QIGen: Generating Efficient Kernels for Quantized Inference on Large Language Models

Tommaso Pegolotti [1] **Elias Frantar** [2] **Dan Alistarh** [3] **Markus Püschel** [1]

Abstract

We present ongoing work on a new automatic code generation approach for supporting quantized generative inference on LLMs such as LLaMA or OPT on off-the-shelf CPUs. Our approach is informed by the target architecture and a performance model, including both hardware characteristics and method-specific accuracy constraints. Results on CPU-based inference for LLaMA models show that our approach can lead to high performance and high accuracy, comparing favorably to the best existing open-source solution. A preliminary implementation is available at https://github.com/IST-DASLab/QIGen.

1. Introduction

The impressive performance of generative large language models (LLMs) (Black et al., 2022; Zhang et al., 2022; Touvron et al., 2023) has led to significant interest in executing them on user devices with limited computational power. Interestingly, the computational envelope of generative workloads renders this plausible: when executing the popular one-token-at-a-time generation based on a given cached context, the operational cost of obtaining single outputs is relatively small, as the computation can be mapped onto relatively inexpensive matrix-vector products, as opposed to the massive matrix-matrix multiplications that are otherwise common in deep learning.

Thus, the key challenge in personalized generative LLM inference becomes *memory*: well-performing models have extremely large parameter counts, which often exceed the memory capacity of consumer devices, and induce high memory transfer costs at runtime, overwhelming bandwidth. To address this issue, a series of *quantization-based methods* specialized to LLMs have been recently proposed (Dettmers et al., 2022; Dettmers & Zettlemoyer, 2022; Frantar et al., 2022; Park et al., 2022; Xiao et al., 2022; Yao et al., 2022), which work by reducing the bitwidth of data types used for storing weights, activations, or both, with the goal of minimizing the impact on accuracy.

Focusing specifically on generative inference, where the size of the weights is the main bottleneck, the currently best-performing method is GPTQ (Frantar et al., 2022), which achieves near-lossless quantization to 4-bit weights, and can even accurately support 2 and 3-bit weights by reducing the granularity to smaller weight groups, e.g., by jointly quantizing blocks of 64 weights using a shared scale and zero-point. Similar grouping techniques can also allow simpler round-to-nearest (RTN) quantization to preserve accuracy using 4-bit weights (Dettmers & Zettlemoyer, 2022).

Given this algorithmic progress, a remaining key challenge is the efficient system support for these compressed numerical formats, to execute LLMs on user devices accurately and fast. Existing academic proposals such as LLM.int8() (Dettmers et al., 2022), GPTQ (Frantar et al., 2022), and SmoothQuant (Xiao et al., 2022), and open-sourced solutions such as llama.cpp (Gerganov, 2023), manually develop custom kernels for their specific target hardware, such as GPUs or CPUs. Unfortunately, this approach can be extremely time-intensive and error-prone, and requires potentially re-writing kernels from scratch to support new quantization formats and target hardware.

In this paper, we present ongoing work on a new automatic code generation approach, called QIGen, for obtaining efficient and general kernels for generative LLM inference of varying bitwidth. At a high level, our approach provides customized efficient implementations of the low-level matrix operations required to support multiplication operations on quantized variants of LLM weight matrices. Our approach is based on a performance model which is informed both by hardware characteristics, e.g., cache size, and by accuracy constraints pertaining to the quantization methods, e.g., the use of weight grouping. We present results generating efficient low-bitwidth kernels for general purpose CPUs supporting the popular AVX2 intrinsics, which we interface with Pytorch (Paszke et al., 2019), and showcase on the accurate LLaMA family (Touvron et al., 2023).

[1]ETH Zurich [2]IST Austria [3]IST Austria & Neural Magic. Correspondence to: Tommaso Pegolotti <tommaso.pegolotti@inf.ethz.ch>, Dan Alistarh <dan.alistarh@ist.ac.at>.

Accepted to the ICML 2023 FoMo Workshop.

2. Background

Quantization. Quantization is an efficient compression technique for reducing memory utilization by representing data using a limited number of values, typically integer levels. We define a *quantization function* as a map Q from real numbers to integers. Formally,

$$Q : (\mathbb{R}, \mathbb{N}) \mapsto [0, 2^{\mathrm{b}}),$$

where b is the number of bits we want to use to represent the new value. Given a vector $x \in \mathbb{R}^n$, we define $Q(x, \mathrm{b})$ as

$$Q(x, \mathrm{b}) = \mathrm{rnd}\left(\frac{x - \min(x)}{\max(x) - \min(x)}(2^{\mathrm{b}} - 1)\right),$$

where $\max(x)$ and $\min(x)$ is the maximum and minimum value in x, respectively, and rnd rounds to nearest. This equation can be rewritten as $x_q = \mathrm{rnd}((x - \mathrm{z})\mathrm{s}(x))$, with $\mathrm{z} = \min(x)$ and $\mathrm{s}(x) = (2^{\mathrm{b}} - 1)/(\max(x) - \min(x))$. Similarly, the corresponding dequantization function is

$$D(x_q) = \mathrm{s}(x_q \mathrm{z}).$$

As example, consider the dot-product between $y \in \mathbb{R}^n$ and $x_q \in [0, 2^{\mathrm{b}})$. The resulting value is given by

$$\langle y, \mathrm{s}(x_q - \mathrm{z})\rangle = \mathrm{s}(\langle y, x_q\rangle - \mathrm{z}\langle y, 1\rangle). \tag{1}$$

To improve accuracy, we can increase the quantization granularity and store more s and z values for each vector. We denote the resulting partition by groups. To compute the dot-product using these additional values, we only need to modify 1. In particular, we rewrite it as

$$\sum_i^P \langle y_i, \mathrm{s}_i(x_{iq} - \mathrm{z}_i)\rangle = \sum_i^P \mathrm{s}_i\langle y_i, x_{iq}\rangle - \sum_i^P \mathrm{z}_i\langle y_i, 1\rangle, \tag{2}$$

where P is the number of groups.

LLM quantization. There has been significant focus on accurate post-training quantization (PTQ) methods (Nagel et al., 2019) that scale and are accurate for LLMs. Early work (Yao et al., 2022; Dettmers et al., 2022; Park et al., 2022) used direct rounding to the nearest quantization level (RTN), reducing group size to obtain higher accuracy at the cost of more space. LLM.int8() (Dettmers et al., 2022) quantized activations as well, isolating "outlier features" for which higher bit-width is used. These approaches induce quantization errors of 5–10% in perplexity increase for OPT (Zhang et al., 2022) or LLaMA (Touvron et al., 2023) models, relative to the uncompressed baseline. GPTQ (Frantar et al., 2022) proposed a higher-accuracy approach (e.g., 3–5% perplexity increase at 4-bit), via an approximate solver minimizing the layer-wise squared error between the quantized and original layers. Dettmers & Zettlemoyer (2022) provided an in-depth overview of the accuracy-compression trade-offs underlying these methods, establishing that 4-bit quantization is an optimal point for round-to-nearest-based methods, whereas higher compression can be achieved via data-aware methods such as GPTQ.

load(address)	load from memory address
store(address, a)	stores a at memory address
broadcast(a)	fills a register with a
fmadd(a, b, c)	returns $a \cdot b + c$
reduce_add(a)	returns the sum of the elements of a
srli(a, i)	returns a $>> i$
and(a, m)	returns a bitwise and m
cvt_int_float(a)	converts a to float

Table 1: List of vector instructions used in the implementation and their semantics.

All the above methods focused on GPU inference as their main target scenario. By contrast, there has been much less focus on *CPU-based inference*; for this, the open-source LLaMA.cpp/GGML project (Gerganov, 2023) can provide reasonable generative performance on end devices such as Intel/AMD/ARM CPUs, showing that running models locally in such setups is feasible.

Notation. We simplify the names of AVX SIMD vector instructions for readability in our exposition. The semantics of the instructions can be seen in Table 1.

3. Code Generation

We consider CPU-based generative inference as the motivating setup for our work, although our techniques are general, and should be extensible to other settings as well.

Data format. Our implementation of linear layers uses quantized weights obtained via GPTQ, in which the weights are represented using 32-bit integers to store multiple consecutive values. For instance, with 4-bit quantization, a single "unit" can represent 8 values, and with 2-bit quantization, it can represent 16 values. Some bit granularities may require specialized implementations. For example, if we use 3-bit, we store 32 values in three consecutive integers.

We quantize the weight matrices in a column-wise manner. For 4-bit quantization, each entry (i, j) in the quantized matrix contains the values at indices $(8i : 8i + 7, j)$ of the uncompressed matrix. For each column j, we also store one scale s_j and one z_j value. Additionally, we can also quantize the z_j value. The changes in the matrices after quantization are illustrated in Figure 1.

The total memory used to store a weight matrix of size

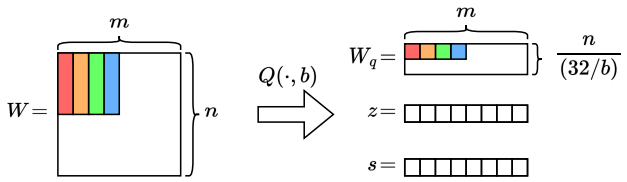


Figure 1: Reduction in matrix size due to quantization. Given a weight matrix W of size $n \times m$, we obtain a compressed matrix W_q of size $n/(32/\mathrm{b}) \times m$ and two one dimensional vectors z and s of size m. We quantize the elements column-wise as shown by the colors. For example, 8 values in the red rectangle become 1 value in the red square.

$N \times M$ is thus $\mathrm{b}NM + 32M + \mathrm{b}M$ instead of $32NM$. The $32M$ is for the s_j stored in single-precision floating point. We obtain a significant reduction in memory usage by a factor of $\approx 32/\mathrm{b}$.

Computation. LLMs typically comprise a series of linear layers, where the input is a vector. As a result, matrix-vector multiplications (GEMV) form the core of our computations. In Algorithm 1, we present a straightforward implementation of a 4-bit qGEMV (quantized general matrix-vector multiply) based on the factorization in (1). The approach involves computing the dot-products using integer representations and scaling and transposing the result once after the final reduction. To extract eight rows from the quantized weight matrix, considering 4-bit quantization, we utilize the auxiliary function **`unpack`** as presented in Algorithm 2.

Algorithm 1 4-bit qGEMV routine.

In: $W \in \mathbb{N}^{\frac{n}{\mathrm{b}} \times m}$, $s \in \mathbb{R}^m$, $z \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, $\hat{x} \in \mathbb{R}$
Out: $y = xW$
for $j = 0 : m$ **do**
  $acc \leftarrow$ **`broadcast`**(0)
  for $i = 0 : 8 : n$ **do**
    $w \leftarrow$ **`load`**$(\&W[i/8][j])$
    $l_{0:8} \leftarrow$ **`unpack`**(w)
    for $ii = 0 : 8$ **do**
      $x_{ii} \leftarrow$ **`broadcast`**$(x[i + ii])$
      $acc \leftarrow$ **`fmadd`**(x_{ii}, l_{ii}, acc)
    end for
  end for
  $w[j] \leftarrow s[j] * ($**`reduce_add`**$(acc) - z[j] * \hat{x})$
end for

Algorithm 2 AVX2 4-bit Unpack Routine.

In: v
Out: l_0 to l_7
mask $\leftarrow$ **`broadcast`**(15)
for $i = 0 : 8$ **do**
  $s_i \leftarrow$ **`srli`**$(v, i * 4)$
  $a_i \leftarrow$ **`and`**(s_i, mask)
  $l_i \leftarrow$ **`cvt_int_float`**(a_i)
end for

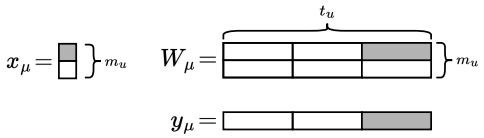


Figure 2: Visual representation of data used in the Micro-GEMV step. To store x_μ, W_μ, and y_μ we need m_u, $m_u t_u$ and t_u registers, respectively. In each step of the algorithm we multiply one element of x_μ by a register containing 8 values of W_μ, to obtain 8 values of y_μ. Considering the grey cells, we have $y_{\mu,3} = y_{\mu,3} + x_{\mu,1} \cdot W_{\mu,1,17:24}$.

Mini-GEMV. To improve performance, we exploit that weight matrices in a neural network are set at compile time. By considering the size of the weight matrices and the cache size of the CPU, we can store the matrices in sequential blocks using the Z-curve order (Valsalam & Skjellum, 2002). This approach improves spatial locality, and thus cache usage, minimizes false sharing when using multiple threads, and minimizes TLB misses.

We utilize a model similar to (Yotov et al., 2005) for optimizing cache performance by dividing the computation into Mini-GEMVs. Specifically, we partition the input and output vectors and the weight matrix into blocks of sizes $m_b \times 1$ and $m_b \times \mathrm{b}t_b/32$, respectively. The term $\mathrm{b}t_b/32$ accounts for packing multiple values into a single 32-bit integer. For temporal locality, we set the block dimensions such that

$$32m_b + \mathrm{b}m_b t_b + 32t_b \leq \gamma, \tag{3}$$

where γ represents the size in bits of the L1 data cache.

Micro-GEMV. In the next step aim for an efficient innermost loop of the computation. In particular, we use different instructions based on the specification of the targeted CPU. The specification may include the available instructions (such as AVX2, AVX512, or AMX), the number of floating-point ports, and the number of available FMA (fused multiply-add) ports. By tailoring the instructions to the specific CPU, we can maximize register usage and instruction-level parallelism, leading to improved performance during computation.

In addition, we divide the Mini-GEMVs further into Micro-GEMVs operating on an m_u-sized part x_μ of x, t_u-sized part y_μ of y and thus an $mu \times t_u$ block W_μ of W as shown in Fig. 2. x_μ and W_μ need to load different values, while y_μ remains in registers. We show a sketch of the modified innermost loop in Algorithm 3. We set

$$m_u + m_u t_u + t_u \leq \eta, \tag{4}$$

where η is the number of vector registers, to reduce the number of register spills. For example, AVX/AVX2 have $\eta = 16$.

Finally, we include m_u and t_u in (3) to simplify generation and reduce cleanup code. The final model equation is therefore

$$\begin{aligned} 32m_b + \mathrm{b}m_b t_b + 32t_b \leq \gamma, \\ t_b \mod t_u = 0, \quad m_b \mod m_u = 0. \end{aligned} \tag{5}$$

We find m_u and t_u through empirical search. With these fixed, we maximize the left-hand side of the inequality to maximize cache utilization.

Algorithm 3 AVX2 4-bit μqGEMV Routine.

$y_1 \leftarrow$ **load**$(\&y[j])$
... // Load the values of y in t_u registers, and use them as vector accumulators.
$y_{t_u} \leftarrow$ **loadu**$(\&y[j + t_u])$
for $i = 0 : m_u : m$ **do**
  $w_{1,1} \leftarrow$ **load**$(\&W[i/8][j])$
  ... // Load the values of W in $t_u m_u$ registers.
  $w_{t_u,m_u} \leftarrow$ **load**$(\&W[(i + m_u)/8][j + t_u])$
  $x_1 \leftarrow$ **broadcast**$(\&x[i])$
  ... // Load the values of x in m_u registers.
  $x_{m_u} \leftarrow$ **broadcast**$(\&x[i + m_u])$
  ... // Compute
end for
store$(\&y[j], y_1)$
... // Store back t_u registers.
store$(\&y[j + t_u], y_{t_u})$

Finally, we present an overview of the code generation meta-algorithm below.

Algorithm 4 QIGen Generation Overview.

1: Fetch γ and η
2: Find m_u and t_u by optimizing (4) via search.
3: Find m_b and t_b by maximising the left-hand side of (5) using integer-programming.
4: Generate the Micro-GEMV kernels for the CPU.

4. Evaluation

We assess the effectiveness and precision of our implementation by comparing it with the Python bindings for llama.cpp (Gerganov, 2023)[1], and by presenting the perplexity values on the standard `wikitext2` dataset (Merity et al., 2016). For this preliminary version, we have executed our generator on the AVX2 instruction set. However, the instructions that we use have equivalents on all SIMD vector architectures.

Goals and setup. We compare our approach to llama.cpp, both in terms of inference throughput (tokens generated / second) as well as in terms of accuracy of the resulting models, measured in terms of perplexity (PPL). We use models generated using the GPTQ quantization method (Frantar et al., 2022), whereas llama.cpp uses pre-generated models using their custom `q4_0` quantization format. For performance measurements we use an AMD EPYC 7742 64-Core processor running 64 threads. We compile our code using `gcc 9.4.0` with `-O3, -mavx, -mavx2, -mfma, -march=native, -ffast-math, -ftree-vectorize` flags, and we parallelize using `OpenMP 4.5`. We compile llama.cpp using OpenBLAS.

Accuracy. We begin by examining the accuracy (perplexity) of our generated models on the `wikitext2` dataset, which is standard in this setting (Frantar et al., 2022; Yao et al., 2022). Moreover, Dettmers and Zettlemoyer have shown that perplexity is closely correlated with average performance across zero-shot tasks (Dettmers & Zettlemoyer, 2022). The results for running GPTQ with standard parameters across the LLaMA model family are shown in Table 2.

	llama-7b (5.68)			llama-13b (5.09)			llama-30b (4.10)		
Group size	4bit	3bit	2bit	4bit	3bit	2bit	4bit	3bit	2bit
128	5.81	6.43	23.58	5.19	5.51	15.88	4.21	4.63	11.13
64	5.79	6.23	14.15	5.18	5.49	11.13	4.19	4.55	9.10
32	5.77	6.11	10.24	5.15	5.40	8.37	4.18	4.47	7.22
16	5.76	5.99	8.30	5.13	5.31	7.11	4.16	4.39	6.19

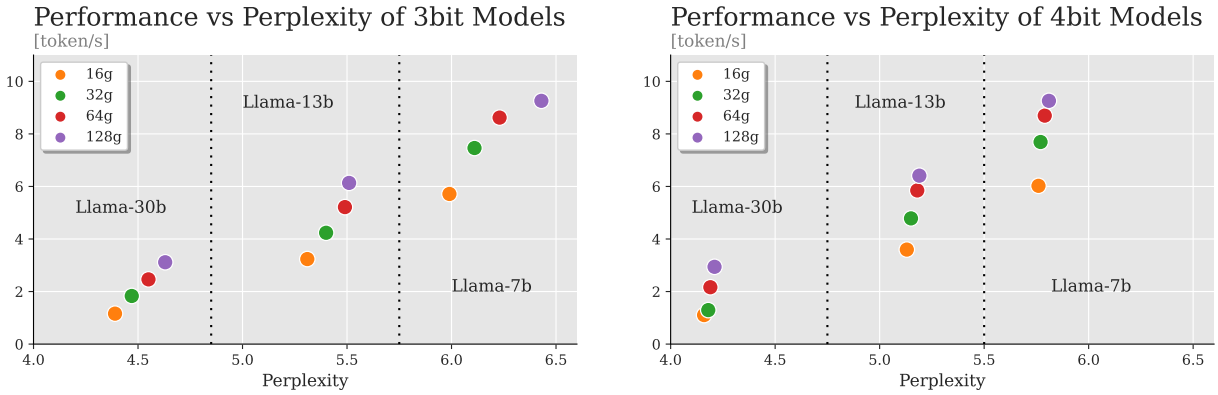
Table 2: Perplexity values evaluated on the `wikitext2` dataset. The perplexity of the floating point model is given in brackets next to their name.

We observe that 4bit quantization generally preserves accuracy, across all models within a few relative percentage points, and that results improve when decreasing group size. (The only exception is LLaMA-7b group size 32, for which we believe some additional hyper-parameter tuning may be beneficial.) As also noted in the original paper, the relative accuracy drop decreases with model size, to the point where small group sizes are nearly lossless on the larger models.

Throughput performance. Next, we compare the performance of our generated kernels to those of llama.cpp. For a fair comparison, we consider models quantized to a 4bit format. For llama.cpp, we specifically executed their `q4_0` quantization format, which yields the best performance among all their supported formats. (According to their documentation, this would correspond to a group size of 32 using our implementation.)

We generate different variants for our kernels, varying the grouping between 16 and 128 elements, as well as using full columns, trading off accuracy for space. For full columns, we can use Equation (1) to reduce the number of floating point operations and loads in the innermost loop. On the other hand, for grouped models, we need to use Equation (2). Note that, while Equation (1) requires storing the values of y only once, Equation (2) requires n/g additional stores, where n is the number of rows and g is the group size.

[1] https://github.com/abetlen/llama-cpp-python



(a) (b)

Figure 3: Changes in performance and perplexity between different group sizes used. In 3(a) and in 3(b) we give the measurements for our 3bit and 4bit implementations respectively. Overall we do not notice major performance difference between 3bit and 4bit kernels.

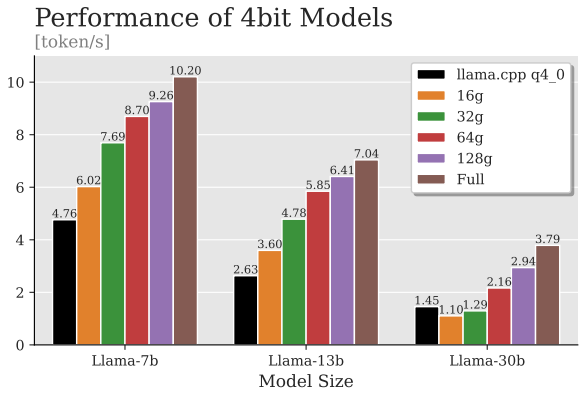


Figure 4: Performance comparison in $token/seconds$ of our different implementations, grouped and non grouped, vs. llama.cpp $q4\_0$ quantization method. We notice performance benefits arising from bigger groups.

	Version	Group Size	Memory (MiB)
LlaMA 7b	FP32	-	26555
	3 Bit	128	8072
	4 Bit	128	8814
LlaMA 13b	FP32	-	50712
	3 Bit	128	14333
	4 Bit	128	16483
LlaMA 30b	FP32	-	125614
	3 Bit	128	31173
	4 Bit	128	37176

Table 3: Comparison of memory usage in MiB used to generate a 128 token sentence.

From the results shown in Fig. 4, we observe that our kernels outperform llama.cpp by up to 2.6× when using full columns on the 13 billion parameter model, and by approximately 2× when using group implementations. However, we observed a lower performance than llama.cpp for the 30b case for group sizes 16 and 32. We believe that this is due to the assumptions we introduce to solve (5), and should be improvable by further examination.

We investigate the full accuracy/performance trade-off in Fig. 3, where we show the relationship between the performance in tokens/seconds and the perplexity of our 3bit and 4bit models in Fig. 3(a) and Fig. 3(b) respectively. We found that for smaller group sizes, performance decreased while accuracy increased. It is important to note that overall, the performance between the two implementations is similar.

Memory consumption. Finally, we report in Table 3 the total memory in MiB used to generate a 128 token long sentence using floating point weights and our 3bit and 4bit quantized kernels with a group size of 128. The results were obtained for LlaMA models with 7, 13, and 30 billion parameters. The 3bit kernels showed a reduction of up to 4x, while the 4bit kernels showed a reduction of up to 3.3x times compared to the floating point implementations.

5. Discussion

We provided evidence that an automatic code generation approach can yield strong results for quantized inference over large language models. Our results show that one can obtain state-of-the-art CPU inference performance using our methods, with minimal accuracy loss when compared to the uncompressed baseline. Our results can be extended along several directions: improving practical performance of existing kernels through fine-tuning, as well as targeting

different CPU architectures and accelerator hardware.

References

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

Dettmers, T. and Zettlemoyer, L. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*, 2022.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Gerganov, G. llama.cpp: Low-Latency Audio Streaming Library for C++. https://github.com/ggerganov/llama.cpp, 2023. Accessed: May 30, 2023.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Nagel, M., Baalen, M. v., Blankevoort, T., and Welling, M. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 1325–1334, 2019.

Park, G., Park, B., Kwon, S. J., Kim, B., Lee, Y., and Lee, D. nuQmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Valsalam, V. and Skjellum, A. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14:805–839, 08 2002. doi: 10.1002/cpe.630.

Xiao, G., Lin, J., Seznec, M., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., and Stodghill, P. A comparison of empirical and model-driven optimization. *Proceedings of the IEEE*, 93 (2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.