2} \odot M_D \right\|_2,$$

where $M_D = I \otimes \mathbf{1}_{d \times d}$ is the block diagonal mask. If, in addition, H is μ-incoherent in the sense that its matrix of eigenvectors U has

$$\|U_{ij}\| \le \frac{\mu}{\sqrt{nd}},$$

then

$$\operatorname{tr}(D) \le \frac{\mu^2}{n} \operatorname{tr}\left(H^{1/2}\right)^2.$$

Proof. Consider the optimization problem

$$\begin{array}{rl} \text{minimize:} & \operatorname{tr}\left(R^T H R\right) \\ \text{subject to:} & R \text{ unit block lower diagonal.} \end{array}$$

Observe that the derivative of the loss is

$$\nabla f(R) = HR.$$

If $R = L^{-1}$, then $HR = L^T D$. But this must be a block upper triangular matrix, because it's the product of a unit upper triangular matrix (L^T) and a block diagonal matrix D. It follows that $\nabla f(L^{-1})$ is zero in all the directions in which we could move R, since R only varies in the strictly lower triangular directions. Therefore, $R = L^{-1}$ is the solution to this optimization problem, and for any R, $\nabla f(R) \ge \nabla f(L^{-1}) = \operatorname{tr}(D)$.

Now, let M denote the strictly block lower triangular mask, and observe that $M + M^T + M_D = \mathbf{1}_{nd \times nd}$. Set $\alpha = \left\| H^{1/2} \odot M_D \right\|_2^{-1}$, and consider $R = \left(I + \alpha M \odot H^{1/2}\right)^{-1}$. Observe that

$$\begin{aligned} \left(I + \alpha M \odot H^{1/2}\right)^T \left(I + \alpha M \odot H^{1/2}\right) &= I + \alpha M \odot H^{1/2} + \alpha M^T \odot H^{1/2} + \alpha^2 (M^T \odot H^{1/2})(M \odot H^{1/2}) \\ &\succeq I + \alpha(M + M^T) \odot H^{1/2} \\ &\succeq \alpha M_D \odot H^{1/2} + \alpha(M + M^T) \odot H^{1/2} \\ &\succeq \alpha H^{1/2}. \end{aligned}$$

It follows by inverting both sides that $RR^T \preceq \alpha^{-1} H^{-1/2}$.

So, for this R,

$$\operatorname{tr}\left(R^T H R\right) = \operatorname{tr}\left(HRR^T\right) \le \alpha^{-1} \operatorname{tr}\left(H^{1/2}\right).$$

This proves the first part of the lemma. For the second part, observe that

$$\left\| H^{1/2} \odot M_D \right\|_2 \le \sum_{i=1}^{nd} \lambda_i^{1/2} \left\| (u_i u_i^T) \odot M_D \right\|_2$$
$$\le \frac{\mu^2}{n} \operatorname{tr}\left(H^{1/2}\right).$$

This proves the lemma. □

Theorem 4.1. *Suppose that we round $W \in \mathbb{R}^{m\times n}$ using g-block LDLQ with Hessian H, producing $\hat{W}$. Suppose that H is μ-incoherent, and that we use a (possibly stochastic) vector quantizer $\mathbf{Q}$ that satisfies $\mathbf{E}[(\mathbf{Q}(x)-x)(\mathbf{Q}(x)-x)^T] \preceq \sigma^2 I$ for any $x \in \mathbb{R}^g$. Then*

$$\mathbf{E}[\operatorname{tr}((\hat{W}-W)H(\hat{W}-W)^T)] \le \tfrac{gm\mu^2\sigma^2}{n} \operatorname{tr}(H^{1/2})^2.$$

Proof. First recall that from the description of block LDLQ,

$$\hat{W}_k = \mathbf{Q}(W_k + (W_{:(k-1)} - \hat{W}_{:(k-1)})\mathbf{A}_k).$$

We can also write this in matrix form in terms of the matrix $\mathbf{L}_k$ as

$$\hat{W} = \mathbf{Q}(W + (W - \hat{W})(\mathbf{L}^T - I)).$$

Here, $\mathbf{Q}$ is interpreted as operating independently block-wise. Let η denote the quantization error

$$\eta = (W + (W-\hat{W})(\mathbf{L}^T - I)) - \mathbf{Q}(W + (W-\hat{W})(\mathbf{L}^T - I)).$$

Then

$$\hat{W} = (W + (W-\hat{W})(\mathbf{L}^T - I)) - \eta,$$

which simplifies to

$$(W-\hat{W})\mathbf{L}^T = \eta.$$

This means that

$$\mathbf{E}[\operatorname{tr}((\hat{W}-W)H(\hat{W}-W)^T)] = \mathbf{E}[\operatorname{tr}((\hat{W}-W)\mathbf{L}^T\mathbf{D}\mathbf{L}(\hat{W}-W)^T)] = \mathbf{E}[\operatorname{tr}(\eta^T \mathbf{D}\eta)].$$

But by assumption, $\mathbf{E}[\eta\eta^T] \preceq m\sigma^2 I$ (since each block is just an independent application of $\mathbf{Q}$ and we sum over m rows), so

$$\mathbf{E}[\operatorname{tr}((\hat{W}-W)H(\hat{W}-W)^T)] \le m\sigma^2 \mathbf{E}[\operatorname{tr}(\mathbf{D})].$$

Combining this with the result of Lemma B.1 proves the theorem. □

C. E8P details

C.1. Constructing S

We use the following 29 elements of $\hat{D}_8$ with norm squared 12 to pad S to 256 entries.

```
([3, 1, 1, 1, 3, 3, 3, 3] [1, 3, 1, 1, 3, 3, 3, 3] [1, 1, 3, 1, 3, 3, 3, 3]
 [1, 1, 1, 3, 3, 3, 3, 3] [3, 3, 3, 1, 3, 3, 1, 1] [3, 3, 3, 1, 3, 1, 3, 1]
 [3, 3, 3, 1, 1, 3, 3, 1] [3, 3, 3, 1, 3, 1, 1, 3] [3, 3, 3, 1, 1, 3, 1, 3]
 [3, 3, 3, 1, 1, 1, 3, 3] [3, 3, 1, 3, 3, 3, 1, 1] [3, 3, 1, 3, 3, 1, 3, 1]
 [3, 3, 1, 3, 1, 3, 3, 1] [3, 3, 1, 3, 3, 1, 1, 3] [3, 3, 1, 3, 1, 3, 1, 3]
 [3, 3, 1, 3, 1, 1, 3, 3] [3, 1, 3, 3, 3, 3, 1, 1] [3, 1, 3, 3, 3, 1, 3, 1]
 [3, 1, 3, 3, 1, 3, 3, 1] [3, 1, 3, 3, 3, 1, 1, 3] [3, 1, 3, 3, 1, 3, 1, 3]
 [1, 3, 3, 3, 1, 1, 3, 3] [1, 3, 3, 3, 3, 3, 1, 1] [1, 3, 3, 3, 3, 1, 3, 1]
 [1, 3, 3, 3, 1, 3, 3, 1] [1, 3, 3, 3, 3, 1, 1, 3] [1, 3, 3, 3, 1, 3, 1, 3]
 [1, 1, 3, 3, 1, 3, 3, 3] [3, 3, 1, 1, 3, 3, 3, 1] ) / 2
```

C.2. Example Decoding with E8P

Here, we give an example of decoding with E8P. In this example, the first 8 bits of the codeword encode the entry in S, the next 7 bits encode the 7 right sign flips, and the last bit encodes whether or not we shift by $\frac{1}{4}$. Let the codeword be `0001010110010111`. The first 8 bits `00010101` = 21 would indicate that we start with the 21st entry in S. In this example, let that be the vector

$$s = \left\{\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right\},$$

which is not in $\hat{D}_8$. Thus, s requires an odd number of sign flips to get into $\hat{D}_8$. Then, the next 7 bits `1001011` would indicate that we need to negate the 1st, 2nd, 4th, and 7th from right bits. Since we need an odd number of sign flips, the 8th from right bit is also a sign flip. The sign-decoded vector is then

$$\left\{-\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}\right\},$$

which we can verify is in E_8. Finally, the last bit `1` indicates that we need to add $\frac{1}{4}$, so the final decoded vector is

$$\left\{-\frac{1}{4}, -\frac{3}{4}, \frac{3}{4}, \frac{7}{4}, -\frac{1}{4}, \frac{3}{4}, -\frac{1}{4}, -\frac{1}{4}\right\},$$

which is in $E_8 + \frac{1}{4}$ as desired.

For posterity, we include a copy of our CUDA kernel for matrix-vector multiplication with E8P. This kernel was designed for NVIDIA Ampere and newer GPUs. The same kernel can be found at https://github.com/Cornell-RelaxML/quip-sharp/blob/main/quiptools/quiptools_e8p_gemv.cu.

```
__global__ static void
decode_matvec_e8p_kernel(
    float *__restrict__ output,
    const uint2 *__restrict__ input,
    const uint2 *__restrict__ weights_compressed,
    const uint32_t *__restrict__ codebook_abs,
    int N,
    int K
) {
    int warpId = threadIdx.y;
    int laneId = threadIdx.x;

    for (int iin = blockIdx.x; iin < (N >> 4); iin += gridDim.x) {

        float z0 = 0.0;
        float z1 = 0.0;
        float z2 = 0.0;
        float z3 = 0.0;

        for (int iik = warpId; iik < (K >> 6); iik += 32) {
            uint2 w_compr = weights_compressed[laneId + 32*iik + (K >> 1)*iin];
            uint32_t a = w_compr.x;
            uint32_t b = w_compr.y;

            uint32_t s = b;
            s = s ^ (s >> 4);
            s = s ^ (s >> 8);
            s = s ^ (s >> 16);
            uint32_t sb = (s & 15);
            s = b ^ sb;
            sb = sb | (sb << 16);

            uint32_t input_to_warp = ((const uint32_t*)(&input[16*iik]))[laneId];
            uint32_t shifted_laneId = (laneId & 3) << 3;
```

```
// BLOCK 01
{
uint32_t x = codebook_abs[(a >> 0) & 255];
x = x ^ ((s & 0x11111111) * 14);

uint32_t o = BASE_OFFSET | ((sb & 0x00010001) << 4);

uint32_t w00 = add_as_half2(mask_lop3(x << 4, XMASK, WMASK), o);
uint32_t w01 = add_as_half2(mask_lop3(x << 0, XMASK, WMASK), o);
uint32_t w02 = add_as_half2(mask_lop3(x >> 4, XMASK, WMASK), o);
uint32_t w03 = add_as_half2(mask_lop3(x >> 8, XMASK, WMASK), o);

x = codebook_abs[(a >> 8) & 255];
x = x ^ ((s & 0x22222222) * 7);

o = BASE_OFFSET | ((sb & 0x00020002) << 3);

uint32_t w10 = add_as_half2(mask_lop3(x << 4, XMASK, WMASK), o);
uint32_t w11 = add_as_half2(mask_lop3(x << 0, XMASK, WMASK), o);
uint32_t w12 = add_as_half2(mask_lop3(x >> 4, XMASK, WMASK), o);
uint32_t w13 = add_as_half2(mask_lop3(x >> 8, XMASK, WMASK), o);

uint32_t x_in0 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 0);
uint32_t x_in1 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 1);

asm(
    "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32"
    "␣{␣%0,␣%1,␣%2,␣%3␣},"
    "␣{␣%4,␣%5,␣%6,␣%7␣},"
    "␣{␣%8,␣%9␣},"
    "␣{␣%0,␣%1,␣%2,␣%3␣};"
    : "+f"(z0), "+f"(z1), "+f"(z2), "+f"(z3)
    : "r"(w00), "r"(w10), "r"(w01),  "r"(w11),
      "r"(x_in0), "r"(x_in1)
);

x_in0 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 2);
x_in1 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 3);

asm(
    "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32"
    "␣{␣%0,␣%1,␣%2,␣%3␣},"
    "␣{␣%4,␣%5,␣%6,␣%7␣},"
    "␣{␣%8,␣%9␣},"
    "␣{␣%0,␣%1,␣%2,␣%3␣};"
    : "+f"(z0), "+f"(z1), "+f"(z2), "+f"(z3)
    : "r"(w02), "r"(w12), "r"(w03), "r"(w13),
      "r"(x_in0), "r"(x_in1)
);
}
// BLOCK 23
{
uint32_t x = codebook_abs[(a >> 16) & 255];
s = s >> 2;
x = x ^ ((s & 0x11111111) * 14);

uint32_t o = BASE_OFFSET | ((sb & 0x00040004) << 2);

uint32_t w00 = add_as_half2(mask_lop3(x << 4, XMASK, WMASK), o);
uint32_t w01 = add_as_half2(mask_lop3(x << 0, XMASK, WMASK), o);
uint32_t w02 = add_as_half2(mask_lop3(x >> 4, XMASK, WMASK), o);
uint32_t w03 = add_as_half2(mask_lop3(x >> 8, XMASK, WMASK), o);

x = codebook_abs[(a >> 24) & 255];
```

```
            x = x ^ ((s & 0x22222222) * 7);

            o = BASE_OFFSET | ((sb & 0x00080008) << 1);

            uint32_t w10 = add_as_half2(mask_lop3(x << 4, XMASK, WMASK), o);
            uint32_t w11 = add_as_half2(mask_lop3(x << 0, XMASK, WMASK), o);
            uint32_t w12 = add_as_half2(mask_lop3(x >> 4, XMASK, WMASK), o);
            uint32_t w13 = add_as_half2(mask_lop3(x >> 8, XMASK, WMASK), o);

            uint32_t x_in0 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 4);
            uint32_t x_in1 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 5);

            asm(
                "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32"
                "␣{␣%0,␣%1,␣%2,␣%3␣},"
                "␣{␣%4,␣%5,␣%6,␣%7␣},"
                "␣{␣%8,␣%9␣},"
                "␣{␣%0,␣%1,␣%2,␣%3␣};"
                : "+f"(z0), "+f"(z1), "+f"(z2), "+f"(z3)
                : "r"(w00), "r"(w10), "r"(w01), "r"(w11),
                  "r"(x_in0), "r"(x_in1)
            );

            x_in0 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 6);
            x_in1 = __shfl_sync(FULL_MASK, input_to_warp, shifted_laneId | 7);

            asm(
                "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32"
                "␣{␣%0,␣%1,␣%2,␣%3␣},"
                "␣{␣%4,␣%5,␣%6,␣%7␣},"
                "␣{␣%8,␣%9␣},"
                "␣{␣%0,␣%1,␣%2,␣%3␣};"
                : "+f"(z0), "+f"(z1), "+f"(z2), "+f"(z3)
                : "r"(w02), "r"(w12), "r"(w03), "r"(w13),
                  "r"(x_in0), "r"(x_in1)
            );
            }
        }

        if ((laneId & 1) == 0) {
            atomicAdd(output + (iin << 4) + (laneId >> 1), (laneId & 2) ? z2 : z0);
        }
    }
}
```

C.3. Why not K-Means?

A significant motivating factor behind E8P is that post-incoherence processing, entries of W are approximately Gaussian distributed. However, E8P is uniformly distributed, which raises the question: why not use a K-means based codebook? K-means based codebooks offer strong theoretical performance but have a few issues. First, it is difficult to enforce symmetry in a "learned" K-means codebook. This is crucial to be able to have a compressible codebook. If we force sign symmetry by learning cluster centers on only the positive orthant of a n-dimensional Gaussian, we can get around this but sacrifice accuracy at the axis region. Second, using K-means requires storing a codebook in fp16, whereas the entries of E8P can be stored as 4 bit integers. This means that during inference, the source codebook for a 8 dimension K-means codebook will be 4 times larger than the source codebook of E8P, running the risk of a cache eviction. Finally, we observe that empirically, E8P actually *outperforms* K-means, which is somewhat interesting and suggests that allocating more information to the edge of the distribution, even after incoherence processing, is useful.

C.4. E8P vs. Other Codebook Constructions

Below, we compare the quality of quantized models from E8P vs. other codebooks. The D_4 lattice achieves the 4 dimensional kissing number but has lower dimensionality than E_8, giving poorer shaping. An 8-dimensional K-Means codebook has similar shaping as E8P but worse packing density. Although RHT-transformed weights are approximately Gaussian and not Uniform, we find that a Uniform codebook (E8P) performs better than a Gaussian codebook (K-means)

Table 7. E8P and E_8-based codebooks outperform other codebooks based on lower-dimensional lattices or different distributions. Numbers without fine-tuning.

MODEL	CODEBOOK	CODEBOOK DIM.	BITS	WIKI2 PPL (CTX 4096)	C4 PPL (CTX 4096, C4_NEW)
2-70B	FP16	1	16	3.120	5.533
2-70B	**E8P**	**8**	**2**	**4.156**	**6.535**
2-70B	E_8 LATTICE	8	2.37	3.702	6.082
2-70B	D_4 LATTICE	4	2	4.408	6.797
2-70B	D_4 LATTICE	4	2.21	3.970	6.332
2-70B	K-MEANS	8	2	4.452	6.925

D. Fine-Tuning During Quantization

In Algorithm 5 we describe our fine tuning procedure for QuIP#.

Algorithm 5 QuIP# with Fine-Tuning

```
Require: Unquantized Model $M$, Development Set $\mathcal{D}$, Quantization Order $O$,
Ensure: Quantized Model $M$
  $X \leftarrow M_{\text{embedding}}(\mathcal{D})$
  $C \leftarrow M(\mathcal{D})_{\text{logits}}$
  for Decoder Block $D \in M$ do
    $Y \leftarrow D(X)$
    $X_{train}, Y_{train}, X_{valid}, Y_{valid} \leftarrow$ split$(X, Y)$
    for Linear Layer $L \in D$ in order specified by $O$ do
      $\hat{L} \leftarrow$ QuIP#-NoFT$(L)$
      Disable gradients for the weight matrix (but not $S_U, S_V$) of $\hat{L}$.
      Optimize $D$ to minimize MSE$(D(X_{train}), Y_{train})$ using $X_{valid}, Y_{valid}$ for early stopping.
    end for
    $X \leftarrow Y$
  end for
  {At this point, the learnable parameters in $M$ are the layernorms, all $S_U$ and $S_V$, and the language model head.}
  $\mathcal{D}_{train}, C_{train}, \mathcal{D}_{valid}, C_{valid} \leftarrow$ split$(\mathcal{D}, C)$
  Optimize $M$ to minimize CrossEntropy$(M(\mathcal{D}_{train}), C_{train})$ using $\mathcal{D}_{valid}, C_{valid}$ for early stopping.
```

Table 8. QuIP# outperforms OmniQuant even with grouping. More numbers can be found in the OmniQuant paper.

MODEL	METHOD	EFFECTIVE BITS	WIKI2 PPL (CTX 2048)	C4 PPL (CTX 2048)
2-70B	FP16	16	3.32	5.52
2-70B	QUIP#	2	4.16	6.12
2-70B	OMNIQ W2A16	2	7.81	12.28
2-70B	OMNIQ W2A16 G64	2.25	6.11	7.88
2-70B	OMNIQ W3A16	3	3.92	6.06

E. Additional Results

E.1. QuIP# vs. OmniQuant with Grouping

E.2. QuIP# on Mixtral 8x7B (Jiang et al., 2024) and Falcon 180B (Almazrouei et al., 2023)

Table 9. 2 bit QuIP# without fine-tuning on Mixtral 8x7B, a mixture of experts (MoE), and Falcon 180B, a non-Llama-based model. QuIP# scales to different architectures without issue.

MODEL	BITS	WIKI2	C4	ARCC	ARCE	BOOLQ	PIQA	WINO
MIXTRAL-8X7B	16	3.45	6.85	0.56	0.74	0.85	0.84	0.75
MIXTRAL-8X7B	2	4.69	8.25	0.49	0.68	0.81	0.80	0.73
FALCON-180B	16	3.30	6.31	0.61	0.82	0.87	0.85	0.81
FALCON-180B	2	4.18	7.06	0.58	0.81	0.84	0.84	0.81

E.3. Zeroshot performance for ablation on lattice codebooks and fine-tuning

Table 10. Ablation on lattice codebooks and fine-tuning. QuIP# no FT and E_8 uses the RHT to perform incoherence processing but does not use lattice codebooks or fine-tuning. QuIP# No FT uses lattice codebooks but not fine-tuning. QuIP# uses lattice codebooks and performs fine-tuning.

Model	Method	Bits	ArcC (acc_norm)	ArcE (acc_norm)	BoolQ (acc)	PiQA (acc_norm)	Wino (acc)
2-70	Native	16	48.0	59.7	76.6	80.9	76.8
2-70	QuIP# no FT & no E_8	4	**49.4**	**60.1**	77.6	**80.7**	76.1
2-70	QuIP# No FT	4	48.3	**60.1**	**78.4**	80.6	76.2
2-70	QuIP#	4	48.3	59.4	77.4	**80.7**	**77.1**
2-70	QuIP# no FT & no E_8	3	47.4	59.1	75.8	**80.9**	**77.5**
2-70	QuIP# No FT	3	47.9	**59.9**	**78.8**	79.9	77.0
2-70	QuIP#	3	**48.4**	59.5	74.8	80.3	76.4
2-70	QuIP# no FT & no E_8	2	43.5	56.2	75.1	78.1	**76.0**
2-70	QuIP# No FT	2	47.2	**59.5**	79.1	78.6	74.2
2-70	QuIP#	2	**47.7**	59.1	**80.3**	**79.4**	75.9
2-13	Native	16	44.3	58.0	69.0	79.0	69.9
2-13	QuIP# no FT & no E_8	4	43.7	**58.6**	**70.1**	78.7	69.6
2-13	QuIP# No FT	4	42.9	56.4	67.8	**78.9**	**69.9**
2-13	QuIP#	4	**44.2**	57.7	69.7	**78.9**	**69.9**
2-13	QuIP# no FT & no E_8	3	42.1	55.2	70.0	77.8	**69.5**
2-13	QuIP# No FT	3	41.9	**57.7**	**73.3**	78.1	68.0
2-13	QuIP#	3	**43.3**	**57.7**	69.8	**78.4**	69.1
2-13	QuIP# no FT & no E_8	2	36.3	50.8	67.4	73.4	63.1
2-13	QuIP# No FT	2	37.1	50.1	66.5	75.7	63.6
2-13	QuIP#	2	**41.3**	**55.1**	**68.3**	**77.4**	**67.7**
2-7	Native	16	40.6	53.5	71.0	76.9	67.0
2-7	QuIP# no FT & no E_8	4	39.5	51.9	71.3	76.6	67.3
2-7	QuIP# No FT	4	**40.4**	**53.7**	68.5	**77.2**	67.5
2-7	QuIP#	4	40.1	53.4	**69.9**	76.5	**67.6**
2-7	QuIP# no FT & no E_8	3	38.1	52.6	65.2	76.1	65.1
2-7	QuIP# No FT	3	37.7	53.1	**70.6**	**76.7**	**67.6**
2-7	QuIP#	3	**39.4**	**53.8**	69.7	76.1	66.5
2-7	QuIP# no FT & no E_8	2	29.2	42.5	63.3	68.0	59.0
2-7	QuIP# No FT	2	32.5	42.8	62.3	71.2	62.4
2-7	QuIP#	2	**36.1**	**50.5**	**68.3**	**74.9**	**64.9**

E.4. More Scaling Plots

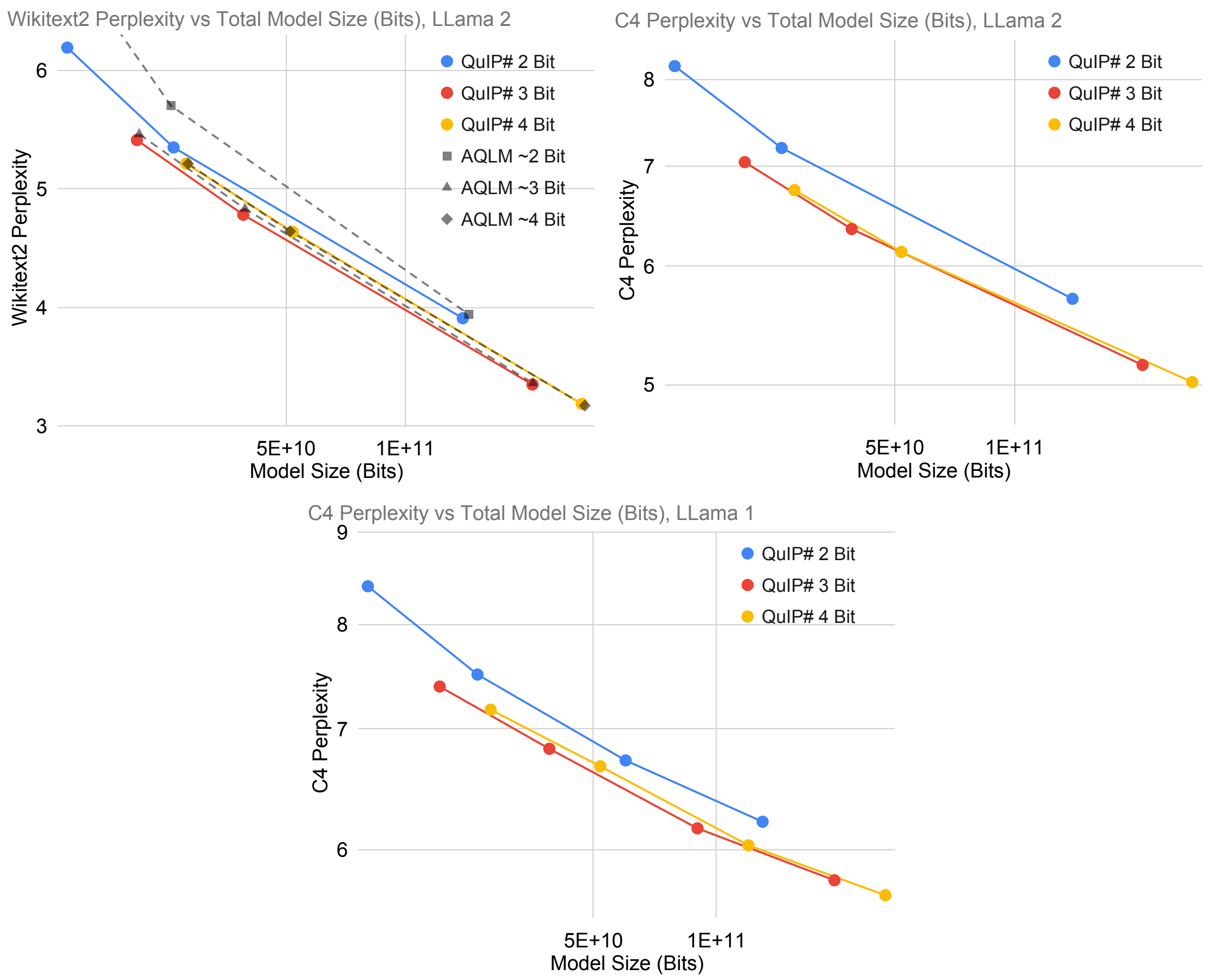

Figure 5. QuIP# scaling. (Top Left) Llama 2 Wikitext 2 perplexity vs AQLM. Context length 4096. QuIP# 2 and 3 bit scale better than AQLM 2 and 3 bit. (Top Right) Llama 2 C4 Perplexity. Context length 4096. (Bottom) Llama 1 C4 Perplexity. Context length 2048.

F. Implementation Details

This section contains implementation details for our Llama experiments. These details also mostly apply to the Mixtral and Falcon numbers except we use the Falcon dataset (Almazrouei et al., 2023) as it is publicly avaiable.

F.1. Bit Accounting

The additional overhead of QuIP# consists of 1KiB for E8P and $16n$ bits for each nD sign vector if using fine tuning or n without. The 1KiB from E8P is shared over all linear layers, so it adds $\ll 0.01$ bits per weight. For a $m \times n$ matrix, the sign vectors take up $\frac{16(n+m)}{nm}$ bits per weight with fine tuning or $\frac{n+m}{nm}$ without. For LLM-sized matrices (e.g. the smallest matrix in Llama 2 7B is 4096×4096), this is still < 0.01 additional bits per weight.

F.2. Hessian Generation

Hessian matrices H were generated with 6144 sequences of a model's native context length (2048 for Llama 1, 4096 for Llama 2) from the RedPajama 1T (Computer, 2023) dataset.

F.3. Hadamard Matrices

We use Hadamard matrices available at Neil Sloane's website (Sloane).

F.4. Perplexity and Zeroshot Evaluation

We use the OPTQ (Frantar et al., 2023) "Wiktext2" and "C4" (not "C4 New") sampling functions to calculate perplexity for our experiments. We use LM Eval (Gao et al., 2023) to calculate zeroshot numbers.

F.5. Scales

In order to achieve good coverage of the codebook, we scale W by $\rho|W|_F$ before quantizing W. For E8P, we used ρ = 0.9, for RVQ 3 bit we used $\rho \approx= 0.98$ for the first stage and ≈ 2.04 for the second stage, for RVQ 4 bit we used $\rho \approx= 1.03$ for the first stage and ≈ 3.45 for the second stage. These numbers were determined by finding the ρ(s) that minimized the quantization error of quantizing a Gaussian to the codebook. The $\approx$ is because different models have slightly different optimal ρ since incoherence processing does not produce an exact Gaussian. The actual numbers for each model were found with a coarse scale sweep but are all close to these values.

F.6. Fine Tuning

For the within-transformer block section of fine-tuning, we use the Adam optimizer (Kingma & Ba, 2017), a learning rate of 5×10^{-5}, batch size of 8, and sequence length equal to the model's native context length. We train on small development dataset of 256 sequences from RedPajama 1T and validate on 128 sequences. We train for 5 epochs (160 steps) and keep the best model parameters based on the validation set. For the end to end tuning, we use the Adam optimizer, a learning rate of 5×10^{-5}, batch size of 1, sequence length equal to the model's context length (except for 70B, where we had to use 3072 to avoid an OOM on our not very well optimized training script), and the same dataset and epoch setup as before. We observe that outside of using a low enough learning rate, the other hyperparameters did not affect fine-tuning much. For the 2 bit models, we used a learning rate of 5×10^{-4} for S_U and S_V (5×10^{-5} for everything else as above) for both the within-block and end to end fine tuning stages.

F.7. Hardware

All experiments were run on NVIDIA A100 GPUs except for the timing numbers, which were measured on a NVIDIA RTX 4090 to see what was possible with the current state-of-the-art NVIDIA consumer GPU. We find that we can quantize Llama 2 70B without fine tuning in under 10 GPU-hours and with fine tuning in around 100 GPU-hours. Both numbers do not include Hessian generation, which can be done once for a model and reused across many different quantization experiments.

F.8. Code and Prequantized Models

Our code is available at `https://github.com/Cornell-RelaxML/quip-sharp` and prequantized QuIP# models are available at `https://huggingface.co/relaxml`.

G. Example Generation

Below are some example generations from Llama 2 70B chat quantized with QuIP# to 2 bits, truncated to 256 tokens.

Prompt: How much wood could a woodchuck chuck if a woodchuck could chuck wood?

QuIP#:
The answer to this tongue-twister is a bit of a mystery. Woodchucks, also known as groundhogs, are rodents that burrow in the ground and are known for their ability to move large amounts of soil. However, there is no scientific evidence to suggest that they can actually chuck wood. The phrase "how much wood could a woodchuck chuck" was first used in a song called "The Woodchuck's Song" in the late 19th century. The song was popularized by the singer and songwriter, Johnny Mercer, and it became a hit in the 1940s. The song's lyrics include the line "How much wood could a woodchuck chuck if a woodchuck could chuck wood?" but they don't provide an answer. In reality, woodchucks don't chuck wood, they dig burrows. They are able to move a lot of soil and dirt as they dig their burrows, but they don't have the ability to move wood. So, the answer to the tongue-

Prompt: Write a song about large language models to the tune of Mariah Carey's "All I Want For Christmas Is You"

QuIP#:
Verse 1:
I don't want a lot for Christmas
There's just one thing I'm wishing for
A language model that's large and bright
To help me with my writing, day and night

Chorus:
All I want for Christmas is a language model
A big, smart, AI that's my ideal
I'll be happy with just a few billion parameters
All I want for Christmas is a language model, yeah

Verse 2:
I'll be good, I won't be naughty
I'll use it for good, not foraughty
I'll generate text that's nice and neat
And make my writing life a treat

Chorus:
All I want for Christmas is a language model
A big, smart, AI that's my ideal
I'll be happy with just a few billion parameters
All I want for Christmas is a language model, yeah

Bridge:
I'

Prompt: Write some CUDA code that does a matrix multiply

QuIP#:
Here is a simple example of a matrix multiply using CUDA:

```
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void matrixMultiply(float *A, float *B, float *C, int rowsA, int co
lsA, int rowsB, int colsB) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < rowsA && j < colsB) {
    float sum = 0;
    for (int k = 0; k < rowsB; k++) {
      sum += A[i + k * colsA] * B[k + j * rowsB];
    }
    C[i + j * rowsB] = sum;
  }
}

int main() {
  int rowsA = 3;
  int colsA = 3;
  int rowsB = 3;
```