Compressing Large Neural Networks

Algorithms, Systems and Scaling Laws

by

Elias Frantar

September, 2024

A thesis submitted to the
Graduate School
of the
Institute of Science and Technology Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Committee in charge:

Vladimir Kolmogorov, Chair
Dan Alistarh
Christoph Lampert
Nir Shavit

The thesis of Elias Frantar, titled *Compressing Large Neural Networks*, is approved by:

Supervisor: Dan Alistarh, ISTA, Klosterneuburg, Austria

Signature: _____

Committee Member: Christoph Lampert, ISTA, Klosterneuburg, Austria

Signature: _____

Committee Member: Nir Shavit, Massachusetts Institute of Technology, Cambridge, USA

Signature: _____

Defense Chair: Vladimir Kolmogorov, ISTA, Klosterneuburg, Austria

Signature: _____

Signed page is on file

ISTA Thesis, ISSN: 2663-337X

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I accept full responsibility for the content and factual accuracy of this work, including the data and their analysis and presentation, and the text and citation of other work.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: _____

Elias Frantar
September, 2024

Signed page is on file

Abstract

Large language models (LLMs) have made tremendous progress in the past few years, from being able to generate coherent text to matching or surpassing humans in a wide variety of creative, knowledge or reasoning tasks. Much of this can be attributed to massively increased scale, both in the size of the model as well as the amount of training data, from 100s of millions to 100s of billions, or even trillions. This trend is expected to continue, which, although exciting, also raises major practical concerns. Already today's 100+ billion parameter LLMs require top-of-the-line hardware just to run. Hence, it is clear that sustaining these developments will require significant efficiency advances.

Historically, one of the most practical ways of improving model efficiency has been compression, especially in the form of sparsity or quantization. While this has been studied extensively in the past, existing accurate methods are all designed for models around 100 million parameters; scaling them up to ones literally $1000\times$ larger is highly challenging. In this thesis, we introduce a new unified sparsification and quantization approach OBC, which through additional algorithmic enhancements leads to GPTQ and SparseGPT, the first techniques fast and accurate enough to compress 100+ billion parameter models to 4- or even 3-bit precision and 50% weight-sparsity, respectively. Additionally, we show how weight-only quantizion does not just bring space savings but also up to $4.5\times$ faster generation speed, via custom GPU kernels.

In fact, we show for the first time that it is possible to develop an FP16 times INT4 mixed-precision matrix multiplication kernel, called Marlin, which comes close to simultaneously maximizing both memory and compute utilization, making weight-only quantization highly practical even for multi-user serving. Further, we demonstrate that GPTQ can be scaled to widely overparametrized trillion-parameter models, where extreme sub-1-bit compression rates can be achieved without any inference slow-down, by co-designing a bespoke entropy coding scheme together with an efficient kernel.

Finally, we also study compression from the perspective of someone with access to massive amounts of compute resources for training large models completely from scratch. Here the key questions evolve around the joint scaling behavior between compression, model size, and amount of training data used. Based on extensive experimental results for both vision and text models, we introduce the first scaling law which accurately captures the relationship between weight-sparsity, number of non-zero weights and data. This further allows us to characterize the optimal sparsity, which we find to increase the longer a fixed cost model is being trained.

Overall, this thesis presents contributions to three different angles of large model efficiency: affordable but accurate algorithms, highly efficient systems implementations, and fundamental scaling laws for compressed training.

Acknowledgements

Overall, I think I could have hardly hoped for a better PhD: exciting projects, impactful results, as well as many learnings professional and personal. This was the result of lots of hard work, some luck, but to a large extent also major support from many sides.

First of all, I would like to thank my advisor Dan Alistarh for fully believing in me, providing lots of scientific and professional support, as well as always being available to chat about my work, the field of machine learning, or really anything else. Next, I would like to thank all my collaborators for their important contributions to the various projects that I have lead or been part of over the course of my PhD. Additionally, I would like to thank all my colleagues for the many interesting discussions, and the very friendly and productive general atmosphere. I feel really lucky to have been part of such a great research group.

I would also like to thank the European Research Council (ERC) for funding much of my research under the European Union's Horizon 2020 programme (grant agreement No. 805223 ScaleML), as well as ISTA's IT department for their support with the cluster infrastructure. I am also very grateful to have been a student researcher at Google DeepMind twice over the course of my PhD.

have participated twice in Google DeepMind's student research programme, which Additionally, I would like to thank Google DeepMind for providing computational resources and funding as a part of the Student Researcher programme.

Looking back a bit further, I would like to thank Christoph Lampert for getting me excited about ML research during an internship in his group. Similarly, I would like to acknowledge David Gilday and Mike Dobson whose Rubik's Cube solving robots have inspired me to keep trying to push beyond known limits, first on my own robots and then on my PhD work.

Last but certianly not least, I would like to extend my gratitude to my friends and family, and in particular my parents, who have always supported me, in good and bad times during and before my PhD.

About the Author

Elias Frantar completed his bachelor's and master's in Computer Science at TU Wien; for the latter he received an Austrian State Prize awarded to the top $\approx 0.3\%$ of graduates in Austria. He first joined ISTA for an internship in the Lampert group in fall 2019, before coming back as a PhD student in the Alistarh group the following year. Overall, his research is focused on making large machine learning models more efficient. Most prominently, he developed GPTQ and SparseGPT, the first successful low-bit quantization and sparsification algorithms targeting extremely large models. Those techniques have found wide adoption in open-source software and have inspired many follow-up papers. During his PhD, Elias also did two internships at Google DeepMind, where we introduced the first scaling law for weight-sparse foundation models. Generally, Elias really loves optimizing things; this even extends to his hobbies, like creating very fast Rubik's Cube solving robots.

List of Collaborators and Publications

This thesis is primarily based on Elias Frantar's following first-author papers, for which we also provide a detailed list of contributions:

- Elias Frantar, Sidak Pal Singh, and Dan Alistarh. Optimal Brain Compression: A framework for accurate post-training quantization and pruning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022 — Section 3.1
  - EF proposed the OBC algorithm, developed its efficient implementation, conducted all experiments and wrote the majority of the paper.
  - SPS first suggested the use of the OBS framework for quantization and derived the corresponding Optimal Brain Quantization formula.
  - DA provided high-level guidance for the project and contributed to the writing, in particular the Introduction and Conclusion.
- Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training compression for generative pretrained transformers. In *International Conference on Learning Representations (ICLR)*, 2023 — Section 3.2
  - EF came up with the GPTQ algorithm, implemented it as well as the corresponding CUDA kernel, performed perplexity-based experiments and wrote all technical sections of the paper.
  - SA implemented a framework for zero-shot evaluations and executed corresponding experiments.
  - TH and DA provided high-level guidance for the project and contributed significantly to the writing of the non-technical sections in the paper.
- Elias Frantar and Dan Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning (ICML)*, 2023 — Section 3.3
  - EF developed the algorithm, ran experiments and wrote the majority of the paper.
  - DA provided high-level guidance for the project and contributed to the writing, in particular the Introduction and Conclusion.
- Elias Frantar and Dan Alistarh. QMoE: Sub-1-bit compression of trillion parameter models. In *Machine Learning and Systems*, 2024 — Section 4.2
  - EF developed the QMoE framework, as well as the corresponding bespoke compression format and its inference kernel, and wrote the majority of the paper.

  - DA provided high-level guidance for the project and supported writing the paper.

- Elias Frantar, Carlos Riquelme, Neil Houlsby, Dan Alistarh, and Utku Evci. Scaling laws for sparsely-connected foundation models. In *International Conference on Learning Representations (ICLR)*, 2024 — Section 5.1

  - EF developed the experiment code-base, executed the majority of experiments, identified the core scaling law and defined the concept of optimal sparsity. Additionally, he did the majority of the writing.
  - CR ran pruning pretrained model experiments and provided useful suggestions during discussions.
  - NH and DA provided useful suggestions and helped improve the paper's writing.
  - UE provided high level-guidance of the project, helped improve the paper's writing and ran decoder-only model experiments.

Additionally, Section 4.1 is based on a public software project [FA24a], developed exclusively by Elias Frantar with high-level guidance of Dan Alistarh, and the corresponding parts of the following paper preprint: Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. MARLIN: mixed-precision auto-regressive parallel inference on large language models. *arXiv preprint arXiv:2408.11743*, 2024. The contributions of RLC, JC and TH (Sparse MARLIN and end-to-end benchmarks) are not included in this thesis.

During his PhD, Elias Frantar also authored or co-authored several other papers, which are not part of this thesis:

- Elias Frantar, Eldar Kurtic, and Dan Alistarh. M-FAC: Efficient matrix-free approximations of second-order information. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021
- Elias Frantar and Dan Alistarh. SPDY: Accurate pruning with speedup guarantees. In *International Conference on Machine Learning (ICML)*, 2022
- Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The Optimal BERT Surgeon: Scalable and accurate second-order pruning for large language models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022
- Denis Kuznedelev, Eldar Kurtic, Elias Frantar, and Dan Alistarh. CAP: correlation-aware pruning for highly-accurate sparse vision models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023
- Eldar Kurtic, Elias Frantar, and Dan Alistarh. Ziplm: Inference-aware structured pruning of language models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024
- Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. SpQR: A sparse-quantized representation for near-lossless llm weight compression. In *International Conference on Learning Representations (ICLR)*, 2024

- Ilia Markov, Kaveh Alimohammadi, Elias Frantar, and Dan Alistarh. L-GreCo: Layerwise-adaptive gradient compression for efficient data-parallel deep learning. In *Conference on Machine Learning and Systems (MLSys)*, 2024
- Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *International Conference on Machine Learning (ICML)*, 2024
- Ionut-Vlad Modoranu, Aleksei Kalinov, Eldar Kurtic, Elias Frantar, and Dan Alistarh. Error feedback can accurately compress preconditioners. In *International Conference on Machine Learning (ICML)*, 2024
- Arshia Soltani Moakhar, Eugenia Iofinova, Elias Frantar, and Dan Alistarh. SPADE: Sparsity-guided debugging for deep neural networks. In *International Conference on Machine Learning (ICML)*, 2024

Finally, Section 3.3.4 contains a paragraph about the sparse kernel contributed by Elias Frantar to the preprint of Kurtic et al. [KKF+23].

Table of Contents

CHAPTER 1

Introduction

Since their initial image recognition breakthrough in 2011 [KSH12], *deep neural networks* have lead to tremendous progress in many domains. Nowadays, the most accurate approaches for understanding audio [RKX+23], detecting objects in videos [RDGF16] and classifying text [DCLT19] are all based on machine learning. Yet, perhaps the most astonishing advancements have happened only in the past few years, in the area of *generative models*. Kickstarted in 2020 by GPT3 [BMR+20], a model capable of producing novel text basically indistinguishable from human writing, researchers have been developing more and more powerful such *Large Language Models (LLMs)*. Their most recent variants cannot just generate coherent content about virtually any topic, but also write working code and even match or outperform humans in a wide variety of knowledge, logic or reasoning tasks [AAA+23, RST+24]. These abilities stem from *Transformer* models [VSP+17], a particularly robust type of neural network architecture, trained at absolutely enormous scale: trillions of parameters [FZS22] fitted on trillions of data points [TMS+23]. That not enough, current evidence strongly suggests that capabilities will keep improving by increasing the scale even further [CND+23]. While this is an exciting prospect, it is also somewhat worrying from a practical perspective. Such LLMs will quickly become basically impossible to run for individuals without access to top-of-the-line hardware, and even large companies with massive compute resources will struggle to serve these models to users in an economic manner. These are not just futuristic problems: already today's state-of-the-art models require 100s of GBs of expensive GPU memory to use, and they are being served to 100s of millions of people. Consequently, it is clear that improving the efficiency of such models will be a highly important research direction going forward.

How to make machine learning models more efficient has already been extensively studied long before the rise of LLMs [HABN+21, GKD+21]. Amongst the most successful set of related techniques is the field of *compression*, where networks are made smaller and faster by removing various kinds of redundancies. Two of the most popular such approaches are *sparsity* and *quantization*. The former aims to remove model components and the latter reduces numerical precision, while trying to minimize accuracy loss as much as possible. This can be highly effective: for instance, one can remove up to 90% of model parameters in classical vision models with almost no impact on model function [SA20, PIVA21]. Corresponding highly compressed models can bring major memory and speedup benefits in practice [EDGS20], e.g., up to $10\times$ acceleration in some cases [XZC22]. This makes compression techniques a very promising direction for counteracting the ever-increasing size of modern language models. However, only very little is known about the compressibility of LLMs [DLBZ22]. Does their

massive size make them more compressible, or are they potentially less compressible due to be trained on huge quantities of data?

One major roadblock when it comes to applying compression to LLMs is that existing techniques have been designed to work with models up to around a few 100 million parameters. Consequently, many of these techniques, especially the most accurate ones, are complex and expensive, featuring for example long (re-)training of the models to be made less redundant [EGM+20, PIVA21]. Applying the same set of approaches to networks literally $1000\times$ or more larger is computationally infeasible. This issue is amplified by the fact that the individuals which would benefit the most from compression are precisely those without access to major compute resources. All together, this suggests that the development of compression algorithms targeting specifically extremely large models is a highly relevant research direction. Further, the most useful such methods would not just be accurate, but also affordable enough to be applied with limited resources.

Another significant difference between research on compressing small and compressing large models is that the former is often more theoretical in nature, focusing on highly idealized performance metrics like FLOP counts [KRS+20]. In contrast, LLMs exist, are widely used, and need to be made more efficient *right now*. Thus, it is more important than ever that theoretical compression actually leads to real gains. Achieving this in practice requires solving numerous systems challenges. Foremost among them lies the development of highly efficient low-level GPU kernels, which achieve close to optimal hardware utilization. This is particularly tricky as most current accelerators have been designed and heavily optimized for *uncompressed* model execution. Additionally, other major systems problems arise from the sheer scale of the models that should be compressed.

Lastly, compression is just as relevant for people with massive compute clusters as it is for those with only limited resources. Though, key requirements may differ: the primary concern in this setting is how much a model can be compressed, while the cost of doing so is generally only secondary, at least as long as it can be recouped by the corresponding reduction in the cost of serving millions of users. In this context, highly expensive training-based compression techniques are most promising. However, once compute-intensive training is introduced around LLMs, key questions move from compressing one particular model in isolation to overall scaling behavior across an entire network family: specifically, how compression interacts with model size and the amount of training data. Ideally, this can be captured, similar to standard training, by an approximate formula, a so called *scaling law* [KMH+20, HBM+22], which enables accurate extrapolation from mid- to very-large-scale experiments.

This thesis makes progress in all three major research directions outlined above: algorithms, systems and scaling laws. Overall, it is structured as follows:

- First, Chapter 2 provides an overview about the most important concepts underlying the work in this thesis. This includes topics like the Transformer architecture, compression formats, or GPU programming.

- Next, we introduce a new highly accurate post-training compression algorithm, which also unifies sparsity and quantization, in Chapter 3. This approach is then extended towards GPTQ and SparseGPT, the first algorithms fast and accurate enough to perform successful low-bit quantization and sparsification, respectively, of 100+ billion parameter models.

- Chapter 4 begins by discussing how to write a peak efficiency mixed-precision GPU kernel to unlock the full benefits of GPTQ quantization in practice. Additionally, it also covers how to scale compression techniques to even larger trillion-parameter models and introduces a bespoke extreme sub-1-bit compression format, co-designed with a practical inference kernel.
- Chapter 5 discusses the first scaling law, in the context of Transformers trained on massive datasets, characterizing the relationship between weight-sparsity, non-zero parameter count and amount of training data, eventually leading to the concept of optimal sparsity.
- Finally, all main results are summarized in Chapter 6, followed by an outline of several promising avenues for future work.

CHAPTER 2

Background

This chapter provides an overview of the most important machine learning and systems concepts underlying the work in this thesis. Concretely, it covers Transformer models, neural network compression techniques and accelerator programming. It is intended as a summary of key aspects, while providing references to more complete coverage.

2.1 Transformers

Following their inception in 2017 [VSP+17], models based on the Transformer architecture, or simply *Transformers*, have quickly become the most dominant family of neural network architectures, especially for natural language processing [RWC+19, BMR+20] applications, but also in other domains like computer vision [DBK+21].

2.1.1 Core Architecture

On a high level, a Transformer model $\mathcal{M}$ is a function "transforming" a sequence of T input vectors $\mathbf{x^t}$ into a sequence of T output vectors $\mathbf{y^t}$, usually all of the same dimensionality E. For convenience and practical efficiency, these sequences are generally organized as $\mathbb{R}^{T\times E}$ matrices $\mathbf{X}$ and $\mathbf{Y}$, respectively. The overall transformation is then carried out by sequentially applying L *block* functions $\mathcal{L}_\ell$ to the inputs, i.e. calculating $\mathbf{Y} = \mathcal{L}_L(\mathcal{L}_{L-1}(\dots(\mathcal{L}_1(\mathbf{X}))))$. Typically, all $\mathcal{L}_\ell$ are identical in nature, and are defined as follows (or some minor variation):

$$\mathbf{H} = \mathbf{X} + \text{att}(\text{norm}(\mathbf{X}))\mathbf{W^O} \tag{2.1}$$

$$\mathbf{Y} = \mathbf{H} + \text{act}(\text{norm}(\mathbf{H})\mathbf{W^U})\mathbf{W^D}, \tag{2.2}$$

where $\text{norm}(\cdot)$ denotes a per-vector normalization function, nowadays most commonly root-mean-square normalization [ZS19], $\text{act}(\cdot)$ an element-wise non-linearity like $\text{relu}(x) = \max(0, x)$ and $\text{att}(\cdot)$ the attention function, the heart of the Transformer. The function $\text{att}(\mathbf{X})$ is computed by first calculating *queries* $\mathbf{Q} = \mathbf{X}\mathbf{W^Q}$, *keys* $\mathbf{K} = \mathbf{X}\mathbf{W^K}$ and *values* $\mathbf{V} = \mathbf{X}\mathbf{W^V}$ and then returning the output $\mathbf{O}$ as

$$\mathbf{O} = \text{softmax}\Big(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{E/H}}\Big)\mathbf{V}^\top, \tag{2.3}$$

where softmax$(\cdot)$ is the soft maximum function $[f(\mathbf{z})]_i = e^{z_i} / \sum_{j=1}^{n} e^{z_j}$, applied row-wise. Intuitively, attention combines the values corresponding to input vectors in the sequence for which queries and keys are highly similar. Importantly, this is the only operation in the Transformer in which information can flow across sequence elements—all other sub-functions operate on each (sequence) element independently. An important technical detail is that attention is typically executed with H so called *heads*. This means $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ are first split into segments of consecutive $E_{\text{head}} = E/H$ elements (common values of E_{head} are 64 or 128), then Equation 2.3 is applied to each type of segment separately, the results of which are finally concatenated again, followed by mixing via a multiplication with the *output projection* $\mathbf{W^O}$.

$\mathcal{L}_\ell$, as introduced above, is parametrized by six matrices, $\mathbf{W^Q}, \mathbf{W^K}, \mathbf{W^V}, \mathbf{W^O} \in \mathbb{R}^{E \times E}$, $\mathbf{W^U} \in \mathbb{R}^{E \times F}$ and $\mathbf{W^D} \in \mathbb{R}^{F \times E}$, where F is usually several times larger than E, with a common value being $F = 4E$. $\mathbf{W^U}$ and $\mathbf{W^D}$ are also sometimes respectively called *up-* and *down-projection* as they expand and then contract the main vector dimension E, comprising the *fully-connected* part of a Transformer block. Summing the number of elements in all matrices of all blocks gives a total trainable parameter count of

$$\#\text{parameters}(\mathcal{M}) \approx L \cdot (4E^2 + 2EF). \tag{2.4}$$

Correspondingly, the overall number of necessary floating point operations (FLOPs) for evaluating $\mathcal{M}$ can be approximated as

$$\#\text{FLOPs}(\mathcal{M}) \approx 2 \cdot L \cdot (4E^2T + 2ET^2 + 2EFT), \tag{2.5}$$

where the initial 2 follows from the convention of counting multiply and accumulate as two separate operations. Crucially, it scales with the square of the sequence length T since the attention operation "attends" every token with every other. This gives the Transformer a lot of modelling power, but comes at high computational cost for long sequences.

2.1.2 Embeddings

The previous section introduced the Transformer as a function operating on sequences of (high-dimensional) vectors, but this is generally not a natural representation of real-world data. For instance, to process language data, we first need to map text to vectors and finally vectors back to text (e.g., via [SHB16]). This is typically done by first *tokenizing* input text, that is mapping words, sub-words or common-phrases to some index in a *vocabulary* of predefined size V, followed by selecting the corresponding rows in a $\mathbb{R}^{V \times E}$ input *embedding* matrix. The inverse operation can be performed by first multiplying $\mathbf{Y}$ by an output embedding matrix of shape $\mathbb{R}^{E \times V}$ and then applying row-wise softmax$(\cdot)$ to get a probability distribution over the token vocabulary at each position. Importantly, such embedding matrices do not need to be constructed manually but can simply be learned together with the other Transformer parameters. It is also possible to tokenize other, potentially non-discrete, modalities like images, e.g., via a quantized variational auto-encoder [VDOVK17]. On larger models, embedding matrices typically make up only a small fraction of the Transformer's overall compute and memory cost and can thus often be disregarded to simplify efficiency considerations.

2.1.3 Training

Transformers are commonly trained (or *pretrained*, to be more precise) in *unsupervised* manner, requiring a massive dataset as well as a suitable training objective. Nowadays, the most popular

loss function is *autoregressive next-token-prediction* [RWC+19]. The main idea of this is to make the model predict a probability distribution over the next token for any given sequence prefix. This probability distribution should be as close as possible to the training data, i.e., the probabilty of the next token that is actually observed should be high. Importantly, this means that all target labels are directly given by the text and no further (human) annotation is necessary. To be more precise, let $x_1, x_2, \ldots, x_T$ denote a sequence of tokens, then its corresponding autoregressive prediction loss is defined as

$$\sum_{t=1}^{T} \log P_{\mathcal{M}}(x_t \mid x_{t-1}, x_{t-2}, \ldots, x_1), \tag{2.6}$$

where $P_{\mathcal{M}}(\cdot)$ is the probability assigned to token x_t by the model given the corresponding sequence prefix. The log-sum comes from equivalently transforming the product of all probabilities, the joint probability of the entire sequence, for better numerical behavior during optimization. From an efficiency perspective, it is key that all terms of the above sum can be computed in just a single evaluation of $\mathcal{M}$ by adding an autoregressive mask, $[M]_{ij} = -\infty$ if $i > j$ and 0 otherwise, to the result of $\mathbf{QK}^\top$ in Equation (2.3). This guarantees that attention is only ever computed between each token and its predecessors ($-\infty$ leads to zero post softmax), thus no information contained in future tokens is ever mixed into the next-token-prediction at each position. Exponentiating the average of Equation (2.6) over many tokens and sequences also yields a popular quality measure of a trained model, called *perplexity*. The lower this value, the better the network is at modelling natural text.

Practical training of Transformers involves repeatedly sampling B sequences from the training dataset and packing them together into *batches* (potentially introducing padding to ensure uniform lengths). The model then operates on tensors of shape $B \times T \times E$, fully parallelizing across the first dimension, which typically yields much improved compute utilization. The actual objective optimization happens by continuously moving model parameters into the direction of Equation (2.6)'s gradient, computed via backpropagation [RHW86]. A more detailed description of this process can be found in, for example, [GBC16]. One particularly important aspect of the gradient calculation is that it requires temporarily storing the inputs to (almost) all model layers, in order to efficiently perform backpropagation. For large models, this can quickly lead to huge memory requirements, thus rendering even short training, often called *finetuning*, very expensive. In contrast, when $\mathcal{M}$ is merely evaluated, a layer's input can generally be dropped as soon as its corresponding output is calculated, thus needing much less memory overall. This large discrepancy in storage costs between *forward* passes (model evaluations) and *backward* passes (gradient calculations) is a strong motivation for developing model processing techniques for which just the former suffice.

2.1.4 Inference

Unlike other families of machine learning models, language Transformers are generally used quite differently during *inference*, the process of actually applying existing models, relative to how they are trained. This is because they are *generative* models that can be used for generating new text as follows: first, the model is applied to some starting sequence of length T_{start}, the *prompt*, to produce a probability distribution from which the first response token $x_{T_{\text{start}}+1}$ is sampled. Next, this token is appended to the prompt and the model is run once more on this now length $T_{\text{start}} + 1$ sequence, yielding $x_{T_{\text{start}}+2}$. This process is repeated either until a length limit is hit or the model outputs a special end-of-response token. In this way, a

well trained language Transformer can respond to free-form questions, execute instructions or even solve math problems.

However, a naive execution of this one-at-a-time sampling strategy is hopelessly inefficient. Generating T tokens, for simplicity assuming a prompt of length of 1, would require approximately $(1 + 2 + \cdots + T) \cdot \#\text{FLOPs}(\mathcal{M}) \approx O(T^2 \cdot \#\text{FLOPs}(\mathcal{M}))$ compute. In other words, the runtime of *all layers* scales quadratically in the sequence length T as every token is run through the network again and again for each successive generation. In contrast, during training, all prefixes are known in advance and thus all next-token probabilities can be predicted in parallel, with quadratic scaling *only in the attention* operation. Fortunately, this severe inference bottleneck can be completely resolved with two observations [Sha19]:

1. All intermediate states of a sequence prefix remain identical in all subsequent forward passes due to the causal design of attention (see Section 2.1.3).
2. The state of token j interacts with the states of previous tokens $i < j$ only in the attention operation.

Consequently, it is possible to reuse cached keys and values of earlier tokens for predicting later ones, making the overall compute cost of sequential one-at-a-time generation equivalent to parallel prediction (as used during training). In each forward pass, only the intermediate states for the most recent token must be calculated. It should be noted that the original generation prompt is fully known and can thus be processed in parallel. This operation is often called the *prefill* which populates the initial key- and value-caches (KV-cache for short).

2.1.5 Scaling

Perhaps the key behind the enormous success of Transformer models are their exceptional *scaling* properties [KMH+20, BMR+20]. Training larger and larger networks on more a more data consistently improves their capabilities. Importantly, this trend has been shown to hold over many orders of magnitude, to the point where training models with trillions (that is $> 10^{12}$) of parameters [FZS22, ABG+22] on trillions of tokens [TLI+23, TMS+23] can be justified despite enormous costs. Nowadays, such models produce text very close to human writing [BMR+20] and exhibit strong performance across a very wide ranges of tasks, in some cases even outperforming human experts [AAA+23, RST+24].

Underlying these developments has been the discovery that Transformers do not just keep continuously improving with increased compute, but that they do so in a highly predictable manner [KMH+20, HBM+22]. In fact, it is possible to determine *scaling law* formulas based on medium cost training runs, which relatively accurately predict the capabilities of much larger, and consequently much more expensive, models. Understanding the precise relationships between higher-level properties, like model size or the amount training data used, is not just interesting from a scientific point of view, but also helps to significantly reduce risks of very-large-scale experiments with tens of thousands of accelerators, costing hundreds of millions of dollars.

2.2 Model Compression

Improving the efficiency of neural networks is a research direction with a long history [LDS89, HSW93]. While the focus of this area used to lie on making networks suitable for real-time

or mobile settings, similar techniques are nowadays critical to make massive state-of-the-art models run at all on anything that is not the very top-end hardware [DLBZ22], as well as for improving the economics of serving such models to huge user-bases.

One particularly successful area of efficiency research is *model compression*. Trained networks generally do not make optimal use of every model parameter. Thus, it is often possible to significantly reduce a model's resource consumption by removing redundancies at no, or only very little, impact on the function/accuracy. In other words, the model representation is *compressed*. This can be accomplished in various different ways, ranging from structural reshaping of the model architecture to fine-grained changes in the bit-representation of individual parameters. This thesis focuses on two forms of compression that have been shown to be highly practical [HABN+21, GKD+21]: *sparsity* and *quantization*.

2.2.1 Sparsity

The main idea behind sparsity is to set as many parameters as possible in the (large) weight matrices of neural networks to *exactly zero*. Such weights can be considered "deleted" and may be skipped during computation, as a multiplication with zero always results in zero. At the same time, this can also reduce memory costs, for instance, by storing only non-zero values and their corresponding matrix indices. This form of compression is motivated by the empirical observation that many weight values tend to have very small magnitude and their complete removal thus has only minimal effect on a model's output.

More formally, given a model with parameters $\mathbf{w} \in \mathbb{R}^d$, the goal of sparsification, or *pruning*, is to find a new set of weights $\mathbf{w}'(\mathbf{w}) \in \mathbb{R}^d$ together with a sparsity mask $\mathbf{m} \in \{0,1\}^d$, where a 1 indicates that the corresponding weight w'_i is non-zero. $\mathbf{w}'$ and $\mathbf{m}$ should minimize the task loss $\mathcal{L}(\cdot)$, while achieving sparsity S:

$$\text{argmin}_{\mathbf{w}'(\mathbf{w}),\mathbf{m}}\, \mathcal{L}(\mathbf{w}'(\mathbf{w}) \odot \mathbf{m}) \quad \text{s.t.} \quad 1 - \sum_{i=1}^{d} m_i \geq S. \tag{2.7}$$

Crucially, the post-sparsification weights $\mathbf{w}'(\mathbf{w})$ may be derived from the original ones, but can also differ significantly if this is beneficial to compensate for accuracy drops caused by the removal of a large fraction of parameters. In some scenarios [EGM+20], an initial set of weights $\mathbf{w}$ is not available, meaning that $\mathbf{w}'$ and $\mathbf{m}$ must be found from scratch, potentially rendering the problem even more difficult.

Sparsity can be applied at differing levels of granularity. On the one extreme lies completely *unstructured* removal, where weights may be deleted in whatever pattern is most favorable to preserve accuracy. On the other, there is *structured* pruning, where large groups of weights, for instance entire rows or columns, must either be removed completely or not at all. Unstructured pruning is ideal from an accuracy standpoint due to maximum flexibility, at the disadvantage of making practical acceleration quite challenging because of the highly irregular weight access patterns caused by essentially random sparsity masks. Nevertheless, it is possible to achieve substantial, although usually less than ideal, speedups on CPUs [KKG+20, EDGS20] or older GPUs [GZYE20] in practice, via clever algorithms and systems engineering. In contrast, structured pruning is usually trivial to accelerate with close to ideal speedups, even on dedicated neural network accelerators, but maintaining accuracy is challenging, as fully deleting large structures is often quite destructive to a model's function. More recently, *semi-structured* formats, which trade off imposing some structure for significantly improved acceleration capabilities, are gaining in popularity. The most prominent such format is n:m

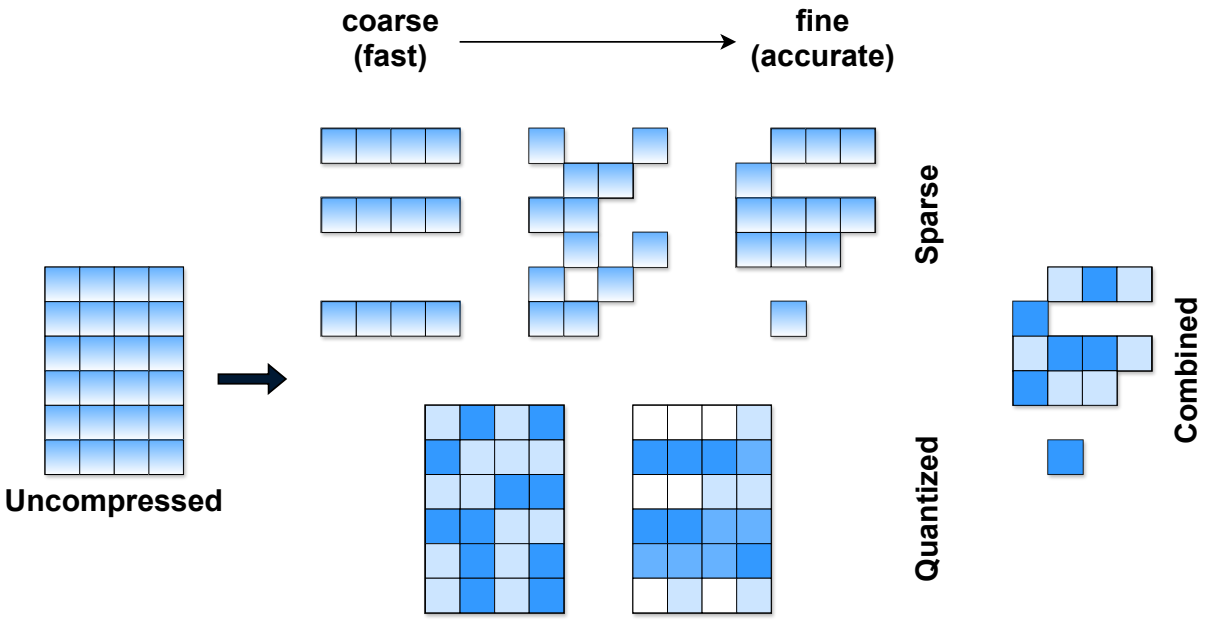


Figure 2.1: Compression at different levels of granularity. (top) structured, semi-structured (2:4) and unstructured sparsity. (bottom) per-matrix and per-row quantization.

sparsity [MLP+21, PY21], a pattern where each m consecutive weights contain exactly n non-zero values. Its 2:4 variant has dedicated hardware support on modern NVIDIA GPUs, allowing up to $2\times$ speedup relative to state-of-the-art dense matrix multiplies. Figure 2.1 (top) illustrates different types of sparsity.

2.2.2 Quantization

Quantization, meaning to reduce the numerical precision of storage and/or computation, is probably the most popular type of model compression in practice. More precisely, a weight matrix is considered to be quantized if each element is from a (small) discrete set of possible values $\mathcal{Q}$.

In principle, $\mathcal{Q}$ can have arbitrary structure. However, for practical purposes, it is common to pick a *bit-width* b and fix $\mathcal{Q} = \{0, 1, \dots, 2^b - 1\}$. This unsigned grid of 2^b integer values can be moved closer to a given weight distribution via a linear transformation $q' = sq - z$ involving a *scale* s and a *zero-point* z. Crucially, given two linearly quantized matrices $s_1\mathbf{Q_1} - z_1$ and $s_2\mathbf{Q_2} - z_2$ where $\mathbf{Q_1}, \mathbf{Q_2} \in \mathcal{Q}^{d\times d}$, multiplication can be rearranged as follows:

$$(s_1\mathbf{Q_1} - z_1)(s_2\mathbf{Q_2} - z_2) = (s_1 s_2)\mathbf{Q_1}\mathbf{Q_2} - (z_1 s_2)\mathbf{1}\mathbf{Q_2} - (z_2 s_1)\mathbf{1}\mathbf{Q_1} - z_1 z_2, \tag{2.8}$$

where $\mathbf{1}$ denotes a d-dimensional vector of all ones and scalars are assumed to be broadcasted onto relevant matrices. This means, the computationally dominating part of this calculation, the $O(d^3)$ matrix multiply $\mathbf{Q_1}\mathbf{Q_2}$, can be fully executed at lower precision, which can be well accelerated on hardware. Notably, this trick relies on the linearity of the quantization grid. A simple strategy of deriving s and z for some set of weights is via minimum and maximum observed values. Finding the closest $s\mathbf{q} - z$ to some weight vector $\mathbf{w}$, can be accomplished by simple nearest rounding. Putting everything together, quantizing weights $\mathbf{w} \in \mathbb{R}^d$ can be accomplished via (assuming $\mathbf{w}$ contains at least one positive and one negative value):

$$s = \frac{\max(\mathbf{w}) - \min(\mathbf{w})}{2^b - 1}, \quad z = -\min(\mathbf{w}), \quad \mathbf{q} = s \cdot \operatorname{round}\left(\frac{\mathbf{w} + z}{s}\right) - z. \tag{2.9}$$

Since s and z are constant between many weights, they can be stored separately in standard precision, while each quantized weight value is exactly a b-bit number, thus reducing overall memory (and potentially compute) costs by essentially the decrease in bit-width. Nowadays, weights are almost always stored in 16-bit precision per default, hence quantizing to, e.g., 4-bit yields a $4\times$ reduction in size.

Similar to sparsity (see Section 2.2.1), quantization can also be applied at varying levels of granularity. Instead of sharing s and z over full weight matrices, one can maintain scales and zeros for each layer output, or even more fine-grained, for each group of g consecutive weights. As long as g is not too small, this approach still achieves very good compression: for instance, at 4-bit with groupsize 128, $4d$ bits are required to store the quantized weights and $(16 + 16)d/128$ bits to store all scales and zeros at 16-bit precision, averaging out to 4.25 bits per parameter. While fine-grained quantization also causes some computational overhead, it is generally nowhere near as problematic as fully unstructured sparsity, since enough regularity in the representation remains. Although, there is one critical detail: groups must be formed across the reduction axis of the matrix multiply; otherwise, the technique in Equation 2.8 is not applicable. Finally, it should be noted that achieving computational speedups through quantization on current hardware requires both matmul operands (weights *and* activations) to be quantized. This is unlike sparsity, where compressing just the weight matrices suffices, in principle. Nevertheless, in specific instances, as this thesis will show, there can also be major performance benefits from quantizing *just* the weights.

2.2.3 Training-Based vs. Post-Training

Compression methods can be broadly categorized into *training-based* [EGM+20, SA20] and *post-training* [HCI+21, NAVB+20], or also sometimes called *one-shot* methods. The former involve significant amounts training or re-training: for example, gradual magnitude pruning (GMP) [ZG17] periodically removes a small fraction of weights with lowest absolute value, while training or finetuning a model. Importantly, a training-based sparsification approach must not necessarily start from a randomly initialized model like [PIVA21], but may also be bootstrapped with an existing checkpoint [SA20]. In contrast, *post-training* techniques are characterized by compressing a pretrained model in a single step, without any kind of retraining. While *no-training* methods may be a better name for this category, we stick to the by now rather established, albeit slightly confusing, term *post-training*.

Generally, training-based methods are most powerful, producing the most accurate models at the highest compression rates. However, they are often also quite expensive, with the best results achieved only with considerably extended training times [EGM+20, PIVA21] and after careful hyper-parameter tuning [KA22]. Meanwhile, post-training methods are usually not nearly as accurate, but therefore very cheap and robust. They can often achieve decent compression rates, essentially "at-the-push-of-a-button", with minimal compute resources. This has made them quite popular recently, especially in the context of inference engines or serving platforms, where existing models should be accelerated but only little compute and/or training data is available for compression. Nevertheless, training-based methods are currently most widespread.

2.2.4 Large Models

Although model compression is a well studied area, essentially all existing research has focused on neural networks up to at most a few hundred million parameters, with a trend towards

developing increasingly complex and expensive methods for pushing compression rates further and further. At the same time, some of the most exciting new models, with breakthrough language understanding capabilities, are becoming extremely large in size, up to hundreds of billions [BMR+20] or even trillions of parameters [FZS22, ABG+22]. While such models used to be available only to a small handful of institutions, recent efforts [ZRG+22, SFA+22] have released similarly large networks to the wider research and practitioner community. As those 100+ billion parameter models are really unwieldy to run on all but the very top end hardware, significantly compressing them seems particularly appealing. Unfortunately, this is a challenging endeavor.

First of all, training-based methods are essentially infeasible to run on those models for most people, and even those select few in possession of massive compute clusters may not want to risk expensive compression experiments that eventually fail due to finicky hyper-parameter tuning. This moves the spotlight onto the previously rather niche post-training approaches. These are generally orders of magnitude cheaper to run, making them seem a significantly more promising pathway towards affordable compression of super large models. However, simple and trivially scalable approaches [DLBZ22] may not preserve accuracy at significant compression, while existing accurate techniques designed for 100 million parameter models [NAVB+20, LGT+21] may not be fast enough to run at literally $1000\times$ larger scale.

Enormous parameter count is not the only challenge brought by modern language models. Most standard pruning benchmarks like ResNet50/ImageNet [ZG17] or BERT/SQuAD [SWR20] feature datasets with tens of thousands to a few million samples, which are all seen many times during training. By contrast, LLMs are typically trained on many orders of magnitude more data, so much, that processing every token just once suffices for good training [BMR+20]. It is unclear, whether or how this affects any prior observations on model compressibility. Are such models harder to compress as they need to memorize more data, or are they actually easier to compress because they are much larger and thus even more overparameterized?

2.3 Hardware Accelerators

Perhaps the main driver behind deep learning progress in general [KSH12], and large language models in particular, has been advances in computing hardware, specialized to precisely the operations most critical for machine learning. Dedicated *accelerators* like (NVIDIA) GPUs or TPUs [JYP+17] are orders of magnitude faster and more energy efficient at executing relevant training and inference workloads, relative to general purpose CPUs. The latter are designed for mostly sequential execution with potentially many branching decisions, while the former are specialized to massively parallelizable tasks, such as matrix-matrix multiplications, the workhorse of modern neural networks.

This section introduces the key components of NVIDIA GPUs, the most widely used type of ML accelerator, from a software perspective. An NVIDIA GPU is programmed in the C++ dialect CUDA, where the part of the code which is actually run on the accelerator is usually referred to as a *kernel*. Developing kernels differs from regular software engineering by being very low-level, often taking into account aspects like cache levels or register assignments, and generally prioritizing performance above all else.

2.3.1 Parallelization Hierarchy

On a high-level, a GPU is designed to run potentially tens of thousands of *threads* in parallel. This can only be achieved by making individual threads relatively weak and adhering to various parallelization restrictions and trade-offs, eventually leading to a hierarchical parallelization structure.

On the highest level, most GPUs have between 50 to 100 *Streaming Multiprocessors (SMs)*, essentially separate computing cores. They can run fully independently, which is also why communication between them is quite expensive. On each SM, there may reside up to between $32 - 64$ so called *warps*, that is groups of exactly 32 threads each. In a single cycle, all threads in a warp must always execute the same instruction (but potentially on different data). If some threads in a warp do not participate in a particular instruction (e.g., by a branching decision), they must necessarily idle in this cycle and cannot compute anything else. Shuffling around data within a warp using dedicated instructions is by far the fastest form of communication on a GPU. While an SM can, in principle, run dozens of warps "in parallel", it can only issue instructions of at most a small handful of warps in each cycle. However, GPUs follow an asynchronous computation model. If one warp is reading data from global memory, a time-consuming operation, other warps may perform actual computation. Further, even the same warp that is reading data may continue executing operations that do not require the result of the initial data read; only once such a dependency is hit, the warp must stall. Almost all instructions have a certain latency of at least a few cycles. To hide that, computation is realized via pipelines to which warps submit instructions, receiving the results once they are actually executed. There are also different pipelines (integer arithmetic, floating point arithmetic, data loading, etc.) which may be engaged simultaneously, provided that they are fed with sufficiently many instructions by different warps.

Kernel programs are organized as *thread blocks*, that is, grid assignments which partition a problem across SMs, warps and threads. How efficiently a GPU can solve a task is heavily dependent on how well it can be mapped onto the existing parallelization hierarchy. Ideally, there will be no communication between SMs, some communication between warps and the most communication within a warp. A perfect example of a well supported operation is a matrix-matrix multiply: each element of the output is fully independent of one another.

2.3.2 Memory Hierarchy

A problem being well parallelizable across a huge number of threads is a necessary but not a sufficient condition for being well suited to GPU computation. Another critical factor is given by memory access patterns. Modern GPUs have such high compute throughput that they can be easily bottlenecked by memory reads, if those cannot properly utilize the memory hierarchy.

First, to use a GPU, all data must be transferred from CPU system memory (RAM) to GPU main memory (DRAM). This is very slow and initiating such transfers *during* a GPU computation should only be a last resort workaround if it is impossible to store all data in DRAM. While CPU servers may easily have a few TBs of RAM, even the most expensive top-end GPUs have less than 100 GB of DRAM, which is why efficiently running very large models often requires splitting weights across several GPUs. Although GPU DRAM is at least an order of magnitude faster than CPU RAM, reading from this *global memory* is still very slow compared to any arithmetic operations and should be minimized as much as possible. In addition to the global DRAM, every SM also has a few MBs of *shared memory*, located in SRAM, which is more than $10\times$ faster compared to DRAM. Every warp can access exactly

the shared memory of the SM that it lives on; warp-to-warp communication is also generally realized via shared memory. Finally, like on other computing hardware, the actual computation happens within *registers* that bring no extra memory access costs. Compared to CPUs, GPUs have a very large number of registers (up to 65K per SM) to support very large thread counts. Peak efficiency kernels often utilize almost all of the available registers to avoid much slower shared memory (or worse, global) reads and writes as much as possible.

In addition to explicitly controlled global and shared memory, GPUs also feature L1 and L2 caches. The former is located in SRAM and thus behaves exactly like shared memory, although not explicitly managed. The latter is much larger (10s of MBs) and shared between all SMs, while being $2-4\times$ faster than global memory. To maximize global memory bandwidth, it is critical that threads access consecutive memory locations, as this can often be *coalesced* into a single access. Further, each thread may read data vectors of up to 16 bytes in a single transaction. In contrast, shared memory is organized into 32 *banks*, requiring each thread in a warp to read from a different bank for maximum efficiency. This happens for consecutive locations, but can, with some care, also be achieved for other access patterns. Finally, it should be noted again that memory access and compute can and should be overlapped (see also Section 2.3.2), which sometimes requires explicit software pipelining or register buffering.

2.3.3 Tensor Cores

Graphics Processing Units (GPUs) were, as the name suggests, originally created for graphics applications. While their massively parallel design is very well suited for matrix multiplications, significant further gains can be made with hardware features designed for this particular operation. The introduction of *tensor cores*, dedicated units for accelerating matmuls, has probably brought the biggest leap in machine learning compute in recent years.

A tensor core is a hardware circuit which is able to execute $\mathbf{D} = \mathbf{AB} + \mathbf{C}$ for small matrices (shapes differ between GPU generations and data types) in a single cycle. For example, on Ampere-class devices, tensor cores can multiply a 16×16 and a 16×8 matrix in a single cycle, that is $16 \cdot 16 \cdot 8 = 2048$ FP16 multiply-accumulates. The input and output matrices are held in registers distributed across all threads in a warp, hence a tensor core operation always involves a complete warp. A very useful feature of tensor cores is that the *accumulators* $\mathbf{D}$ and $\mathbf{C}$ can have higher precision (e.g., FP32) than the operands $\mathbf{A}$ and $\mathbf{B}$ (e.g., FP16), at no performance penalty. This is relevant to avoid numerical issues in large matmuls. One downside of tensor cores is that they are so fast that any other additional operations, even sometimes index calculations, can easily slow down throughput if they cannot be perfectly overlapped. In the context of compression, this makes it challenging to utilize tensor cores well in many formats, which is however a necessity to remain competitive in terms of speed with standard uncompressed execution.

CHAPTER 3

Algorithms

This chapter introduces several new post-training compression algorithms, in particular ones specifically targeting extremely large models. It begins by first developing a highly accurate layer-wise compression approach which unifies quantization and pruning in Section 3.1. This algorithm, initially developed for small models of $\approx$ 100 million parameters, is then adapted and optimized in major ways to eventually yield GPTQ and SparseGPT, the first methods that are fast and accurate enough to perform low-bit compression and nontrivial sparsification of models up to 100+ billion parameters, respectively.

Section 3.1 is based on the NeurIPS 2022 paper "Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning" [FSA22], Section 3.2 on the ICLR 2023 paper "OPTQ: Accurate Post-Training Quanitzation for Generative Pretrained Transformers" [FAHA23], and Section 3.3 on the ICML 2023 paper "SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot" [FA23].

3.1 Optimal Brain Compression: Accurate Post-Training Quantization and Pruning

3.1.1 Motivation & Overview

Motivated by the growing parameter counts and computational costs of state-of-the-art ML models, hundreds of pruning and quantization approaches have been proposed and analyzed [HABN$^+$21, GKD$^+$21], with the general goal of obtaining efficient variants of deep neural nets (DNNs) which would preserve accuracy while maximizing compression. Despite impressive progress, compression is still a laborious process: the pruning and quantization stages are often done independently, and recovering model accuracy after compression often requires partial or even full retraining of the compressed model.

An alternative but challenging scenario is the *post-training compression* setup [NAVB$^+$20, LGT$^+$21, HNH$^+$21, LGW$^+$21], in which we are given a trained but uncompressed model, together with a small amount of *calibration data*, and must produce an accurate compressed model in *one shot*, i.e. a single compression step, without retraining, and with limited computational costs. This is motivated by practical scenarios such as the MLPerf Inference Benchmark [RCK$^+$20], and is the setting we focus on in this chapter.

Compression via *weight pruning* started with seminal work by LeCun et al. [LDS89], complemented by Hassibi and Stork [HSW93], who proposed a mathematical framework called the *Optimal Brain Surgeon (OBS)*, for choosing the set of weights to remove from a trained neural network, by leveraging second-order information. Recent advances, e.g. [DCP17, WGFZ19, SA20, FKA21] showed that OBS can lead to state-of-the-art compression at DNN scale, by introducing numerical methods which can approximate the second-order information needed by OBS at the massive parameter counts of modern models. However, these approaches do not apply to the *post-training* setting, as they require gradual pruning, as well as significant retraining, to recover good accuracy.

An alternative approach, which is standard in the context of *post-training* compression, has been to break the compression task into layer-wise sub-problems, identifying a compressed weight approximation for each layer, given a sub-sample of the layer's inputs and outputs based on calibration data. This line of work, e.g. [WCHC20, NAVB+20, HNH+21], introduced elegant solvers for the resulting layer-wise weight quantization problem, which achieve state-of-the-art results for post-training quantization. Recently, AdaPrune [HCI+21] showed that this approach can also be effective for post-training weight pruning.

In this context, a natural question is whether existing approaches for pruning and quantization can be *unified* in order to cover both types of compression in the post-training setting, thus making DNN compression simpler and, hopefully, more accurate. This question is also of practical importance, since both GPU and CPU platforms now *jointly* support sparse and quantized formats [MLP+21, Neu22], and, as we illustrate experimentally, the resulting models could be executed with compound speedups.

Contribution. In this work, we provide a mathematical framework for compression via pruning or quantization, which leads to state-of-the-art accuracy-versus-compression trade-offs in the challenging *post-training compression* setup. Our framework starts from the layer-wise compression problem described above, by which the global compression task, defined either for pruning or quantization, is first split into layer-wise sub-problems, based on the layer behavior on the calibration data. Specifically, given a layer ℓ defined by weights $\mathbf{W}_\ell$, and layer inputs $\mathbf{X}_\ell$, we aim to find a compressed version of the weights $\widehat{\mathbf{W}}_\ell$ which minimizes the output difference relative to the uncompressed layer, measured via the squared error between the original and compressed layer, acting on the sample input $||\mathbf{W}_\ell\mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell\mathbf{X}_\ell||_2^2$, under a fixed compression constraint on $\widehat{\mathbf{W}}_\ell$.

Although solving this problem optimally for sparsity or quantization constraints is NP-hard [BD08, NAVB+20], it is a key step in all state-of-the-art post-training compression methods, both for pruning [HCI+21, FA22] and for quantization [NAVB+20, HCI+21, LGT+21]. Once this is solved per layer, a solution to the global problem can be obtained by combining layer-wise solutions, which is handy especially for non-uniform compression, e.g. [HLL+18, FA22]. Thus, several approximations for this problem have been proposed [NAVB+20, HNH+21, HCI+21].

We show that there still is significant room for improvement when solving the layer-wise compression problem. Roughly, our approach is to specialize the OBS framework to the squared error formulation above: in this case, the framework can in theory produce an exact greedy solution, but a direct implementation would have infeasible $\Theta(d^4)$ computational cost, where d is the layer dimension. Our main technical contribution is a series of algorithms which reduce this computational cost, *without any approximations*, to $O(d \cdot d_{col}^2)$ where d_{col} is the column dimension of the weight matrix. In practice, these improvements are significant

enough to allow us to implement the exact OBS greedy solution, which prunes *one weight at a time*, and updates *all remaining weights* after each step, at the scale of modern DNNs with tens of millions of parameters, within reasonable time, on a single GPU. We provide efficient implementations of our methods at `https://github.com/IST-DASLab/OBC`.

In turn, this algorithmic development allows us to apply the OBS approach to *quantization*. The resulting algorithm, called the *Optimal Brain Quantizer (OBQ)*, quantizes weights iteratively one-at-a-time, depending on their impact on the loss increase, after which it applies a closed-form update to the remaining unquantized weights, further reducing the loss. This solves the two problems efficiently, and in a unified manner—we call the unified framework the *Optimal Brain Compressor (OBC)*.

Experimental Results. We apply OBC to standard tasks and models covering image classification, object detection, and language modelling applications. We first show that our framework yields significantly better solutions for the layer-wise compression problem, which leads to higher-accuracy end-to-end compressed models for both pruning and quantization, relative to the corresponding state-of-the-art techniques, often by significant margins. Second, we show that our pruning and quantization approaches can be compounded, with surprisingly strong results: we obtain a $12\times$ reduction in theoretical operations with a 2% accuracy drop for GPU-supported compound compression [MLP+21], and a $4\times$ speedup in *actual runtime* with only 1% accuracy drop for a CPU-based sparsity-aware runtime [Neu22]. Together, these results suggest for the first time that post-training compression can be competitive with full retraining.

3.1.2 Related Work

Optimal Brain Surgeon (OBS). The classic OBS framework [LDS89, HSW93] was originally applied to networks with hundreds of weights; more recently, methods such as WoodFisher [SA20] rendered the approach computationally feasible for DNNs by using a block-diagonal Fisher approximation of the Hessian, while follow-up methods introduced more efficient and general algorithms for handling the inverse Fisher matrix [FKA21], or customize this approximation to specific model families [KCN+22]. Earlier work called Layer-wise OBS (L-OBS) [DCP17] was inspired by the K-FAC approximation [MG15, GM16]: L-OBS approximates the OBS framework not for the global objective, but for a quadratic per-layer loss, while also pruning all weights based on a single Hessian computation. At a high level, our approach is similar, in that we apply OBS layer-wise; however, we apply OBS *exactly*, that is, pruning one weight at a time, and exactly recomputing the Hessian after every pruning step. This is made computationally tractable by several new algorithmic ideas, and yields significantly improved results relative to L-OBS. This prior work on pruning considered settings with extensive finetuning. By contrast, we will focus on the post-training setting, where only a small amount of calibration data is available.

Post-Training Quantization. This setting has been primarily considered for quantization, and most state-of-the-art methods work by performing layer-wise compression. Specifically, BitSplit [DCP17] optimizes the quantized weights bit by bit, while AdaRound [NAVB+20] finds a weight rounding policy through gradient based optimization with an annealed penalty term that encourages weights to move towards points on the quantization grid. AdaQuant [HNH+21] relaxes the AdaRound constraint, allowing weights to change during quantization-aware optimization, via straight-through estimation [NFA+21]. BRECQ [LGT+21] suggested

that accuracy can be improved further by integrating second-order information into the layer-wise losses and by jointly optimizing hand-crafted blocks of related layers.

A key step of AdaRound, AdaQuant and BRECQ is to quantize layers incrementally, in *sequential* order, so that errors accumulated in earlier layers can be compensated by weight adjustments in later ones. This significantly improves performance, but reduces flexibility, as the entire process may need to be re-done whenever we wish to change compression parameters of one layer. We instead target *independent* compression of each layer, allowing the end model to be simply "stitched" together from layer-wise results. Despite operating independently on each layer, we find that, after correcting basic statistics such as batchnorm, our method performs on par to sequential ones for uniform quantization.

Post-Training Sparsification. The layer-wise approach was shown to also be effective for post-training pruning by AdaPrune [HCI+21], which pruned weights to the GPU-supported N:M pattern [ZMZ+21]. AdaPrune first drops parameters according to their magnitude [ZG17] and then reoptimizes the remaining weights to reconstruct the pre-compression calibration set output. This is similar to [HZS17, ELRCB18] which also perform layer-wise reoptimization of the remaining weights. Follow-up work [FA22] noted that the results of AdaPrune can be improved further by performing more frequent pruning/optimization steps. Our algorithm pushes this idea to the limit, performing *full reoptimization* after every single pruned weight, while remaining computationally tractable. We further use a more sophisticated weight selection metric which incorporates second-order information. Finally, [FA22] also introduces *global AdaPrune*, a more expensive global optimization step applied on top of the layer-wise AdaPrune results, which can bring additional accuracy gains. This can also be applied to our pruned models.

Non-Uniform Compression. An orthogonal practical question is how to compress different layers to maximize accuracy under a given resource constraint, such as latency or energy consumption. Existing methods can be roughly categorized into search-based and solver-based approaches. The former, e.g. AMC [HLL+18] or HAQ [WLL+19], search for a layer-wise compression policy directly via, for example, reinforcement learning or genetic programming [YGZL20], whereas the latter, e.g. HAWQv3 [YDZ+21] or AdaQuant [HNH+21], construct a relaxed version of the overall problem that is then solved exactly. We focus here on solver-based approaches, as they can rapidly adapt to different scenarios when combined with accurate independent layer-wise compression schemes; however, our techniques could be of interest for search-based methods as well. Concretely, we use the problem formulation of AdaQuant [HNH+21] to which we apply the DP algorithm of SPDY [FA22] to achieve fast solving times even with a large number of possible choices per layer.

3.1.3 Problem Definition and Background

The Layerwise Compression Problem. Following prior work on post-training compression, e.g. [NAVB+20, HNH+21], we define the problem as follows. Mathematically, we model a layer ℓ as a function $f_\ell(X_\ell, W_\ell)$ acting on inputs X_ℓ, parametrized by weights W_ℓ. The goal of layer-wise compression is to find a "compressed" version of W_ℓ that performs as similarly as possible to the original weights. More formally, the compressed weights $\widehat{W}_\ell$ should minimize the expected layer output change as measured by some loss $\mathcal{L}$ while at the same time satisfying a generic compression constraint, which we denote by $\mathcal{C}(\widehat{W}_\ell) > C$, which will be customized

depending on the compression type:

$$\text{argmin}_{\widehat{W}_\ell} \quad \mathbb{E}_{X_\ell} \mathcal{L}(f_\ell(X_\ell, W_\ell), f_\ell(X_\ell, \widehat{W}_\ell)) \quad \text{subject to} \quad \mathcal{C}(\widehat{W}_\ell) > C. \tag{3.1}$$

The expectation over the layer inputs X_ℓ is usually approximated by taking the mean over a small set of N input samples. This low-data setting is one of the primary applications of layer-wise compression. Further, most works [WCHC20, NAVB+20, HNH+21] focus on compressing linear and convolutional layers, which can be unfolded into linear ones, as these are prevalent in practice, and use the squared loss to measure the approximation error. This definition of the loss can be motivated, via a sequence of approximations, from second-order information: please see [NAVB+20] for a precise derivation. Furthermore, this approximation approach has been shown to work well in many applications [NAVB+20, HNH+21, FA22].

We follow these conventions as well, and work with the specific layer-wise compression problem stated formally below, where the weights $\mathbf{W}_\ell$ are a $d_{\text{row}} \times d_{\text{col}}$ matrix (for conv-layers d_{col} corresponds to the total number of weights in a single filter), and the input $\mathbf{X}_\ell$ has dimensions $d_{\text{col}} \times N$.

$$\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2 \quad \text{s.t.} \quad \mathcal{C}(\widehat{\mathbf{W}}_\ell) > C. \tag{3.2}$$

The Optimal Brain Surgeon (OBS) Framework. The OBS framework [LDS89, HSW93] considers the problem of accurately pruning a trained dense neural network. It starts from the Taylor approximation at the given point (assumed to have negligible gradient), and provides explicit formulas for the optimal single weight to remove, as well as the optimal update of the remaining weights which would compensate for the removal. More precisely, let $\mathbf{H}$ denote the Hessian matrix of the loss at the given (dense) model. Then the weight to prune w_p which incurs the minimal increase in loss and the corresponding update of the remaining weights $\boldsymbol{\delta_p}$ can be calculated as follows:

$$w_p = \text{argmin}_{w_p} \frac{w_p^2}{[\mathbf{H}^{-1}]_{pp}}, \quad \boldsymbol{\delta_p} = -\frac{w_p}{[\mathbf{H}^{-1}]_{pp}} \cdot \mathbf{H}^{-1}_{:,p}, \tag{3.3}$$

where $[\mathbf{H}^{-1}]_{pp}$ denotes the pth diagonal entry of the inverse Hessian, and $\mathbf{H}^{-1}_{:,p}$ is its pth column.

OBS for Layer-Wise Pruning. We will now instantiate this framework for the layer-wise pruning problem, defined above. First, the loss in equation (3.2) is quadratic and since our starting point is given by the dense weights achieving the minimal loss of 0, the assumptions of the OBS framework are fully met, meaning that its formulas are *exact* for this specific problem formulation. Thus, iterating the OBS framework to remove one weight at a time would yield an exact *greedy solution* for the layer-wise pruning problem, as it takes the (locally) optimal decision at each step. While this greedy approach does not guarantee convergence to a global optimum, such approaches can be very effective for dealing with problem instances that are too large to be handled by exact methods.

3.1.4 An Optimal Greedy Solver for Sparsity

The obvious challenge is that applying the OBS framework in its true form, i.e. pruning a single weight at a time using the exact formulas in (3.3), is computationally very demanding. The Hessian $\mathbf{H}$ is a $d \times d$ matrix where $d = d_{\text{row}} \cdot d_{\text{col}}$, which is already expensive to store and compute with. Additionally, this matrix needs to be updated and inverted at each of the

$O(d)$ steps with a computational complexity of $\Theta(d^3)$. Clearly, an $O(d^4)$ total runtime is too inefficient for pruning most layers of modern neural networks, as d is usually $\geq 10^5$ or even $\geq 10^6$ for several layers. However, as we will now show, it is actually possible to reduce the overall costs of this process to $O(d_{\text{row}} \cdot d_{\text{col}}^3)$ time and $\Theta(d_{\text{col}}^2)$ memory, making it efficient enough to prune, e.g., all layers of a medium-sized model such as ResNet50 in a bit more than one hour on a single NVIDIA RTX 3090 GPU. We emphasize that the techniques we introduce are exact; unlike prior work [DCP17, SA20], we do not rely on any approximations.

The ExactOBS Algorithm. In the following, we introduce our efficient instantiation of the OBS framework, for the layer-wise compression problem, which we call ExactOBS, in step-by-step fashion. We start by rewriting the matrix squared error in (3.2) as the sum of the squared errors for each row in the weight matrix. As we are always dealing with a fixed layer ℓ, we drop the subscript ℓ to simplify notation. The objective is then equivalent to $\sum_{i=1}^{d_{\text{row}}} ||\mathbf{W}_{\mathbf{i},:}\mathbf{X} - \widehat{\mathbf{W}}_{\mathbf{i},:}\mathbf{X}||_2^2$.

This way of writing the error makes it clear that removing a single weight $[\mathbf{W}]_{ij}$ only affects the error of the corresponding output row $\mathbf{Y}_{\mathbf{i},:} = \mathbf{W}_{\mathbf{i},:}\mathbf{X}$. Hence, there is no Hessian interaction between different rows and so it suffices to work only with the individual $d_{\text{col}} \times d_{\text{col}}$ Hessians corresponding to each of the d_{row} rows. Further, as the dense layer output $\mathbf{Y} = \mathbf{W}\mathbf{X}$ is fixed, the objective for each row has standard least squares form and its Hessian is given by $\mathbf{H} = \mathbf{2}\mathbf{X}\mathbf{X}^\top$.

Although this observation already reduces computational complexity, two key challenges remain: (a) applying OBS to each row still costs $O(d_{\text{col}} \cdot d_{\text{col}}^3)$ time, which is too slow for large layers, and (b) we need fast access to the Hessian inverses of all d_{row} rows, since we want to prune the minimum score weight of the whole matrix rather than just per row in each step. In particular, (b) requires $O(d_{\text{row}} \cdot d_{\text{col}}^2)$ GPU memory, which is likely to be infeasible.

Step 1: Handling a Single Row. We first describe how to efficiently prune weights from a single row with d_{col} parameters. For simplicity, we denote such a row by $\mathbf{w}$ with corresponding Hessian $\mathbf{H}$. The full algorithm for this procedure is given in Algorithm 3.1; in the following, we provide a detailed description. The key idea is to avoid having to do the full $\Theta(N \cdot d_{\text{col}}^2)$ calculation and $\Theta(d_{\text{col}}^3)$ inversion of $\mathbf{H}$ in each step. The former is easy, as the weights themselves do not enter the calculation of $\mathbf{H} = 2\mathbf{X}\mathbf{X}^\top$, and the Hessian for the weights with pruning mask M denoted by $\mathbf{H}_M$ is thus simply comprised of the corresponding rows and columns in the fully dense version $\mathbf{H}$. Hence, we only have to compute $\mathbf{H}$ (which is actually the same for all rows) once, from which we can then extract the rows and columns corresponding to M as needed.

Critically, this trick is *not* applicable to the inverse, as $(\mathbf{H}_M)^{-1} \neq (\mathbf{H}^{-1})_M$. However, using the fact that the removal of one parameter p simply drops the corresponding row and column from $\mathbf{H}$, we can actually update the inverse to remove parameter p directly using a single step of Gaussian elimination, with cost $\Theta(d_{col}^2)$. The following result, whose proof is in Appendix A.1.1, formalizes this.

Lemma 1 (Row & Column Removal)**.** *Given an invertible $d_{col} \times d_{col}$ matrix $\mathbf{H}$ and its inverse $\mathbf{H}^{-1}$, we want to efficiently compute the inverse of $\mathbf{H}$ with row and column p removed, which we denote by $\mathbf{H}_{-p}$. This can be accomplished through the following formula:*

$$\mathbf{H}_{-p}^{-1} = \left(\mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{pp}}\mathbf{H}_{:,p}^{-1}\mathbf{H}_{p,:}^{-1}\right)_{-p}, \tag{3.4}$$

which corresponds to performing Gaussian elimination of row and column p in $\mathbf{H}^{-1}$ followed by dropping them completely. This has $\Theta(d_{\text{col}}^2)$ time complexity.

The resulting pseudocode is shown in Algorithm 3.1, where we avoid constantly resizing $\mathbf{H}^{-1}$ (and correspondingly changing indices) by utilizing the fact that row and column p have no effect on any future calculations after they have been eliminated by Lemma 1 as they are 0 (and the non-zero diagonal element is never accessed again). One can check that this algorithm applies OBS to a single row of $\mathbf{W}$ with a per-step cost of $\Theta(d_{\text{col}}^2)$, and thus $\Theta(k \cdot d_{\text{col}}^2)$ overall time for pruning k weights.

Algorithm 3.1: Prune $k \leq d_{\text{col}}$ weights from row $\mathbf{w}$ with inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top)^{-1}$ according to OBS in $O(k \cdot d_{\text{col}}^2)$ time.

$M = \{1, \ldots, d_{\text{col}}\}$
for $i = 1, \ldots, k$ **do**
  $p \leftarrow \text{argmin}_{p \in M} \frac{1}{[\mathbf{H}^{-1}]_{pp}} \cdot w_p^2$
  $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{H}^{-1}_{:,p} \frac{1}{[\mathbf{H}^{-1}]_{pp}} \cdot w_p$
  $\mathbf{H}^{-1} \leftarrow \mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{pp}} \mathbf{H}^{-1}_{:,p} \mathbf{H}^{-1}_{p,:}$
  $M \leftarrow M - \{p\}$
end for

Step 2: Jointly Considering All Rows. Applying the OBS framework to the full weight matrix $\mathbf{W}$ rather than just to each row independently requires fast access to all d_{row} row-wise inverse Hessians, in order to select the weight with the smallest overall pruning score in each step. However, storing d_{row} matrices of size $d_{\text{col}} \times d_{\text{col}}$ each in GPU memory can be too expensive; while it would be possible to offload some Hessians to main memory, this could result in a large number of expensive memory transfers. However, since there is no Hessian interaction between rows, the final compressed weights of each row only depend on the total number of parameters that were pruned in it. Similarly, the change in loss incurred by pruning some weight only depends on the previously pruned weights in the same row, which also means that the order in which weights are pruned in each row is fixed.

The consequence of these insights is that we can process each row independently, pruning all weights in order while always recording the corresponding change in loss $\delta\mathcal{L}_p = w_p^2 / [\mathbf{H}^{-1}]_{pp}$. At the end, we know $\delta\mathcal{L}_p$ for all d weights and can then simply determine the global mask that would be chosen by OBS on the full matrix by selecting the weights with the lowest values in order, requiring only $\Theta(d)$ extra memory. We note that once the per-row masks M_i are known, we can directly solve for the optimal update of the remaining weights via the corresponding group OBS formula [KCN+22] $\boldsymbol{\delta}_{\boldsymbol{M_i}} = \mathbf{H}^{-1}_{:,M_i}((\mathbf{H}^{-1})_{M_i})^{-1}\mathbf{w}_{M_i}$. This will be considerably faster in practice than simply rerunning the iterative pruning process in Algorithm 3.1. Alternatively, if enough CPU memory is available, one can keep the full *pruning trace* of each row, that is, the full weight vector after every individual pruning step, in CPU memory and ultimately simply reload the entries corresponding to the global mask. This requires $O(d_{\text{row}} \cdot d_{\text{col}}^2)$ extra CPU memory but avoids a second computation pass to reconstruct the not pruned weights and will therefore be faster. Figure 3.1 visualizes both options just discussed.

Implementation Details. In practice, the matrix $\mathbf{H}$ might not always be invertible for reasons such as using too few data samples or dead / linearly dependent inputs. The former

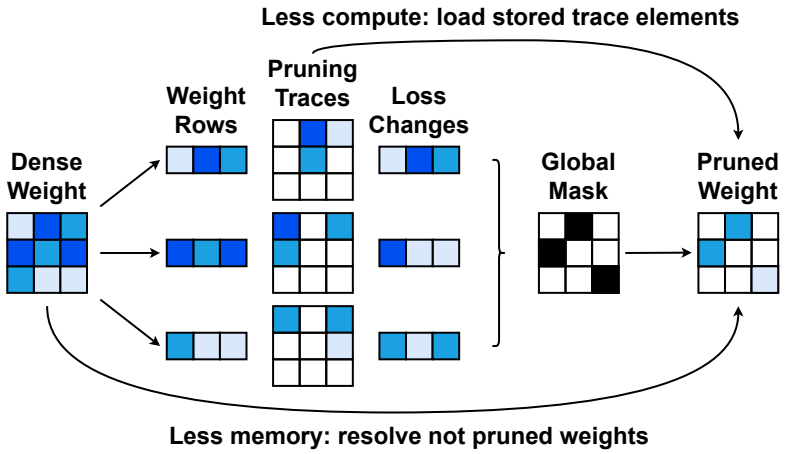


Figure 3.1: Efficient global OBS using the row-wise results.

can usually be addressed by extending the calibration dataset with augmentations (additional augmented samples only need to be accumulated into the Hessian once and are thus very cheap to include) and the latter can be prevented by adding a small diagonal dampening term to the Hessian before inverting it. Second, a direct GPU implementation of Algorithm 3.1 will perform a large number of small CUDA calls, which can be expensive. This overhead can be removed by using batch operations to process multiple matrix rows simultaneously—for more details please see our sample implementation. Finally, when applied to an already sparse weight matrix, the complexity of our algorithm can scale cubicly with the row-density by working with a dense version of the weights / Hessians consisting only of the non-zero elements and mapping the pruning result back at the end.

N:M Sparsity. Our method can be easily extended to various forms of *semi-structured* sparsity. This includes, for example, the N:M sparsity pattern [ZMZ+21], which enforces exactly N non-zero values in each block of M consecutive weights, and is becoming popular due to support on newer NVIDIA hardware [MLP+21]. Adapting our algorithm to this pattern requires only one simple change: instead of selecting the weight with the smallest change in loss, we select the weight with the smallest change in loss that is in a block with $< N$ pruned weights. We note that all rows have exactly the same sparsity $1 - N/M$ in the N:M pattern and so we can terminate per-row pruning as soon as this target sparsity value is reached. For the same reason, there is no need for the global mask selection step described earlier. Thus, our method will be even more efficient in this case.

Block-Sparsity. Another practically relevant pruning pattern, particularly in the context of CPU acceleration [EDGS20, KKG+20], is *block-pruning*, where zeros appear only in consecutive blocks of size c, which is typically a small number like 4 or 8. We follow recent work [KCN+22] that extends the OBS framework to pruning small groups of connected weights in order to account for the correlation between them, using the following formulas for the target block and weight update, respectively:

$$\mathbf{w}_P = \text{argmin}_{\mathbf{w}_P} \mathbf{w}_P^\top ((\mathbf{H}^{-1})_P)^{-1} \mathbf{w}_P, \quad \boldsymbol{\delta_P} = -\mathbf{H}^{-1}_{:,P} ((\mathbf{H}^{-1})_P)^{-1} \mathbf{w}_P, \tag{3.5}$$

where P denotes the set of indices corresponding to one block. Algorithm 3.1 can easily be adapted to operate on blocks using the above equations and applying the update of $\mathbf{H}^{-1}$ via Lemma 1 successively for all $p \in P$. Although there are now only d_{col}/c steps per row, each

update of $\mathbf{H^{-1}}$ also takes $O(c \cdot d_{\text{col}}^2)$ time and so the overall asymptotic runtime stays the same. Additional practical overhead only comes from the extra $O(c^2 \cdot d_{\text{col}}^2)$ terms that are the result of computing and multiplying with the $c \times c$ matrices $((\mathbf{H}^{-1})_P)^{-1}$.

3.1.5 The Optimal Brain Quantizer (OBQ)

Although the classical OBS framework [LDS89, HSW93] has inspired a long line of work on pruning methods for DNNs [SA20, FKA21, LZK+21], so far it has not been used for quantization. We now show that our results from the previous section can in fact be extended to quantization in an effective and accurate way, via a method which we call the Optimal Brain Quantizer (OBQ), in the spirit of [LDS89, HSW93].

The Quantization Order and Update Derivations. Under the standard assumption that the gradient at the current point $\mathbf{w}$ is negligible, the OBS formulas for the optimal weight to be pruned w_p and the corresponding update $\boldsymbol{\delta_p}$ can be derived by writing the locally quadratic problem under the constraint that element p of $\boldsymbol{\delta_p}$ is equal to $-w_p$, which means that w_p is zero after applying the update to $\mathbf{w}$. This problem has the following Lagrangian:

$$L(\boldsymbol{\delta_p}, \lambda) = \boldsymbol{\delta_p}^\top \mathbf{H} \boldsymbol{\delta_p} + \lambda(\mathbf{e_p}^\top \boldsymbol{\delta_p} - (-w_p)), \tag{3.6}$$

where $\mathbf{H}$ denotes the Hessian at $\mathbf{w}$ and $\mathbf{e_p}$ is the pth canonical basis vector. The optimal solution is then derived by first finding the optimal solution to $\boldsymbol{\delta_p}$ via setting the derivative $\partial L / \partial \boldsymbol{\delta_p}$ to zero and then substituting this solution back into L and solving for λ; please see e.g. [HSW93, SA20] for examples.

Assume a setting in which we are looking to quantize the weights in a layer on a fixed grid of width Δ while minimizing the loss. To map OBS to a *quantized* projection, we can set the target of the Lagrangian constraint in (3.6) to $(\text{quant}(w_p) - w_p)$, where $\text{quant}(w_p)$ is the weight rounding given by quantization; then $w_p = \text{quant}(w_p)$ after the update.

Assuming we wish to quantize weights iteratively, one-at-a-time, we can derive formulas for the "optimal" weight to quantize at a step, in terms of minimizing the loss increase, and for the corresponding optimal update to the unquantized weights, in similar fashion as discussed above:

$$w_p = \text{argmin}_{w_p} \frac{(\text{quant}(w_p) - w_p)^2}{[\mathbf{H}^{-1}]_{pp}}, \quad \boldsymbol{\delta_p} = -\frac{w_p - \text{quant}(w_p)}{[\mathbf{H}^{-1}]_{pp}} \cdot \mathbf{H}^{-1}_{:,p}. \tag{3.7}$$

In fact, since $-w_p$ is a constant during all derivations, we can just substitute it with $(\text{quant}(w_p) - w_p)$ in the final result. We note that the resulting formulas are a generalization of standard OBS for pruning, if $\text{quant}(\cdot)$ always "quantizes" a weight to 0, then we recover the original form.

Quantizing Full Layers. At first glance, OBQ might appear curious since one usually quantizes *all* weights in a layer, leaving no more weights to update. At the same time, the weight selection metric influences only the quantization order, but not the quantization value. However, this view changes when considering OBQ in the context of our efficient one-weight-at-a-time pruning algorithm described in the previous section. Specifically, using OBQ, we can greedily quantize the currently "easiest" weight by the above metric, and then adjust all the remaining unquantized weights to compensate for this loss of precision, thus changing their value. We then choose the next weight to quantize, and so on. This can result in quantization assignments that are different from the ones that would have been chosen by

rounding initially, and in better overall quantization results. Concretely, to realize this, we can plug (3.7) into Algorithm 3.1 in order to iteratively quantize weights for a given layer, leading to the similar Algorithm in Appendix A.1.2, thus essentially unifying pruning and quantization.

Quantization Outliers. One practical issue with this greedy scheme can occur especially when applied to quantization grids that permit some outliers in order to achieve a lower error on the majority of weights, which are currently standard [CKYK19, NCB+21]. Since these outliers can have high quantization error, they will usually be quantized last, when there are only few other unquantized weights available that may be adjusted to compensate for the large error incurred by quantizing the outliers. This effect can become worse when some weights are pushed even further outside the grid by intermediate updates. We prevent this with a simple but effective heuristic: we quantize outliers, e.g. weights with a quantization error $> \Delta/2$ where Δ is the distance between quantized values, as soon as they appear (which typically happens only a few times per layer). With this heuristic, OBQ yields a highly effective layer-wise quantization scheme, as our experiments in the next section demonstrate. Finally, we note that the OBQ version of the techniques discussed in Section 3.1.4 has all the same runtime and memory characteristics (barring the global step in Figure 3.1, which is unnecessary for quantization).

3.1.6 Experiments

Objectives, Models & Datasets. To demonstrate the effectiveness and flexibility of our method, we consider several different standard *post-training compression* scenarios [NAVB+20, HNH+21, HCI+21]. We begin with settings where only a single type of compression is applied: concretely, we consider unstructured pruning for given FLOP targets, global 2:4 and 4:8 pruning, as well as uniform weight quantization. Additionally, we also study two practical tasks that feature joint pruning and quantization: a GPU scenario where quantization and N:M pruning are combined, as well as a CPU scenario combining quantization and block pruning. We work with variants of the following models and tasks: ResNet [HZRS16] for image classification on Imagenet [RDS+15], YOLOv5 [Joc22] for object detection on COCO [LMB+14] and BERT [DCLT19] for question answering on SQuAD [RZLL16]. Our smaller BERT models denoted by BERT3 and BERT6 correspond to the smaller 3 and 6 layer variants of BERT-base, respectively, trained by [KCN+22]. Appendix A.1.3 detailed runtime information on our algorithms.

Experimental Setup. All of our calibration datasets consist of 1024 random training samples. For ImageNet, where we use roughly 0.1% of the training data, we additionally apply standard flipping and cropping augmentations to artificially increase the size of this dataset by $10\times$; other tasks do not use any augmentations. While the effect of augmentations is typically minor, they are very cheap to include for our method. For ResNet models, batchnorm statistics are reset using 100 batches of 128 samples from the calibration set with standard augmentations. For other models, we apply mean and variance correction [NBBW19, BNS19] after all normalization layers (so that the correction parameters can be easily merged and incur no extra cost) on a single batch of samples of size 128 (for YOLO) and 512 (for BERT). We found this to be more effective than batchnorm tuning for YOLO, and the BERT models have no batchnorm layers.

When compressing to a given FLOP or timing constraint, we need to solve the problem of identifying per-layer compression targets, which match the constraint, while maximizing

accuracy. To identify these non-uniform targets, we follow the approach of [FKA21]: we first collect a "model database" containing for each compression level (e.g. bit-width or sparsity setting) the corresponding (independently) compressed version of each layer. For building a joint sparse and quantized database we simply sparsify layers first and then apply quantization to the remaining weights. Next, similarly to [HNH+21], we compute the layer-wise calibration losses (without augmentations) for all compression levels, corresponding to the models with exactly one layer compressed to a certain level. Then, given layer-wise FLOP or timing information, we set up a constrained layer-wise compression problem of the form described in AdaQuant [HNH+21] and solve it with the dynamic programming algorithm of SPDY [FKA21]. This returns an optimal per-layer assignment of compression levels, for which we can then easily produce the corresponding model, via a two-step process: we first stitch together layers at the corresponding compression levels from the database, and then perform the discussed statistics correction to recover some extra accuracy [HNH+21].

Unstructured Sparsity. We begin our experiments with *unstructured* sparsity, comparing against global magnitude pruning (GMP) [ZG17], the approximate layer-wise OBS method L-OBS [DCP17], and the post-training pruning state-of-the-art method AdaPrune [HCI+21]. As a sanity check, we examine in Figure 3.2 whether our method provides better results in terms of layer-wise squared error, pruning the first layer of a ResNet18 (RN18) model to several sparsities. In this metric, ExactOBS performs best by a wide margin ahead of AdaPrune, which significantly outperforms the other two methods.

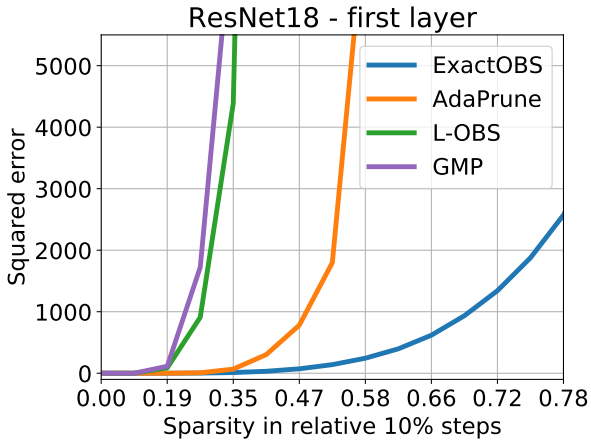


Figure 3.2: RN18 squared error.

Next, in Table 3.1, we turn our attention to the practical problem of pruning various models to achieve a given FLOP reduction of $2\times$–$4\times$, applying the per-layer target sparsity optimization technique described above. Our ExactOBS generally performs best (except for YOLOv5l $2\times$ where all methods perform similarly in terms of mAP@0.5) and at $4\times$ FLOP reduction even with a $> 1\%$ gap to the next best method. Interestingly, on the hard-to-prune BERT model, ExactOBS appears to be the only method which still produces reasonable results at higher reduction targets. For BERT $3\times$ and $4\times$, where the performance drop of all methods is $> 2\%$, we additionally assess the compatibility of our results with the more powerful (but also more expensive) post processing method *global AdaPrune* [FKA21]. While this global optimization technique is able to recover lost accuracy, the ExactOBS models still maintain a $> 0.5\%$ and $> 2\%$ F1 advantage, respectively (see Table 3.2).

Method	ResNet50 – 76.13			YOLOv5l – 66.97			BERT – 88.53		
	2×	3×	4×	2×	3×	4×	2×	3×	4×
GMP	74.86	71.44	64.84	65.83	62.30	55.09	65.64	12.52	09.23
L-OBS	75.48	73.73	71.24	**66.21**	64.47	61.15	77.67	3.62	6.63
AdaPrune	75.53	74.47	72.39	66.00	64.88	62.71	87.12	70.32	18.75
ExactOBS	**75.64**	**75.01**	**74.05**	66.14	**65.35**	**64.05**	**87.81**	**85.87**	**82.10**

Table 3.1: Unstructured pruning for different FLOP reduction targets.

Methods	BERT	
	3×	4×
gAP + AdaPrune	86.99	84.10
gAP + ExactOBS	**87.57**	**86.42**

Table 3.2: Further improving results in Table 3.1 with $> 3\%$ performance drops through more expensive post-processing via global AdaPrune (gAP).

N:M Sparsity. Next, we study the performance of our method for *semi-structured* sparsity via the N:M pattern. Specifically, we compare against the 4:8 results of AdaPrune with batchnorm tuning [HCI+21] on ResNet models (see Table 3.3) in addition to a 2:4 comparison on BERT models (see Table 3.4). We highlight that ExactOBS matches or even slightly exceeds the 4:8 results of AdaPrune with the considerably more stringent 2:4 pattern, which is already well supported on NVIDIA hardware. Furthermore, in a 2:4 comparison on BERT models, ExactOBS achieves 1–2% higher F1 scores.

Model	Dense	AdaPrune	ExactOBS	
		4:8	2:4	4:8
ResNet18	69.76	68.63	68.81	**69.18**
ResNet34	73.31	72.36	72.66	**72.95**
ResNet50	76.13	74.75	74.71	**75.20**

Table 3.3: Semi-structured N:M pruning (+ batchnorm tuning) of all layers except the first and the last.

Model	Dense	AdaPrune	ExactOBS
BERT3	84.66	82.75	**83.54**
BERT6	88.33	85.02	**86.97**
BERT	88.53	85.24	**86.77**

Table 3.4: Semi-structured 2:4 pruning of all layers except the embeddings.

Quantization. Additionally, we compare OBQ's *independent* performance (after batchnorm tuning) with the state-of-the-art *sequential* post-training methods AdaQuant [HNH+21], AdaRound [NAVB+20] and BRECQ [LGT+21]. We perform standard asymmetric per-channel quantization of all weights, using the authors' implementations. We rerun all methods on Torchvision [MR10] ResNets to ensure a uniform baseline. The quantization grids for OBQ as

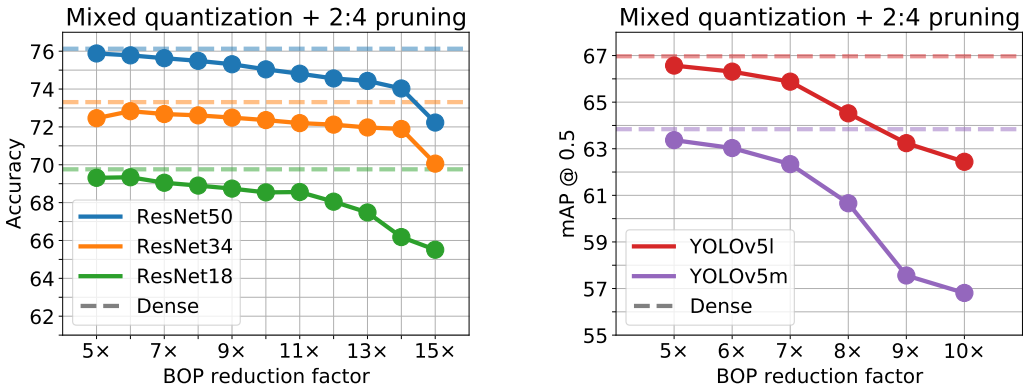
well as AdaRound are determined with the same LAPQ [NCB+21] procedure that is used by BRECQ. Surprisingly, we find that, despite optimizing layers independently, OBQ achieves very similar (sometimes even slightly better) accuracies as existing non-independent methods for 4 and 3 bits. This suggests that it should be well-suited for mixed precision applications where one needs to quickly generate many non-uniform models optimized for different constraints. (However, we note that ExactOBS can also be applied sequentially.)

Method	Lw.	Ind.	ResNet18 – 69.76			ResNet50 – 76.13		
			4bit	3bit	2bit	4bit	3bit	2bit
AdaRound	yes	no	69.34	68.37	63.37	75.84	75.14	71.58
AdaQuant	yes	no	68.12	59.21	00.10	74.68	64.98	00.10
BRECQ	no	no	69.37	68.47	64.70	75.88	75.32	72.41
OBQ (ours)	yes	yes	69.56	68.69	64.04	75.72	75.24	70.71

Table 3.5: Comparison with state-of-the-art post-training methods for asymmetric per-channel weight quantization of all layers. We mark whether methods are Layer-wise (Lw.) or Independent (Ind.).

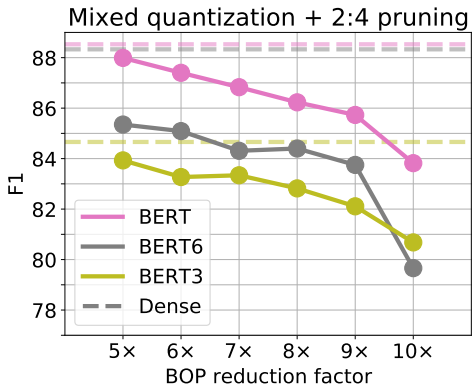
BOP-Constrained Mixed GPU Compression. We now consider a practical setting where we are given a trained model together with some calibration data and want to compress this model for efficient inference on an NVIDIA GPU which supports 8-bit and 4-bit arithmetic, also in combination with 2:4 sparsity. Thus, there are 4 possible compression choices per layer: 8bit weights + 8bit activations (8w8a), 4w4a, 8w8a + 2:4 and 4w4a + 2:4. Unlike in the previous section, we do *symmetric* per-channel quantization of the weights as it has better hardware support; activations are quantized asymmetrically per-tensor. We then generate mixed precision configurations for various BOP (number of bits times FLOPs) reduction targets and visualize the resulting compression-accuracy trade-off curves in Figure 3.3. In summary, at the cost of a $\approx 2.5\%$ relative performance drop, we can achieve a $12-14\times$ BOP reduction for ResNets and a $7-8\times$ reduction for the more challenging YOLO and BERT models (relative to the compute in compressible layers). To the best of our knowledge, we are the first to consider joint N:M pruning and quantization in a post-training setting. Recent work [CHBS23] also studies joint 4w4a + 2:4 compression for ResNet18 but with 90 epochs of (sparse) Quantization-Aware Training (QAT) on the full dataset and report 67.33% accuracy. Although not perfectly comparable (we keep the first layer dense and their dense baseline has 0.94% higher accuracy and uses 4:8 sparse activations), we achieve similar 67.20% accuracy for 4w4a + 2:4 *post training*, which emphasizes the effectiveness of our methods for joint sparsification and quantization.

Time-Constrained CPU Compression. Lastly, we explore a similar scenario, but targeting actual CPU inference speedup on a 12-core Intel Xeon Silver 4214 CPU using the DeepSparse inference engine [Neu22, KKG+20], which provides acceleration for joint 8-bit quantization and block-sparsity with blocksize 4. In this case, we work with real layer-wise timing data (for batchsize 64), as in [FA22]. There are 30 available block-sparsity targets per-layer, in steps of pruning 10% of the remaining weights, all of which are further quantized to 8 bits. The base acceleration of the dense 8 bit model is $\approx 2.7\times$ on top of which sparsity speedup acts roughly multiplicatively. Figure 3.4 shows results for ResNet50 and several (real-time) speedup targets—we achieve $4\times$ and $5\times$ (actual) speedup with 1% and 2% accuracy loss,



(a) ResNet variants.

(b) YOLO variants.



(c) BERT variants.

Figure 3.3: Mixed quantization and 2:4 pruning for various BOP reduction targets.

respectively. These are the first full post-training results in this setting (the authors of [FKA21] only performed 4-block pruning post-training, followed by 5 epochs of QAT on the entire ImageNet dataset), and they show very encouraging accuracy-speedup trade-offs.

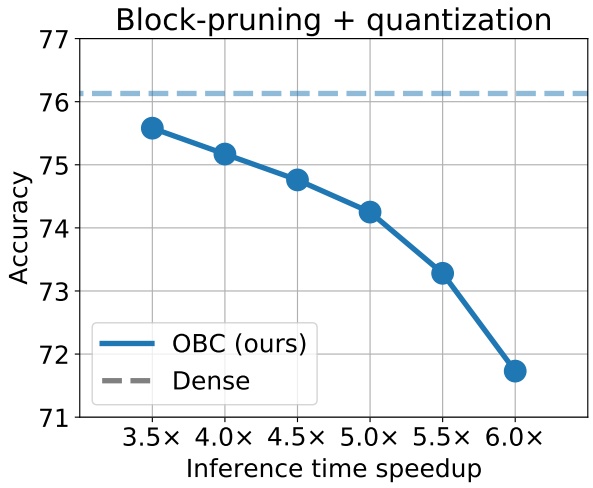


Figure 3.4: Joint block-pruning and quantization for CPU inference time speedups.

3.1.7 Conclusions & Future Work

We have presented a new efficient and accurate approach for solving the layer-wise compression problem, and built on it to obtain state-of-the-art post-training compression solutions for both pruning and quantization. Our framework should be naturally extensible to *structured* pruning, which in fact should allow for further optimizations, and should also be compatible with further compression via unstructured pruning and quantization. Our results suggest that post-training compression may be able to reach comparable accuracies to much more expensive retraining methods. The next section will investigate this, in particular in the context of more resource-intensive models, such as very large-scale language models.

3.2 GPTQ: Post-Training Quantization for Generative Pre-Trained Transformers

3.2.1 Motivation & Overview

Pre-trained generative models from the Transformer [VSP+17] family, commonly known as GPT or OPT [RWC+19, BMR+20, ZRG+22], have shown breakthrough performance for complex language modelling tasks, leading to massive academic and practical interest. One major obstacle to their usability is computational and storage cost, which ranks among the highest for known models. For instance, the best-performing model variants, e.g. GPT3-175B, have in the order of 175 billion parameters and require tens-to-hundreds of GPU years to train [ZRG+22]. Even the simpler task of inferencing over a pre-trained model, which is our focus in this section, is highly challenging: for instance, the parameters of GPT3-175B occupy 326GB (counting in multiples of 1024) of memory when stored in a compact float16 format. This exceeds the capacity of even the highest-end single GPUs, and thus inference must be performed using more complex and expensive setups, such as multi-GPU deployments.

Although a standard approach to eliminating these overheads is *model compression*, e.g. [HABN+21, GKD+21], surprisingly little is known about compressing such models for inference. One reason is that more complex methods for low-bitwidth quantization or model pruning usually require *model retraining*, which is extremely expensive for billion-parameter models. Alternatively, *post-training* methods [NAVB+20, WCHC20, HNH+20, NCB+21], which compress the model in one shot, without retraining, would be very appealing. Unfortunately, the more accurate variants of such methods [LGT+21, HNH+21] (and Section 3.1) are complex and challenging to scale to billions of parameters [YAZ+22]. To date, only basic variants of round-to-nearest quantization [YAZ+22, DLBZ22] have been applied at the scale of GPT-175B; while this works well for low compression targets, e.g., 8-bit weights, they fail to preserve accuracy at higher rates. It therefore remains open whether one-shot *post-training quantization* to higher compression rates is generally-feasible.

Contribution. In this section, we present a new post-training quantization method, called GPTQ[1], which is efficient enough to execute on models with hundreds of billions of parameters in at most a few hours, and precise enough to compress such models to 3 or 4 bits per parameter without significant loss of accuracy. For illustration, GPTQ can quantize the largest

[1]This merges the name of the GPT model family with the abbreviation for post-training quantization (PTQ).

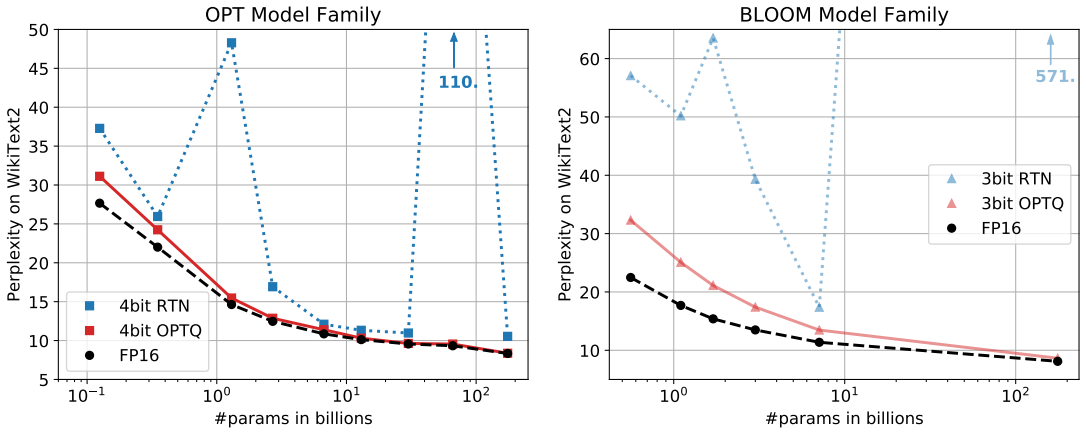


Figure 3.5: Quantizing OPT models to 4 and BLOOM models to 3 bit precision, comparing GPTQ with the FP16 baseline and round-to-nearest (RTN) [YAZ+22, DLBZ22].

publicly-available models, OPT-175B and BLOOM-176B, in approximately four GPU hours, with minimal increase in perplexity, known to be a very stringent accuracy metric.

Further, we show that our model can also provide robust results in the *extreme quantization* regime, in which models are quantized to 2 bits per component, or even *ternary values*. On the practical side, we develop an execution harness which allows us to execute the resulting compressed models efficiently for generative tasks. Specifically, we are able to run the compressed OPT-175B model for the first time on a single NVIDIA A100 GPU, or using only two more cost-effective NVIDIA A6000 GPUs. We also implement bespoke GPU kernels which are able to leverage compression for faster memory loading, resulting in speedups of $\approx 3.25\times$ when using A100 GPUs, and $4.5\times$ when using A6000 GPUs.

To our knowledge, we are the first to show that extremely accurate language models with hundreds of billions of parameters can be quantized to 3-4 bits/component: prior *post-training methods* only remain accurate at 8 bits [YAZ+22, DLBZ22], while prior *training-based* techniques have only tackled models that are smaller by one to two orders of magnitude [WYZ+22]. This high degree of compression may appear natural, as these networks are overparametrized; yet, as we discuss in our detailed analysis of results, compression induces non-trivial tradeoffs between the accuracy of the language modeling (perplexity), bit-width, and the size of the original model.

We hope that our work will stimulate further research in this area, and can be a further step towards making these models available to a wider audience. In terms of limitations, our method currently does not provide speedups for the actual multiplications, due to the lack of hardware support for mixed-precision operands (e.g. FP16 x INT4) on mainstream architectures. Moreover, our current results do not include activation quantization, as they are not a significant bottleneck in our target scenarios; however, this can be supported using orthogonal techniques [YAZ+22].

3.2.2 Related Work

Quantization methods fall broadly into two categories: quantization during training, and post-training methods. The former quantize models during typically extensive retraining and/or finetuning, using some approximate differentiation mechanism for the rounding operation [GKD+21, NFA+21]. By contrast, post-training ("one-shot") methods quantize a pretrained

model using modest resources, typically a few thousand data samples and a few hours of computation. Post-training approaches are particularly interesting for massive models, for which full model training or even finetuning can be expensive. We focus on this scenario here.

Post-training Quantization. For an overview of prior post-training quantization work, please see Section 3.1.2. Most existing methods of this kind have focused on vision models, whereas we focus on language here. Further, while these approaches can produce good results for models up to ≈ 100 million parameters in a few GPU hours, scaling them to networks orders of magnitude larger is challenging.

Large-model Quantization. With the recent open-source releases of language models like BLOOM [SFA+22] or OPT-175B [ZRG+22], researchers have started to develop affordable methods for compressing such giant networks for inference. While all existing works—ZeroQuant [YAZ+22], LLM.int8() [DLBZ22], and nuQmm [PPK+22]—carefully select quantization granularity, e.g., vector-wise, they ultimately just round weights to the nearest (RTN) quantization level, in order to maintain acceptable runtimes for very large models. ZeroQuant further proposes layer-wise knowledge distillation, similar to AdaQuant, but the largest model it can apply this approach to has only 1.3 billion parameters. At this scale, ZeroQuant already takes ≈ 3 hours of compute; GPTQ quantizes models $100\times$ larger in ≈ 4 hours. LLM.int8() observes that *activation outliers* in a few feature dimensions break the quantization of larger models, and proposes to fix this problem by keeping those dimensions in higher precision. Lastly, nuQmm develops efficient GPU kernels for a specific binary-coding based quantization scheme.

Relative to this line of work, we show that a significantly more complex and accurate quantizer can be implemented efficiently at large model scale. Specifically, GPTQ more than doubles the amount of compression relative to these prior techniques, at similar accuracy.

3.2.3 The GPTQ Algorithm

Background. Similar to Section 3.1, GPTQ operates in the *layer-wise quantization* framework (see Section 3.1.3 for details). Our approach is based on the Optimal Brain Quantization (OBQ) method introduced in Section 3.1.5 for solving this layer-wise problem, to which we perform a series of major modifications, which allow it to scale to large language models, providing more than *three orders of magnitude* computational speedup. We note that OBQ can achieve reasonable runtimes on medium-sized models: for instance, it can fully quantize the ResNet-50 model (25M parameters) in ≈ 1 hour on a single GPU, which is roughly in line with other post-training methods achieving state-of-the-art accuracy (Appendix A.1.3). However, the fact that OBQ's runtime for a $d_{\text{row}} \times d_{\text{col}}$ matrix $\mathbf{W}$ has *cubic* input dependency $O(d_{\text{row}} \cdot d_{\text{col}}^3)$ means that applying it to models with billions of parameters is extremely expensive.

Step 1: Arbitrary Order Insight. As explained in Section 3.1.5, OBQ quantizes weights in greedy order, i.e. it always picks the weight which currently incurs the least additional quantization error. Interestingly, we find that, while this quite natural strategy does indeed seem to perform very well, its improvement over quantizing the weights in arbitrary order is generally small, in particular on large, heavily-parametrized layers. Most likely, this is because the slightly lower number of quantized weights with large individual error is balanced out by those weights being quantized towards the end of the process, when only few other unquantized

weights that can be adjusted for compensation remain. As we will now discuss, this insight that *any fixed order may perform well*, especially on large models, has interesting ramifications.

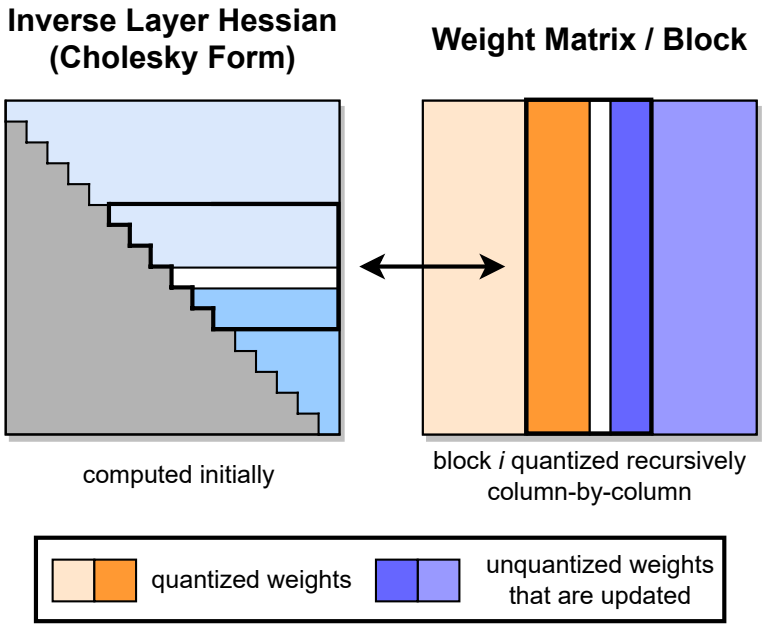


Figure 3.6: GPTQ quantization procedure. Blocks of consecutive *columns* (bolded) are quantized at a given step, using the inverse Hessian information stored in the Cholesky decomposition, and the remaining weights (blue) are updated at the end of the step. The quantization procedure is applied recursively inside each block: the white middle column is currently being quantized.

The original OBQ method quantizes rows of $\mathbf{W}$ independently, in a specific order defined by the corresponding errors. By contrast, we will aim to quantize the weights of *all rows in the same order*, and will show that this typically yields results with a final squared error that is similar to the original solutions. As a consequence, the set of unquantized weights F and similarly $\mathbf{H}_F^{-1}$ is always the same for all rows (see Figure 3.6 for an illustration). In more detail, the latter is due to the fact that $\mathbf{H}_F$ depends only on the layer inputs $\mathbf{X}_F$, which are the same for all rows, and not on any weights. Therefore, we have to perform the update of $\mathbf{H}_F^{-1}$ given by Equation (3.4) only d_{col} times, once per column, rather than $d_{\text{row}} \cdot d_{\text{col}}$ times, once per weight. This reduces the overall runtime from $O(d_{\text{row}} \cdot d_{\text{col}}^3)$ to $O(\max\{d_{\text{row}} \cdot d_{\text{col}}^2, d_{\text{col}}^3\})$, i.e., by a factor of $\min\{d_{\text{row}}, d_{\text{col}}\}$. For larger models, this difference consists of several orders of magnitude. However, before this algorithm can actually be applied to very large models in practice, two additional major problems need to be addressed.

Step 2: Lazy Batch-Updates. First, a direct implementation of the scheme described previously will not be fast in practice, because the algorithm has a relatively low compute-to-memory-access ratio. For example, Equation (3.4) needs to update all elements of a potentially huge matrix using just a few FLOPs for each entry. Such operations cannot properly utilize the massive compute capabilities of modern GPUs, and will be bottlenecked by the significantly lower memory bandwidth.

Fortunately, this problem can be resolved by the following observation: The final rounding decisions for column i are only affected by updates performed on this very column, and so updates to later columns are irrelevant at this point in the process. This makes it possible

to "lazily batch" updates together, thus achieving much better GPU utilization. Concretely, we apply the algorithm to $B = 128$ columns at a time, keeping updates contained to those columns and the corresponding $B \times B$ block of $\mathbf{H}^{-1}$ (see also Figure 3.6). Only once a block has been fully processed, we perform global updates of the entire $\mathbf{H}^{-1}$ and $\mathbf{W}$ matrices using the multi-weight versions of Equations (3.7) and (3.4) given below, with Q denoting a set of indices, and $\mathbf{H}^{-1}_{-Q}$ denoting the inverse matrix with the corresponding rows and columns removed:

$$\boldsymbol{\delta}_F = -(\mathbf{w}_Q - \text{quant}(\mathbf{w}_Q))([\mathbf{H}_F^{-1}]_{QQ})^{-1}(\mathbf{H}_F^{-1})_{:,Q}, \tag{3.8}$$

$$\mathbf{H}^{-1}_{-Q} = \left(\mathbf{H}^{-1} - \mathbf{H}^{-1}_{:,Q}([\mathbf{H}^{-1}]_{QQ})^{-1}\mathbf{H}^{-1}_{Q,:}\right)_{-Q}. \tag{3.9}$$

Although this strategy does not reduce the theoretical amount of compute, it effectively addresses the memory-throughput bottleneck. This provides an order of magnitude speedup for very large models in practice, making it a critical component of our algorithm.

Step 3: Cholesky Reformulation. The final technical issue we have to address is given by numerical inaccuracies, which can become a major problem at the scale of existing models, especially when combined with the block updates discussed in the previous step. Specifically, it can occur that the matrix $\mathbf{H}_F^{-1}$ becomes indefinite, which we notice can cause the algorithm to aggressively update the remaining weights in incorrect directions, resulting in an arbitrarily-bad quantization of the corresponding layer. In practice, we observed that the probability of this happening increases with model size: concretely, it almost certainly occurs for at least a few layers on models that are larger than a few billion parameters. The main issue appears to be the repeated applications of Equation (3.9), which accumulate various numerical errors, especially through the additional matrix inversion.

For smaller models, applying dampening, that is adding a small constant λ (we always choose 1% of the average diagonal value) to the diagonal elements of $\mathbf{H}$ appears to be sufficient to avoid numerical issues. However, larger models require a more robust and general approach.

To address this, we begin by noting that the only information required from $\mathbf{H}_{F_q}^{-1}$, where F_q denotes the set of unquantized weights when quantizing weight q, is row q, or more precisely, the elements in this row starting with the diagonal. The consequence is that we could precompute all of these rows using a more numerically-stable method without any significant increase in memory consumption. Indeed, the row removal via (3.4) for our symmetric $\mathbf{H}^{-1}$ essentially corresponds to taking a Cholesky decomposition, except for the minor difference that the latter divides row q by $([\mathbf{H}_{F_q}^{-1}]_{qq})^{1/2}$. Hence, we can leverage state-of-the-art Cholesky kernels to compute all information we will need from $\mathbf{H}^{-1}$ upfront. In combination with mild dampening, the resulting method is robust enough to execute on huge models without issues. As a bonus, using a well-optimized Cholesky kernel also yields further speedup. We detail all small changes necessary for the Cholesky version of the algorithm next.

The Full Algorithm. Finally, we present the full pseudocode for GPTQ in Algorithm 3.2, including the optimizations discussed above.

Algorithm 3.2: Quantize $\mathbf{W}$ given inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize B.

$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ // *quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ // *block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ // *Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
  for $j = i, \ldots, i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ // *quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) \,/\, [\mathbf{H}^{-1}]_{jj}$ // *quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ // *update weights in block*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ // *update all remaining weights*
end for

3.2.4 Experimental Validation

Overview. We begin our experiments by validating the accuracy of GPTQ relative to other accurate-but-expensive quantizers, on smaller models, for which these methods provide reasonable runtimes. Next, we examine GPTQ's runtime scaling for very large models. Then, we present 3- and 4-bit quantization results for the entire BLOOM and OPT model families, evaluated via perplexity on challenging language generation tasks. In addition, we show that our method is also stable for 2-bit quantization when the granularity is reduced to small blocks of consecutive weights. To complement this perplexity analysis, we also evaluate the resulting quantized models on a series of standard zero-shot tasks. Finally, we focus on the two largest (and interesting) openly-available models, Bloom-176B and OPT-175B, where we perform a detailed evaluation on several tasks. For these models, we also present practical improvements, namely reducing the number of GPUs required for inference as well as end-to-end speedups for generative tasks.

Setup. We implemented GPTQ in PyTorch [PGM+19] and worked with the HuggingFace integrations of the BLOOM [LSW+22] and OPT [ZRG+22] model families. We quantized all models (including the 175 billion parameter variants) *using a single NVIDIA A100 GPU* with 80GB of memory. Our entire GPTQ calibration data consists of 128 random 2048 token segments from the C4 dataset [RSR+20a], i.e., excerpts from randomly crawled websites, which represents generic text data. We emphasize that this means that GPTQ does not see any task-specific data, and our results thus remain actually "zero-shot". We perform standard uniform per-row asymmetric quantization on the min-max grid, similar to [DLBZ22].

To ensure that the entire compression procedure can be performed with significantly less GPU memory than what would be required to run the full precision model, some care must be taken. Specifically, we always load one Transformer block, consisting of 6 layers, at a time into GPU memory and then accumulate the layer-Hessians and perform quantization. Finally, the current block inputs are sent through the fully quantized block again to produce the new inputs for the quantization of the next block. Hence, the quantization process operates not on the layer inputs in the full precision model but on the actual layer inputs in the already partially quantized one. We find that this brings noticeable improvements at negligible extra cost.

Baselines. Our primary baseline, denoted by RTN, consists of rounding all weights to the nearest quantized value on exactly the same asymmetric per-row grid that is also used for GPTQ, meaning that it corresponds precisely to the state-of-the-art weight quantization of LLM.int8(). This is currently the method of choice in all works on quantization of very large language models [DLBZ22, YAZ+22, PPK+22]: its runtime scales well to networks with many billions of parameters, as it simply performs direct rounding. As we will also discuss further, more accurate methods, such as AdaRound [NAVB+20] or BRECQ [LGT+21], are currently too slow for models with many billions of parameters, the main focus of this work. Nevertheless, we also show that GPTQ is competitive with such methods for small models, while scaling to huge ones like OPT-175B as well.

Quantizing Small Models. As a first ablation study, we compare GPTQ's performance relative to state-of-the-art post-training quantization (PTQ) methods, on ResNet18 and ResNet50, which are standard PTQ benchmarks, in the same setup as in Section 3.1.6. As can be seen in Table 3.6, GPTQ performs on par at 4-bit, and slightly worse than the most accurate methods at 3-bit. At the same time, it significantly outperforms AdaQuant, the fastest amongst prior PTQ methods. Further, we compare against the full greedy OBQ method on two smaller language models: BERT-base [DCLT19] and OPT-125M. The results are shown in Table 3.7. At 4 bits, both methods perform similarly, and for 3 bits, GPTQ surprisingly performs slightly better. We suspect that this is because some of the additional heuristics used by OBQ, such as early outlier rounding, might require careful adjustments for optimal performance on non-vision models. Overall, GPTQ appears to be competitive with state-of-the-art post-training methods for smaller models, while taking only < 1 minute rather than ≈ 1 hour. This enables scaling to much larger models.

Method	RN18 – 69.76 %		RN50 – 76.13%	
	4bit	3bit	4bit	3bit
AdaRound	69.34	68.37	75.84	75.14
AdaQuant	68.12	59.21	74.68	64.98
BRECQ	69.37	68.47	75.88	75.32
OBQ	69.56	68.69	75.72	75.24
GPTQ	69.37	67.88	75.71	74.87

Table 3.6: Comparison with state-of-the-art post-training methods for vision models.

Method	BERT-base 88.53 F1 ↑		OPT-125M 27.66 PPL ↓	
	4bit	3bit	4bit	3bit
OBQ	88.23	85.29	32.52	69.32
GPTQ	88.18	86.02	31.12	53.85

Table 3.7: Comparison of GPTQ relative to OBQ on BERT-base/SQuAD and OPT-125M/WikiText2.

Runtime. Next we measure the full model quantization time (on a single NVIDIA A100 GPU) via GPTQ; the results are shown in Table 3.8. As can be seen, GPTQ quantizes 1-3

billion parameter models in a matter of minutes and 175B ones in a few hours. For reference, the straight-through based method ZeroQuant-LKD [YAZ+22] reports a 3 hour runtime (on the same hardware) for a 1.3B model, which would linearly extrapolate to several hundred hours (a few weeks) for 175B models. Adaptive rounding-based methods typically employ a lot more SGD steps and would thus be even more expensive [NAVB+20, LGT+21].

OPT	13B	30B	66B	175B
Runtime	20.9m	44.9m	1.6h	4.2h
BLOOM	1.7B	3B	7.1B	176B
Runtime	2.9m	5.2m	10.0m	3.8h

Table 3.8: GPTQ runtime for full quantization of the 4 largest OPT and BLOOM models.

Language Generation. We begin our large-scale study by compressing the entire OPT and BLOOM model families to 3- and 4-bit. We then evaluate those models on several language tasks including WikiText2 [MXBS17] (see Figure 3.5 as well as Tables 3.9 and 3.10), Penn Treebank (PTB) [MKM+94] and C4 [RSR+20a] (both in Appendix A.2.1). We focus on these perplexity-based tasks, as they are known to be particularly sensitive to model quantization [YAZ+22]. On OPT models, GPTQ clearly outperforms RTN, by significant margins. For example, GPTQ loses only 0.03 perplexity at 4-bit on the 175B model, while RTN drops 2.2 points, performing worse than the $10\times$ smaller full-precision 13B model. At 3-bit, RTN collapses completely, while GPTQ can still maintain reasonable perplexity, in particular for larger models. BLOOM shows a similar pattern: the gaps between methods are however usually a bit smaller, indicating that this model family might be easier to quantize. One interesting trend (see also Figure 3.5) is that larger models generally (with the exception of OPT-66B[2]) appear easier to quantize. This is good news for practical applications, as these are the cases where compression is also the most necessary.

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	27.65	22.00	14.63	12.47	10.86	10.13	9.56	9.34	8.34
RTN	4	37.28	25.94	48.17	16.92	12.10	11.32	10.98	110	10.54
GPTQ	4	**31.12**	**24.24**	**15.47**	**12.87**	**11.39**	**10.31**	**9.63**	**9.55**	**8.37**
RTN	3	1.3e3	64.57	1.3e4	1.6e4	5.8e3	3.4e3	1.6e3	6.1e3	7.3e3
GPTQ	3	**53.85**	**33.79**	**20.97**	**16.88**	**14.86**	**11.61**	**10.27**	**14.16**	**8.68**

Table 3.9: OPT perplexity results on WikiText2.

175 Billion Parameter Models. We now examine BLOOM-176B and OPT-175B, the largest dense openly-available models. Table 3.11 summarizes results across Wikitext-2, PTB, C4. We observe that, at 4 bits, GPTQ models reach only ≤ 0.25 lower perplexity than the full-precision versions, with a large gap to RTN results on OPT-175B. At 3-bit, RTN collapses, while GPTQ is still able to maintain good performance on most tasks, losing only $0.3 - 0.6$ points for more than $5\times$ compression. We note that GPTQ's accuracy can be

[2]Upon closer inspection of the OPT-66B model, it appears that this is correlated with the fact that this trained model has a significant fraction of dead units in the early layers, which may make it harder to compress.

BLOOM	Bits	560M	1.1B	1.7B	3B	7.1B	176B
full	16	22.42	17.69	15.39	13.48	11.37	8.11
RTN	4	25.90	22.00	16.97	14.76	12.10	8.37
GPTQ	4	**24.03**	**19.05**	**16.48**	**14.20**	**11.73**	**8.21**
RTN	3	57.08	50.19	63.59	39.36	17.38	571
GPTQ	3	**32.31**	**25.08**	**21.11**	**17.40**	**13.47**	**8.64**

Table 3.10: BLOOM perplexity results for WikiText2.

further improved via finer-granularity grouping [PPK+22]: group-size 1024 ($\approx$ 0.02 extra bits) improves perplexities by about 0.2 on average and group-size 128 ($\approx$ 0.15 extra bits) by another 0.1, which is only $0.1 - 0.3$ off from the uncompressed accuracy. We note that grouping interacts very well with GPTQ, as the group parameters can be determined during the quantization process of each layer, always using the most current updated weights.

Method	Bits	OPT-175B				BLOOM-176B			
		Wiki2	PTB	C4	LAMB. ↑	Wiki2	PTB	C4	LAMB. ↑
Baseline	16	8.34	12.01	10.13	75.59	8.11	14.59	11.71	67.40
RTN	4	10.54	14.22	11.61	71.34	8.37	15.00	12.04	66.70
GPTQ	4	**8.37**	**12.26**	**10.28**	**76.80**	**8.21**	**14.75**	**11.81**	**67.71**
RTN	3	7.3e3	8.0e3	4.6e3	0	571.	107.	598.	0.17
GPTQ	3	**8.68**	**12.68**	**10.67**	**76.19**	**8.64**	**15.57**	**12.27**	**65.10**
GPTQ	3/g1024	8.45	12.48	10.47	77.39	8.35	15.01	11.98	67.47
GPTQ	3/g128	8.45	12.37	10.36	76.42	8.26	14.89	11.85	67.86

Table 3.11: Results summary for OPT-175B and BLOOM-176B. "g1024" and "g128" denote results with groupings of size 1024 and 128, respectively.

Practical Speedups. Finally, we study practical applications. As an interesting use-case, we focus on the OPT-175B model: quantized to 3 bits, this model takes approximately 63GB of memory, including the embeddings and the output layer, which are kept in full FP16 precision. Additionally, storing the complete history of keys and values for all layers, a common optimization for generation tasks, consumes another ≈ 9GB for the maximum of 2048 tokens. Hence, we can actually fit the entire quantized model into a single 80GB A100 GPU, which can be executed by dynamically dequantizing layers as they are required during inference (the model would not fully fit using 4 bits). For reference, standard FP16 execution requires 5x80GB GPUs, and the state-of-the-art 8bit LLM.int8() quantizer [DLBZ22] requires 3 such GPUs.

Next, we consider language generation, one of the most appealing applications of these models, with the goal of latency reduction.Unlike LLM.int8(), which reduces memory costs but has the same runtime as the FP16 baseline, we show that our quantized models can achieve significant speedups for this application. For language generation, the model processes and outputs one token at-a-time, which for OPT-175B can easily take a few 100s of milliseconds per token. Increasing the speed at which the user receives generated results is challenging, as compute is dominated by matrix-vector products.Unlike matrix-matrix products, these are primarily limited

by memory bandwidth. We address this problem by developing a quantized-matrix full-precision-vector product kernel which performs a matrix vector product by dynamically dequantizing weights when needed. Most notably, this does *not* require any activation quantization. While dequantization consumes extra compute, the kernel has to access a lot less memory, leading to significant speedups, as shown in Table 3.12. We note that almost all of the speedup is due to our kernels, as communication costs are negligible in our standard HuggingFace-accelerate-like setting (see Appendix A.2.2 for details).

GPU	FP16	3bit	Speedup	GPU reduction
A6000 – 48GB	589ms	130ms	$4.53\times$	$8 \rightarrow 2$
A100 – 80GB	230ms	71ms	$3.24\times$	$5 \rightarrow 1$

Table 3.12: Average per-token latency (batch size 1) when generating lenght 128 sequences.

For example, using our kernels, the 3-bit OPT-175B model obtained via GPTQ running on a single A100 is about $\mathbf{3.25}\times$ faster than the FP16 version (running on 5 GPUs) in terms of average time per token. More accessible GPUs, such as the NVIDIA A6000, have much lower memory bandwidth, so this strategy is even more effective: executing the 3-bit OPT-175B model on 2x A6000 GPUs reduces latency from 589 milliseconds for FP16 inference (on 8 GPUs) to 130 milliseconds, a $\mathbf{4.5}\times$ latency reduction.

Zero-Shot Tasks. While our focus is on language generation, we also evaluate the performance of quantized models on some popular zero-shot tasks, namely LAMBADA [PKL+16], ARC (Easy and Challenge) [BPM+18] and PIQA [TP03]. Figure 3.7 visualizes model performance on LAMBADA (and see also "Lamb." results in Table 3.11). We observe similar behavior as before: the outliers are that 1) quantization appears "easier" across the whole spectrum of models at 4-bit, where even RTN performs relatively well, and 2) at 3-bit, RTN breaks down, while GPTQ still provides good accuracy.

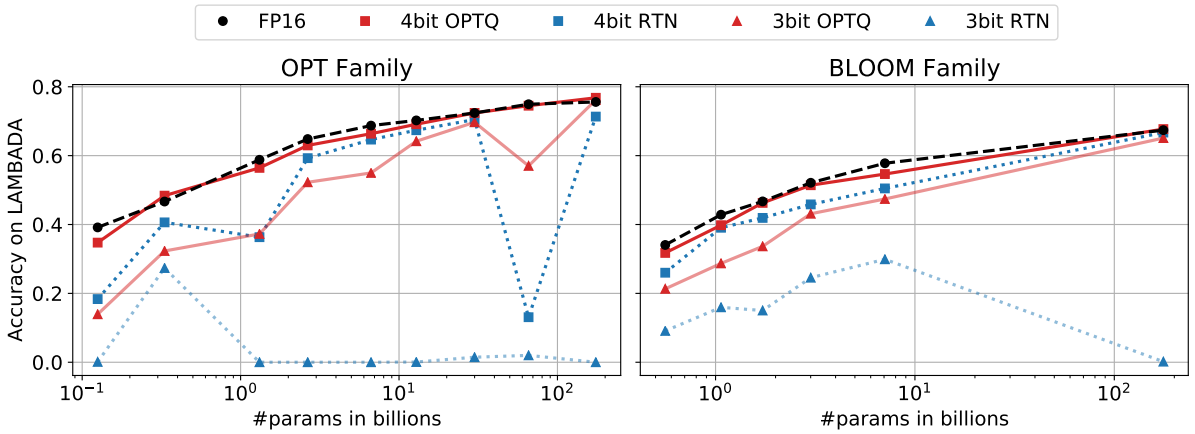


Figure 3.7: The accuracy of OPT and BLOOM models post-GPTQ, measured on LAMBADA.

Additional Tricks. While our experiments so far have focused exclusively on vanilla row-wise quantization, we want to emphasize that GPTQ is *compatible with essentially any choice of quantization grid*. For example, it is easily combined with standard *grouping* [AGL+17, PPK+22], i.e. applying independent quantization to groups of g consecutive weights. As shown in the last rows of Table 3.11, this can bring noticeable extra accuracy for

the largest models at 3-bit. Further, as visualized in Figure 3.8, it significantly reduces the accuracy losses for medium sized models at 4-bit precision.

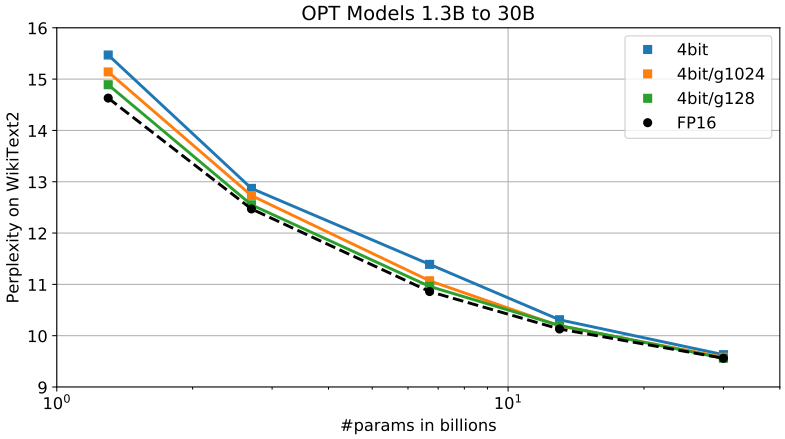


Figure 3.8: GPTQ at 4-bit with different group-sizes on medium sized OPT models.

Extreme Quantization. Lastly, grouping also makes it possible to achieve reasonable performance for extreme quantization, to around 2-bits per component on average. Table 3.13 shows results on WikiText2 when quantizing the biggest models to 2-bit with varying group-sizes. At ≈ 2.2 bit (group-size 128; using FP16 scale and 2-bit zero point per group) the perplexity increase is already less than 1.5 points, while dropping to 0.6 - 0.7 at ≈ 2.6 bit (group-size 32), which is only slightly worse than vanilla 3-bit and might be interesting for practical kernel implementations. Further, if we reduce group size to 8, we can apply *ternary* (-1, 0, $+1$) quantization, which achieves 9.20 WikiText2 PPL on OPT-175B, a less than 1 point drop. While this leads to worse compression on average relative to the 2-bit numbers above, this pattern could be efficiently implemented on custom hardware such as FPGAs. In summary, these results are an encouraging first step towards pushing highly-accurate *one-shot* compression of very large language models, even lower than 3 bits per value on average.

Model	FP16	g128	g64	g32	3-bit
OPT-175B	8.34	9.58	9.18	8.94	8.68
BLOOM	8.11	9.55	9.17	8.83	8.64

Table 3.13: 2-bit GPTQ quantization results with varying group-sizes; perplexity on WikiText2.

3.2.5 Summary and Limitations

We have presented GPTQ, an approximate second-order method for quantizing truly large language models. GPTQ can accurately compress some of the largest publicly-available models down to 3 and 4 bits, which leads to significant usability improvements, and to end-to-end speedups, at low accuracy loss. We hope that our method will make these models accessible to more researchers and practitioners. At the same time, we emphasize some significant limitations: On the technical side, our method obtains speedups from reduced memory movement, and does not lead to computational reductions. In addition, our study focuses on generative tasks, and does not consider activation quantization. These are natural directions for future work, and we believe this can be achieved with carefully-designed GPU kernels and existing techniques [YAZ+22, WYZ+22].

3.3 SparseGPT: Massive Language Models Can Be Accurately Pruned

3.3.1 Motivation & Overview

A complementary compression approach to quantization, which we considered in Section 3.2, is *pruning*, removing network elements, from individual weights (unstructured pruning) to higher-granularity structures such as rows/columns of the weight matrices (structured pruning). Pruning has a long history [LDS89, HSW93], and has been applied successfully in the case of vision and smaller-scale language models [HABN+21]. Yet, the best-performing pruning methods require *extensive retraining* of the model to recover accuracy. In turn, this is extremely expensive for GPT-scale models. While some accurate *one-shot* pruning methods exist, like [HCI+21] or the OBC approach outlined in Section 3.1, compressing a model without retraining, unfortunately even they become very expensive when applied to models with billions of parameters. Thus, to date, there is essentially no work on accurate pruning of billion-parameter models.

Overview. In this section, we propose `SparseGPT`, the first accurate one-shot pruning method which works efficiently at the scale of models with 10-100+ billion parameters. `SparseGPT` works by reducing the pruning problem to a set of extremely large-scale instances of *sparse regression*. It then solves these instances via a new approximate sparse regression solver, which is efficient enough to execute in a few hours on the largest openly-available GPT models (175B parameters), on a single GPU. At the same time, `SparseGPT` is accurate enough to drop negligible accuracy post-pruning, without any fine-tuning. For example, when executed on the largest publicly-available generative language models (OPT-175B and BLOOM-176B), `SparseGPT` induces 50-60% sparsity in one-shot, with minor accuracy loss, measured either in terms of perplexity or zero-shot accuracy.

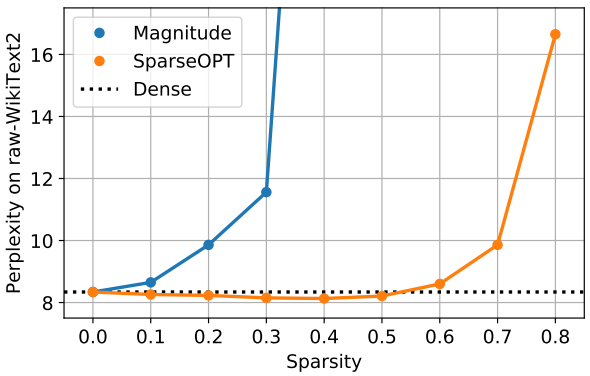


Figure 3.9: Sparsity-vs-perplexity comparison of `SparseGPT` against magnitude pruning on OPT-175B, when pruning to different *uniform* per-layer sparsities.

Our experiments, from which we provide a snapshot in Figures 3.9 and 3.10, lead to the following observations. First, as shown in Figure 3.9, `SparseGPT` can induce uniform layer-wise sparsity of up to 60% in e.g. the 175-billion-parameter variant of the OPT family [ZRG+22], with minor accuracy loss. By contrast, the only known one-shot baseline which easily extends to this scale, Magnitude Pruning [Hag94, HPTD15], preserves accuracy only until 10% sparsity, and completely collapses beyond 30% sparsity. Second, as shown in Figure 3.10, `SparseGPT` can also accurately impose sparsity in the more stringent, but hardware-friendly, 2:4 and 4:8

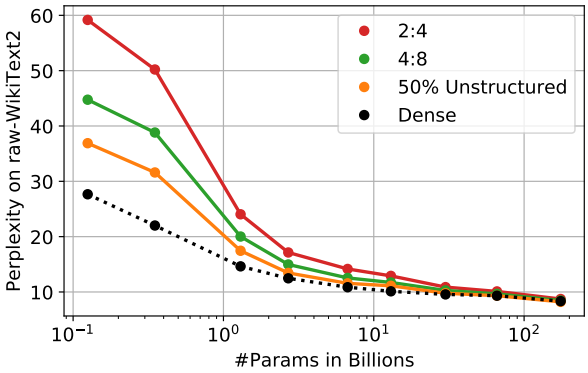


Figure 3.10: Perplexity vs. model and sparsity type when compressing the entire OPT model family (135M, 350M, ..., 66B, 175B) to different sparsity patterns using `SparseGPT`.

semi-structured sparsity patterns [MLP+21], although this comes at an accuracy loss relative to the dense baseline for smaller models.

One key positive finding, illustrated in Figure 3.10, is that *larger models are more compressible*: they drop significantly less accuracy at a fixed sparsity, relative to their smaller counterparts. (For example, the largest models from the OPT and BLOOM families can be sparsified to 50% with almost no increase in perplexity.) In addition, our method allows sparsity to be *compounded* with weight quantization techniques as the one introduced in Section 3.2: for instance, we can induce 50% weight sparsity jointly with 4-bit weight quantization with negligible perplexity increase on OPT-175B.

One notable property of `SparseGPT` is that it is *entirely local*, in the sense that it relies solely on weight updates designed to preserve the input-output relationship for each layer, which are computed without any global gradient information. As such, we find it remarkable that one can directly identify such sparse models in the "neighborhood" of dense pretrained models, whose output correlates extremely closely with that of the dense model.

3.3.2 Background

Overall, we will operate in a layer-wise post-training setup very similar to the two previous sections. For clarity, we repeat some of the previous definitions, highlighting new aspects that will be particularly relevant for deriving the `SparseGPT` algorithm.

Layer-Wise Pruning. Overall, we will operate in a very similar layer-wise post-training setting Post-training compression is usually done by splitting the full-model compression problem into *layer-wise* subproblems, whose solution quality is measured in terms of the ℓ_2-error between the output, for given inputs $\mathbf{X}_\ell$, of the uncompressed layer with weights $\mathbf{W}_\ell$ and that of the compressed one. Specifically, for pruning, [HCI+21] posed this problem as that of finding, for each layer ℓ, a sparsity mask $\mathbf{M}_\ell$ with a certain target density, and possibly updated weights $\widehat{\mathbf{W}}_\ell$ such that

$$\text{argmin}_{\text{mask } \mathbf{M}_\ell, \widehat{\mathbf{W}}_\ell} ||\mathbf{W}_\ell \mathbf{X}_\ell - (\mathbf{M}_\ell \odot \widehat{\mathbf{W}}_\ell)\mathbf{X}_\ell||_2^2. \tag{3.10}$$

Mask Selection & Weight Reconstruction. A key aspect of the layer-wise pruning problem in (3.10) is that both the mask $\mathbf{M}_\ell$ as well as the remaining weights $\widehat{\mathbf{W}}_\ell$ are optimized

jointly, which makes this problem NP-hard [BD08]. Thus, exactly solving it for larger layers is unrealistic, leading all existing methods to resort to approximations.

A particularly popular approach is to separate the problem into *mask selection* and *weight reconstruction* [HLL+18, KKM+22, HCI+21]. Concretely, this means to first choose a pruning mask $\mathbf{M}$ according to some saliency criterion, like the weight magnitude [ZG17], and then optimize the remaining unpruned weights while keeping the mask unchanged. Importantly, once the mask is fixed, (3.10) turns into a *linear squared error problem* that is easily optimized.

Existing Solvers. Early work [Kin97] applied iterated linear regression to small networks. More recently, the AdaPrune approach [HCI+21] has shown good results for this problem on modern models via magnitude-based weight selection, followed by applying SGD steps to reconstruct the remaining weights. Follow-up works demonstrate that pruning accuracy can be further improved by removing the strict separation between mask selection and weight reconstruction. Iterative AdaPrune [FA22] performs pruning in gradual steps with reoptimization in between and OBC [FSA22] introduces a greedy solver which removes weights one-at-a-time, fully reconstructing the remaining weights after each iteration, via efficient closed-form equations.

Difficulty of Scaling to 100+ Billion Parameters. Prior post-training techniques have all been designed to accurately compress models up to a few hundred million parameters with several minutes to a few hours of compute. However, our goal here is to sparsify models up to $1000\times$ larger.

Even AdaPrune, the method optimized for an ideal speed/accuracy trade-off, takes a few hours to sparsify models with just 1.3 billion parameters (see also Section 3.3.4), scaling linearly to several hundred hours (a few weeks) for 175B Transformers. More accurate approaches are at least several times more expensive [FA22] than AdaPrune or even exhibit worse than linear scaling [FSA22]. This suggests that scaling up existing accurate post-training techniques to extremely large models is a challenging endeavor. Hence, we propose a new layer-wise solver `SparseGPT`, based on careful approximations to closed form equations, which easily scales to giant models, both in terms of runtime as well as accuracy.

3.3.3 The `SparseGPT` Algorithm

Fast Approximate Reconstruction

Motivation. As outlined in Section 3.3.2, for a fixed pruning mask $\mathbf{M}$, the optimal values of all weights in the mask can be calculated exactly by solving the sparse reconstruction problem corresponding to each matrix row $\mathbf{w^i}$ via:

$$\mathbf{w^i}_{\mathbf{M_i}} = (\mathbf{X}_{\mathbf{M_i}}\mathbf{X}_{\mathbf{M_i}}^\top)^{-1}\mathbf{X}_{\mathbf{M_i}}(\mathbf{w}_{\mathbf{M_i}}\mathbf{X}_{\mathbf{M_i}})^\top, \tag{3.11}$$

where $\mathbf{X}_{\mathbf{M_i}}$ denotes only the subset of input features whose corresponding weights have not been pruned in row i, and $\mathbf{w}_{\mathbf{M_i}}$ represents their respective weights. However, this requires inverting the Hessian matrix $\mathbf{H}_{\mathbf{M_i}} = \mathbf{X}_{\mathbf{M_i}}\mathbf{X}_{\mathbf{M_i}}^\top$ corresponding to the values preserved by the pruning mask $\mathbf{M_i}$ for row i, i.e. computing $(\mathbf{H}_{\mathbf{M_i}})^{-1}$, separately for all rows $1 \leq i \leq d_{\text{row}}$. One such inversion takes $O(d_{\text{col}}^3)$ time, for a total computational complexity of $O(d_{\text{row}} \cdot d_{\text{col}}^3)$ over d_{row} rows. For a Transformer model, this means that the overall runtime scales with the 4th power of the hidden dimension d_{hidden}; we need a speedup by at least a full factor of d_{hidden} to arrive at a practical algorithm.

Different Row-Hessian Challenge. The high computational complexity of optimally reconstrucing the unpruned weights following Equation 3.11 mainly stems from the fact that solving *each row* requires the *individual* inversion of a $O(d_{\text{col}} \times d_{\text{col}})$ matrix. This is because the row masks $\mathbf{M_i}$ are generally different and $(\mathbf{H}_{\mathbf{M_i}})^{-1} \neq (\mathbf{H}^{-1})_{\mathbf{M_i}}$, i.e., the inverse of a masked Hessian does *not* equal the masked version of the full inverse. This is illustrated also in Figure 3.11. If all row-masks were the same, then we would only need to compute a single shared inverse, as $\mathbf{H} = \mathbf{X}\mathbf{X}^\top$ depends just on the layer inputs which are the same for all rows.

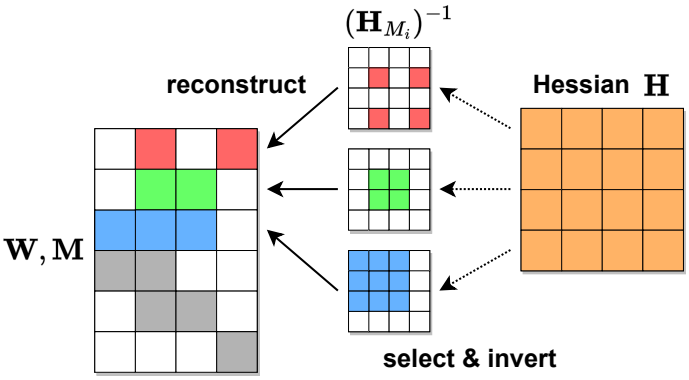


Figure 3.11: Illustration of the row-Hessian challenge: rows are sparsified independently, pruned weights are in white.

Such a constraint could be enforced in the mask selection, but this would have a major impact on the final model accuracy, as sparsifying weights in big structures, like entire columns, is known to be much more difficult than pruning them individually[3]. The key towards designing an approximation algorithm that is both accurate and efficient lies in enabling the reuse of Hessians between rows with distinct pruning masks. We now propose an algorithm that achieves this in a principled manner.

Equivalent Iterative Perspective. To motivate our algorithm, we first have to look at the row-wise weight reconstruction from a different *iterative* perspective, using the classic OBS update [HSW93, SA20, FKA21]. Assuming a quadratic approximation of the loss, for which the current weights $\mathbf{w}$ are optimal, the OBS update $\boldsymbol{\delta}_m$ provides the optimal adjustment of the remaining weights to compensate for the removal of the weight at index m, with error ε_m:

$$\boldsymbol{\delta_m} = -\frac{w_m}{[\mathbf{H}^{-1}]_{mm}} \cdot \mathbf{H}^{-1}_{:,m}, \quad \varepsilon_m = \frac{w_m^2}{[\mathbf{H}^{-1}]_{mm}}. \tag{3.12}$$

Since the loss function corresponding to the layer-wise pruning of one row of $\mathbf{W}$ is a quadratic, the OBS formula is exact in this case. Hence, $\mathbf{w} + \boldsymbol{\delta}_m$ is the optimal weight reconstruction corresponding to mask $\{m\}^C$. Further, given an optimal sparse reconstruction $\mathbf{w}^{(\mathbf{M})}$ corresponding to mask $\mathbf{M}$, we can apply OBS again to find the optimal reconstruction for mask $\mathbf{M}' = \mathbf{M} - \{m\}$. Consequently, this means that instead of solving for a full mask $\mathbf{M} = \{m_1, \ldots, m_p\}^C$ directly, we could iteratively apply OBS to individually prune the weights m_1 up until m_p in order, one-at-a-time, reducing an initially complete mask to $\mathbf{M}$, and will ultimately arrive at *the same* optimal solution as applying the closed-form regression reconstruction with the full $\mathbf{M}$ directly.

[3]For example, structured (column-wise) pruning ResNet50 to $> 50\%$ structured sparsity without accuracy loss is challenging, even with extensive retraining [LZK+21], while unstructured pruning to 90% sparsity is easily achievable with state-of-the-art methods [EGM+20, PIVA21].

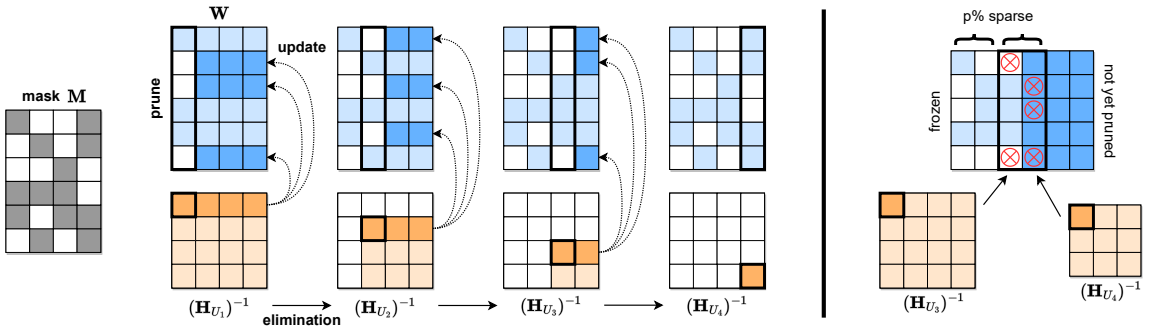


Figure 3.12: [Left] Visualization of the SparseGPT reconstruction algorithm. Given a fixed pruning mask $\mathbf{M}$, we incrementally prune weights in each column of the weight matrix $\mathbf{W}$, using a sequence of Hessian inverses $(\mathbf{H}_{U_j})^{-1}$, and updating the remainder of the weights in those rows, located to the "right" of the column being processed. Specifically, the weights to the "right" of a pruned weight (dark blue) will be updated to compensate for the pruning error, whereas the unpruned weights do not generate updates (light blue). [Right] Illustration of the adaptive mask selection via iterative blocking.

Optimal Partial Updates. Applying the OBS update $\boldsymbol{\delta}_m$ potentially adjusts the values of all available parameters (in the current mask $\mathbf{M}$) in order to compensate for the removal of w_m. However, what if we only update the weights in a subset $\mathbf{U} \subseteq \mathbf{M}$ among remaining unpruned weights? Thus, we could still benefit from error compensation, using only weights in $\mathbf{U}$, while reducing the cost of applying OBS.

Such a partial update can indeed be accomplished by simply computing the OBS update using $\mathbf{H}_\mathbf{U}$, the Hessian corresponding to $\mathbf{U}$, rather than $\mathbf{H}_\mathbf{M}$, and updating only $\mathbf{w}_\mathbf{U}$. Importantly, the loss of our particular layer-wise problem remains quadratic also for $\mathbf{U}$ and the OBS updates are still optimal: the restriction to $\mathbf{U}$ does not incur any extra approximation error by itself, only the error compensation might not be as effective, as less weights are available for adjustment. At the same time, if $|\mathbf{U}| < |\mathbf{M}|$, then inverting $\mathbf{H}_\mathbf{U}$ will be a lot faster than inverting $\mathbf{H}_\mathbf{M}$. We will now utilize this mechanism to accomplish our goal of synchronizing the masked Hessians across all rows of $\mathbf{W}$.

Hessian Synchronization. In the following, assume a fixed ordering of the input features $j = 1, \ldots, d_{\text{col}}$. Since those are typically arranged randomly, we will just preserve the given order for simplicity, but any permutation could in principle be chosen. Next, we define a sequence of d_{col} index subsets U_j recursively as:

$$U_{j+1} = U_j - \{j\} \text{ with } U_1 = \{1, \ldots, d_{\text{col}}\}. \tag{3.13}$$

In words, starting with U_1 being the set of all indices, each subset U_{j+1} is created by removing the smallest index from the previous subset U_j. These subsets also impose a sequence of inverse Hessians $(\mathbf{H}_{U_j})^{-1} = ((\mathbf{X}\mathbf{X}^\top)_{U_j})^{-1}$ which we are going to share across all rows of $\mathbf{W}$. Crucially, following Section 3.1, the updated inverse $(\mathbf{H}_{U_{j+1}})^{-1}$ can be calculated efficiently by removing the first row and column, corresponding to j in the original $\mathbf{H}$, from $\mathbf{B} = (\mathbf{H}_{U_j})^{-1}$ in $O(d_{\text{col}}^2)$ time via one step of Gaussian elimination:

$$(\mathbf{H}_{U_{j+1}})^{-1} = \left(\mathbf{B} - \frac{1}{[\mathbf{B}]_{11}} \cdot \mathbf{B}_{:,1}\mathbf{B}_{1,:}\right)_{2:,2:}, \tag{3.14}$$

with $(\mathbf{H}_{U_1})^{-1} = \mathbf{H}^{-1}$. Hence, the entire sequence of d_{col} inverse Hessians can be calculated recursively in $O(d_{\text{col}}^3)$ time, i.e. at similar cost to a single extra matrix inversion on top of the initial one for $\mathbf{H}^{-1}$.

Once some weight w_k has been pruned, it should not be updated anymore. Further, when we prune w_k, we want to update as many unpruned weights as possible for maximum error compensation. This leads to the following strategy: iterate through the U_j and their corresponding inverse Hessians $(\mathbf{H}_{U_j})^{-1}$ in order and prune w_j if $j \notin M_i$, for all rows i. Importantly, each inverse Hessian $(\mathbf{H}_{U_j})^{-1}$ is computed only once and reused to remove weight j in all rows where it is part of the pruning mask. A visualization of the algorithm can be found in Figure 3.12.

Computational Complexity. The overall cost consists of three parts: (a) the computation of the initial Hessian, which takes time $\Theta(n \cdot d_{\text{col}}^2)$ where n is the number of input samples used—we found that taking the number of samples n to be a small multiple of d_{col} is sufficient for good and stable results, even on very large models (see Appendix A.3.1); (b) iterating through the inverse Hessian sequence in time $O(d_{\text{col}}^3)$ and (c) the reconstruction/pruning itself. The latter cost can be upper bounded by the time it takes to apply (3.12) to all d_{row} rows of $\mathbf{W}$ for all d_{col} columns in turn, which is $O(d_{\text{col}} d_{\text{row}} d_{\text{col}})$. In total, this sums up to $O(d_{\text{col}}^3 + d_{\text{row}} d_{\text{col}}^2)$. For Transformer models, this is simply $O(d_{\text{hidden}}^3)$, and is thus a full d_{hidden}-factor more efficient than exact reconstruction. This means that we have reached our initial goal, as this complexity will be sufficient to make our scheme practical, even for extremely large models.

Weight Freezing Interpretation. While we have motivated the `SparseGPT` algorithm as an approximation to the exact reconstruction using optimal partial updates, there is also another interesting view of this scheme. Specifically, consider an exact greedy framework which compresses a weight matrix column by column, always optimally updating all not yet compressed weights in each step as we did in the previous two sections. At first glance, `SparseGPT` does not seem to fit into this framework as we only compress some of the weights in each column and also only update a subset of the uncompressed weights. Yet, mechanically, "compressing" a weight ultimately means fixing it to some specific value and ensuring that it is never "decompressed" again via some future update, i.e. that it is *frozen*. Hence, by defining column-wise compression as:

$$\text{compress}(\mathbf{w}^{\mathbf{j}})_i = 0 \text{ if } j \notin M_i \text{ and } w_i^j \text{ otherwise,} \tag{3.15}$$

i.e. zeroing weights not in the mask and fixing the rest to their current value, our algorithm can be interpreted as an exact column-wise greedy scheme. This perspective will allow us to cleanly merge sparsification and quantization into a single compression pass.

Adaptive Mask Selection

So far, we have focused only on weight reconstruction, i.e. assuming a fixed pruning mask $\mathbf{M}$. One simple option for deciding the mask, following AdaPrune [HCI+21], would be via magnitude pruning [ZG17]. However, Section 3.1 shows that updates during pruning change weights significantly due to correlations, and that taking this into account in the mask selection yields better results. This insight can be integrated into `SparseGPT` by *adaptively* choosing the mask while running the reconstruction.

One obvious way of doing so would be picking the $p\%$ easiest weights to prune in each column i when it is compressed, leading to $p\%$ overall sparsity. The big disadvantage of this approach is that sparsity cannot be distributed non-uniformly across columns, imposing additional unnecessary structure. This is particularly problematic for massive language models, which have a small number of highly-sensitive outlier features [DLBZ22, XLS+23].

We remove this disadvantage via *iterative blocking*. More precisely, we always select the pruning mask for $B_s = 128$ columns at a time (see Appendix A.3.1), based on the OBS

reconstruction error ε from Equation (3.12), using the diagonal values in our Hessian sequence. We then perform the next B_s weight updates, before selecting the mask for the next block, and so on. This procedure allows *non-uniform selection* per column, in particular also using the corresponding Hessian information, while at the same time considering also previous weight updates for selection. (For a single column j, the selection criterion becomes the magnitude, as $[\mathbf{H}^{-1}]_{jj}$ is constant across rows.)

Extension to Semi-Structured Sparsity

`SparseGPT` is also easily adapted to *semi-structured* patterns such as the popular n:m sparsity format [ZMZ+21, HCI+21] which delivers speedups in its 2:4 implementation on Ampere NVIDIA GPUs. Specifically, every consecutive m weights should contain exactly n zeros. Hence, we can simply choose blocksize $B_s = m$ and then enforce the zeros-constraint in the mask selection for each row by picking the n weights which incur the lowest error as per Equation (3.12). A similar strategy could also be applied for other semi-structured pruning patterns. Finally, we note that a larger B_s would not be useful in this semi-structured scenario since zeros cannot be distributed non-uniformly between different column-sets of size m.

Full Algorithm Pseudocode

Algorithm 3.3: The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$ *// binary pruning mask*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ *// Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
  for $j = i, \ldots, i + B - 1$ **do**
    if $j \bmod B_s = 0$ **then**
      $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1-p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
    end if
    $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} \,/\, [\mathbf{H}^{-1}]_{jj}$ *// pruning error*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$ *// freeze weights*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ *// update*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ *// update*
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$ *// set pruned weights to 0*

With the weight freezing interpretation discussed at the end of Section 3.3.3, the `SparseGPT` reconstruction can be cast in the column-wise greedy framework of the quantization algorithm GPTQ. This means we can also inherit several algorithmic enhancements from GPTQ, specifically: precomputing all the relevant inverse Hessian sequence information via a Cholesky decomposition to achieve numerical robustness and applying lazy batched weight matrix updates to improve the compute-to-memory ratio of the algorithm. Our adaptive mask selection, as well as its extensions to semi-structured pruning, are compatible with all of those extra techniques as well.

Algorithm 3.3 presents the the unstructured sparsity version of the SparseGPT algorithm in its fully-developed form, integrating all the relevant techniques from GPTQ.

Joint Sparsification & Quantization

Algorithm 3.3 operates in the column-wise greedy framework of GPTQ, thus sharing the computationally heavy steps of computing the Cholesky decomposition of $\mathbf{H}^{-1}$ and continuously updating $\mathbf{W}$. This makes it possible to merge both algorithms into a single joint procedure. Specifically, all weights that are frozen by SparseGPT are additionally quantized, leading to the following generalized errors to be compensated in the subsequent update step:

$$\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{M}_{:,j} \cdot \text{quant}(\mathbf{W}_{:,j})) \,/\, [\mathbf{H}^{-1}]_{jj}, \tag{3.16}$$

where $\text{quant}(\mathbf{w})$ rounds each weight in $\mathbf{w}$ to the nearest value on the quantization grid. Crucially, in this scheme, sparsification and pruning are performed *jointly* in *a single pass* at essentially no extra cost over SparseGPT. Moreover, doing quantization and pruning jointly means that later pruning decisions are influenced by earlier quantization rounding, and vice-versa. This is in contrast to the joint techniques in Section 3.1, which first sparsify a layer and then simply quantize the remaining weights.

3.3.4 Experiments

Setup. We implement SparseGPT in PyTorch [PGM+19] and use the HuggingFace Transformers library [WDS+19] for handling models and datasets. All pruning experiments are conducted on *a single* NVIDIA A100 GPU with 80GB of memory. In this setup, SparseGPT can fully sparsify the 175-billion-parameter models in approximately 4 hours. Similar to Yao et al. [YAZ+22] and Section 3.2, we sparsify Transformer layers sequentially in order, which significantly reduces memory requirements. All our experiments are performed in one-shot, without finetuning, in a similar setup to recent work on post-training quantization of GPT-scale models [YAZ+22, DLBZ22] and Section 3.2. Additionally, in Appendix A.3.6 we investigate the real-world acceleration of our sparse models with existing tools.

For calibration data, we follow the protocol in Section 3.2.4 and use 128 2048-token segments, randomly chosen from the first shard of the C4 [RSR+20a] dataset. This represents generic text data crawled from the internet and makes sure that our experiments remain actually zero-shot since *no task-specific data is seen during pruning*.

Models, Datasets & Evaluation. We primarily work with the OPT model family [ZRG+22], to study scaling behavior, but also consider the 176 billion parameter version of BLOOM [SFA+22]. While *our focus lies on the very largest variants*, we also show some results on smaller models to provide a broader picture.

In terms of metrics, we mainly focus on *perplexity*, which is known to be a challenging and stable metric that is well suited for evaluating the accuracy of compression methods [YAZ+22, DZ23]. We consider the test sets of raw-WikiText2 [MXBS17] and PTB [MKM+94] as well as a subset of the C4 validation data, all popular benchmarks in LLM compression literature [YAZ+22, PPK+22, XLS+23]. For additional interpretability, we also provide ZeroShot accuracy results for Lambada [PKL+16], ARC (Easy and Challenge) [BPM+18], PIQA [TP03] and StoryCloze [MRL+17].

We note that the main focus of our evaluation lies on the *accuracy of the sparse models, relative to the dense baseline* rather than on absolute numbers. Different preprocessing may

influence absolute accuracy, but has little impact on our relative claims. The perplexity is calculated following precisely the procedure described by HuggingFace [Hug22], using full stride. Our ZeroShot evaluations are performed with GPTQ's implementation, which is in turn based on the popular EleutherAI-eval-harness [Ele22]. Additional evaluation details can be found in Appendix A.3.3. All dense and sparse results were computed with exactly the same code, available as supplementary material, to ensure a fair comparison.

Baselines. We compare against the standard magnitude pruning baseline [ZG17], applied layer-wise, which scales to the very largest models. On models up to 1B parameters, we compare also against AdaPrune [HCI+21], the most efficient among existing accurate post-training pruning methods. For this, we use the memory-optimized reimplementation of Frantar & Alistarh [FA22] and further tune the hyper-parameters provided by the AdaPrune authors. We thus achieve a $\approx 3\times$ speedup without impact on solution quality, for our models of interest.

Results

Pruning vs. Model Size. We first study how the difficulty of pruning LLMs changes with their size. We consider the entire OPT model family and uniformly prune all linear layers, excluding the embeddings and the head, as standard [SWR20, KCN+22], to 50% unstructured sparsity, full 4:8 or full 2:4 semi-structured sparsity (the 2:4 pattern is the most stringent). The raw-WikiText2 performance numbers are given in Table 3.14 and visualized in Figure 3.10. The corresponding results for PTB and C4 can be found in Appendix A.3.4 and show very similar trends overall.

OPT - 50%	125M	350M	1.3B
Dense	27.66	22.00	14.62
Magnitude	193.	97.80	1.7e4
AdaPrune	58.66	48.46	32.52
SparseGPT	**36.85**	**31.58**	**17.46**

OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	12.47	10.86	10.13	9.56	9.34	8.35
Magnitude	50%	265.	969.	1.2e4	168.	4.2e3	4.3e4
SparseGPT	50%	**13.48**	**11.55**	**11.17**	**9.79**	**9.32**	**8.21**
SparseGPT	4:8	14.98	12.56	11.77	10.30	9.65	8.45
SparseGPT	2:4	17.18	14.20	12.96	10.90	10.09	8.74

Table 3.14: OPT perplexity results on raw-WikiText2.

One immediate finding is that the accuracy of magnitude-pruned models collapses across all scales, with larger variants generally dropping faster than smaller ones. This is in stark contrast to smaller vision models which can usually be pruned via simple magnitude selection to 50% sparsity or more at very little loss of accuracy [SA20]. It highlights the importance of accurate pruners for massive generative language models, but also the fact that perplexity is a very sensitive metric.

For SparseGPT, the trend is very different: already at 2.7B parameters, the perplexity loss is ≈ 1 point, at 66B, there is essentially zero loss and at the very largest scale there is even

a slight accuracy improvement over the dense baseline, which however seems to be dataset specific (see also Appendix A.3.4). AdaPrune, as expected, also yields a big improvement over magnitude pruning, but is significantly less accurate than SparseGPT. Despite the efficiency of AdaPrune, running it takes approximately ≈ 1.3h on a 350M model and ≈ 4.3h on a 1.3B one, while SparseGPT can fully sparsify 66B and 175B models in roughly the same time, executing on the same A100 GPU.

In general, there is a clear trend of larger models being easier to sparsify, which we speculate is due to overparametrization. A detailed investigation of this phenomenon would be a good direction for future work. For 4:8 and 2:4 sparsity, the behavior is similar, but accuracy drops are typically higher due to the sparsity patterns being more constrained [HCI+21]. Nevertheless, at the largest scale, the perplexity increases are only of 0.11 and 0.39 for 4:8 and 2:4 sparsity, respectively.

Sparsity Scaling for 100+ Billion Parameter Models. Next, we take a closer look at the largest publicly-available dense models, OPT-175B and BLOOM-176B, and investigate how their performance scales with the degree of sparsity induced by either SparseGPT or magnitude pruning. The results are visualized in Figures 3.9 and 3.13.

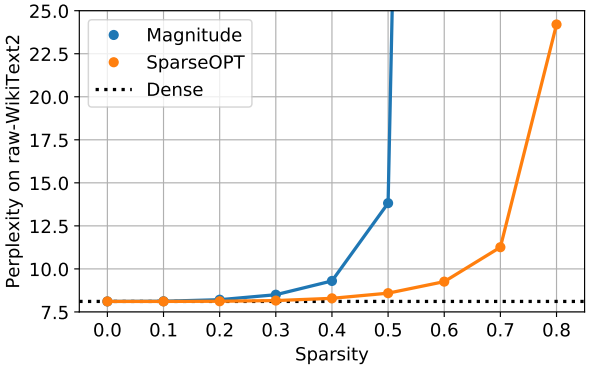


Figure 3.13: Uniform pruning BLOOM-176B.

For the OPT-175B model (Figure 3.9) magnitude pruning can achieve at most 10% sparsity before significant accuracy loss occurs; meanwhile, SparseGPT enables up to 60% sparsity at a comparable perplexity increase. BLOOM-176B (Figure 3.13) appears to be more favorable for magnitude pruning, admitting up 30% sparsity without major loss; still, SparseGPT can deliver 50% sparsity, a $1.66\times$ improvement, at a similar level of perplexity degradation. Even at 80% sparsity, models compressed by SparseGPT still score reasonable perplexities, while magnitude pruning leads to a complete collapse (>100 perplexity) already at 40/60% sparsity for OPT and BLOOM, respectively. Remarkably, SparseGPT removes around *100 billion weights* from these models, with low impact on accuracy.

ZeroShot Experiments. To complement the perplexity evaluations, we provide results on several ZeroShot tasks. These evaluations are known to be relatively noisy [DLBZ22], but more interpretable. Please see Table 3.15.

Overall, a similar trend holds, with magnitude-pruned models collapsing to close to random performance, while SparseGPT models stay close to the original accuracy. However, as expected, these numbers are more noisy: 2:4 pruning appears to achieve noticeably higher accuracy than the dense model on Lambada, despite being the most constrained sparsity

Method	Spars.	Lamb.	PIQA	ARC-e	ARC-c	Story.	**Avg.**
Dense	0%	75.59	81.07	71.04	43.94	79.82	**70.29**
Magnitude	50%	00.02	54.73	28.03	25.60	47.10	**31.10**
SparseGPT	50%	78.47	80.63	70.45	43.94	79.12	**70.52**
SparseGPT	4:8	80.30	79.54	68.85	41.30	78.10	**69.62**
SparseGPT	2:4	80.92	79.54	68.77	39.25	77.08	**69.11**

Table 3.15: ZeroShot results on several datasets for sparsified variants of OPT-175B.

pattern. These effects ultimately average out when considering many different tasks, which is consistent to the literature [YAZ+22, DLBZ22, DZ23].

Joint Sparsification & Quantization. Another interesting research direction is the combination of sparsity and quantization, which would allow combining computational speedups from sparsity [KKG+20, EDGS20] with memory savings from quantization [DLBZ22, DZ23] and Section 3.2. Specifically, if we compress a model to 50% sparse + 4-bit weights, store only the non-zero weights and use a bitmask to indicate their positions, then this has the same overall memory consumption as 3-bit quantization. Hence, in Figure 3.14 (right) we compare SparseGPT 50% + 4-bit with state-of-the-art GPTQ 3-bit numbers. It can be seen that 50% + 4-bit models are more accurate than their respective 3-bit versions for 2.7B+ parameter models, including 175B with 8.29 vs. 8.68 3-bit. We also tested 2:4 and 4:8 in combination with 4-bit on OPT-175B yielding 8.55 and 8.85 perplexities, suggesting that 4bit weight quantization only brings an ≈ 0.1 perplexity increase on top semi-structured sparsity.

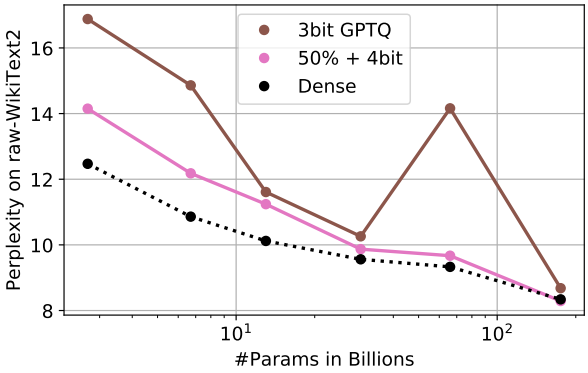


Figure 3.14: Comparing joint 50% sparsity + 4-bit quantization with size-equivalent 3-bit on the OPT family for $\geq$ 2.7B params.

Sensitivity & Partial N:M Sparsity. One important practical question concerning n:m pruning is what to do when the fully sparsified model is not accurate enough? The overall sparsity level cannot simply be lowered uniformly, instead one must choose a subset of layers to n:m-sparsify completely. We now investigate what a good selection is in the context of extremely large language models: we assume that 2/3 of the layers of OPT-175B/BLOOM-176B should be pruned to 2:4 sparsity and consider skipping either all layers of one type (attention, fully-connected-1, fully-connected-2) or skipping one third of consecutive layers (front, middle, back). The results are shown in Figure 3.15.

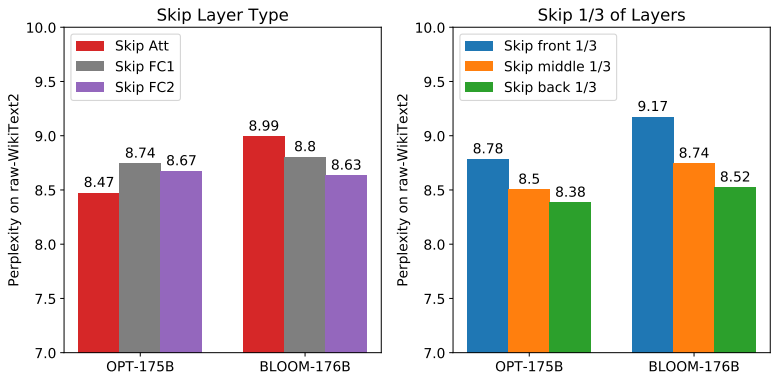


Figure 3.15: Sensitivity results for partial 2:4 pruning.

While the sensitivity of layer-types differs noticeably between models, there appears to be a clear trend when it comes to model parts: *later layers are more sensitive than earlier ones*; skipping the last third of the model gives the best accuracy. This has a very practical consequence in that, due to the sequential nature of `SparseGPT`, we can generate a sequence of increasingly 2:4 sparsified models (e.g. $1/2$, $2/3$, $3/4$, ...) in *a single pruning pass* by combining the first x layers from a `SparseGPT` run with the last $n_{\text{layers}} - x$ of the original model. The accuracy of such model sequences are shown in Appendix A.3.5.

Kernel for Memory-Bound Inference. Although it is very challenging to achieve *computational speedup* from sparsity patterns without direct hardware support on modern GPUs, it is possible to *accelerate memory loading* and thus achieve speedup in the batchsize 1 regime similar to weight-only quantization in Section 3.2.4. We demonstrate this by implementing a custom GPU kernel which assumes bitmask storage of the sparsity mask plus dense storage of the non-zero weights. Crucially, we further assume an n:m pattern with large m, e.g., 16:32, 16:64 or 16:128. This usually yields similar accuracy as fully unstructured pruning, but greatly simplifies kernel design as it enables natural partition of the problem into corresponding blocks. Our kernel first fetches activations and a non-zero weight tile into shared memory. Then it loads and decodes the corresponding bitmask, indexing into shared memory to read non-zero weights as it encounters corresponding 1-bits; importantly, this process is bank-conflict free. As demonstrated by Table 3.16, this kernel achieves substantial speedup on an NVIDIA RTX A6000 GPU. At low sparsity, acceleration is essentially ideal while higher rates run into diminishing returns. This is primarily due to the fact that our current sparsity mask decoding implementation has a constant overhead, irrespective of the sparsity level. Finally, we note that this kernel was originally written for a follow-up project to SparseGPT about further enhancing accuracy via finetuning [KKF+23].

Pattern	16:32	16:64	16:128
Speedup	1.82×	2.89×	3.42×

Table 3.16: Batchsize 1 generation speedups for custom kernel on a 4K × 12K matrix.

3.3.5 Related Work

Pruning Methods. To our knowledge, we are the first to investigate pruning of massive GPT-scale models, e.g. with more than 10 billion parameters. One justification for this

surprising gap is the fact that most existing pruning methods, e.g. [HMD16, GEH19, KA22], require *extensive retraining* following the pruning step in order to recover accuracy, while GPT-scale models usually require massive amounts of computation and parameter tuning both for training or finetuning [ZRG+22]. SparseGPT is a *post-training* method for GPT-scale models, as it does not perform any finetuning. So far, post-training pruning methods have only been investigated at the scale of classic CNN or BERT-type models [HCI+21, KKM+22], which have 100-1000x fewer weights than our models of interest. We discussed the challenges of scaling these methods, and their relationship to SparseGPT, in Section 3.3.2.

Post-Training Quantization. By contrast, there has been significant work on post-training methods for *quantizing* open GPT-scale models [ZRG+22, SFA+22]; see Section 3.2.2 for a overview of early research in this area. More recently, Follow-up work [XLS+23] investigated joint activation and weight quantization to 8 bits, proposing a smoothing-based scheme which reduces the difficulty of activation quantization and is complemented by efficient GPU kernels. Park et al. [PYNC22] tackle the hardness of quantizing activation outliers via *quadapters*, learnable parameters whose goal is to scale activations channel-wise, while keeping the other model parameters unchanged. Dettmers & Zettlemoyer [DZ23] investigate scaling relationships between model size, quantization bits, and different notions of accuracy for massive LLMs, observing high correlations between perplexity scores and aggregated zero-shot accuracy across tasks. As we have shown in Section 3.3.3, the SparseGPT algorithm can be applied in conjunction with GPTQ, the current state-of-the-art algorithm for weight quantization, and should be compatible with activation quantization approaches [XLS+23, PYNC22].

3.3.6 Discussion

We have provided a new post-training pruning method called SparseGPT, specifically tailored to massive language models from the GPT family. Our results show for the first time that large-scale generative pretrained Transformer-family models can be compressed to high sparsity via weight pruning in *one-shot, without any retraining*, at low loss of accuracy, when measured both in terms of perplexity and zero-shot performance. Specifically, we have shown that the largest open-source GPT-family models (e.g. OPT-175B and BLOOM-176B) can reach 50-60% sparsity, dropping more than 100B weights, with low accuracy fluctuations.

In terms of limitations, we focus primarily on uniform per-layer sparsity but non-uniform distributions are a promising topic for future work. Further, SparseGPT is currently not quite as accurate on smaller and medium sized variants as on the very largest ones. We think this may be addressable through careful partial or full finetuning, which is starting to become feasible at the scale of such models up to a few billion parameters. Finally, while we study the sparsification of pretrained foundation models in this work, we think investigating how additional post-pretraining techniques like instruction tuning or reinforcement learning with human feedback interact with compressibility will also be an important future research area.

Overall, our work shows that the high degree of parametrization of massive GPT models allows pruning to directly identify sparse accurate models in the "close neighborhood" of the dense model, without gradient information. Remarkably, the output of such sparse models correlates extremely closely with that of the dense model. We also show that *larger models are easier to sparsify*: at a fixed sparsity level, the relative accuracy drop for the larger sparse models narrows as we increase model size, to the point where inducing 50% sparsity results in practically no accuracy decrease on the largest models, which should be seen as very encouraging for future work on compressing such massive models.

CHAPTER 4

Systems

The previous chapter described how to design fast and accurate algorithms for significantly compressing very large machine learning models. However, actually translating theoretical compression into practical benefits is a challenging problem as well, even more so since uncompressed Transformer software is, due its importance, already extremely well optimized.

This chapter will first introduce Marlin, a mixed-precision inference kernel that achieves near-optimal GPU utilization across a wide range of batchsizes, significantly enhancing the practicality of weight-only quantization (via, e.g., GPTQ). Afterwards, we will turn our attention towards scaling up the post-training compression methodology introduced in Chapter 3 by another order of magnitude, to trillion-parameter Mixture-of-Expert models. In this context, even higher theoretical compression rates can be achieved. While this involves complex, initially not at all hardware-friendly, encoding, we show how sub 1-bit compression rates can actually be made practical by careful co-design of a compression scheme and a corresponding GPU inference kernel.

Section 4.1.3 is based on the Marlin GitHub repository [FA24a] and parts of the paper preprint "MARLIN: Mixed-Precision Auto-Regressive Parallel Inference on Large Language Models" [FCC+24], while Section 4.2 is derived from the MLSys 2024 paper "QMoE: Practical Sub-1-Bit Compression of Trillion Parameter Models" [FA24b].

4.1 Marlin: A Near-Optimal Mixed-Precision Kernel

4.1.1 Motivation

In Section 3.2.4, we demonstrated how weight-only quantized models can bring major inference speedups during generation, by reducing the amount of memory that needs to be transferred to registers, the primary runtime bottleneck in this application. However, our respective kernel was designed only for batchsize 1 inference, i.e., one-token-at-a-time. As the adoption of LLMs is rapidly growing, this is becoming unrealistic in a practical serving setting. Further, even local inference is moving away from the batchsize 1 setup due to the emergence of new techniques like speculative decoding [LKM23] or top-k voting [RST+24]. While there have been some attempts at writing efficient kernels for larger batchsizes [LZW+23, DPHZ23], their speedups relative to uncompressed inference drop rapidly even if the batchsize is only increased a little, say from 1 to 4.

In theory, modern GPUs have very large FLOPs/Bytes ratios, meaning that they can execute floating point operations much faster than they can read bytes from memory. As an example, an A10 GPU has a FLOPs/Bytes ratio of ≈ 200 [NVI22]. Consequently, if the weights are stored in 4-bit precision and every multiply-accumulate counts for 2 FLOPs, inference should theoretically be memory-bound up until batchsize $b_{\text{opt}} \approx 50$. (Processing one input token takes 2 FLOPs per weight and the GPU can execute 100 FLOPs in the time it takes to load one 4-bit weight. Hence, memory loading will dominate runtime as long as the input batchsize is less than 50.) In fact, b_{opt} is the batchsize where latency is neither bound by memory nor by compute, i.e., where we achieve the *lowest latency at maximum throughput*. In principle, this is precisely the batchsize that we would like to operate at in practice: any smaller does not yield speedups and any larger does not improve throughput. However, actually implementing such a mixed-precision (we will focus on FP16-INT4 here) matmul kernel which fully maximizes essentially all GPU resources (compute and memory) simultaneously is a major challenge. In the following section, we will try to come as close as possible to this goal by designing *Marlin*, an extremely optimized *Mixed-precision Auto-Regressive LINear* kernel.

4.1.2 Kernel Design

In what follows, we will aim to implement a matrix multiplication $\mathbf{C} = \mathbf{AB}$ where $\mathbf{A}$ is of shape $m \times k$, $\mathbf{B}$ of shape $k \times n$ and $\mathbf{C}$ of shape $m \times n$. Further, $\mathbf{A}$ is in full FP16 precision, while $\mathbf{B}$ has been (symmetrically) quantized to INT4, either with one FP16 scale per output n, shared between all elements of the corresponding column, or one scale per g consecutive weights in each column, for $(k/g)n$ scales in total. Our kernel is designed for Ampere-class GPUs and will take full advantage of key features specific to this architecture [NVI20].

Bound By Weight Loading. Executing our target matmul requires, in theory, touching exactly $16mk + 4kn + 16mn$ bits of memory (reading both operands and writing the results) while executing exactly mkn multiply-accumulate operations, each counted as 2 FLOPs. If m is small, say $\approx 16 - 32$, our problem has very low arithmetic intensity (assuming reasonable weight matrix $\mathbf{B}$ shapes). Consequently, as FLOPs/Bytes ratios on modern GPUs are generally > 100, it should be completely bound by the cost of reading the quantized weights $\mathbf{B}$ from global GPU memory.

This holds theoretically, while we actually need to organize computation very carefully for this to remain true in practice. In contrast to the previously studied [FAHA23, DZ23] $m = 1$ case, where both $\mathbf{A}$ and $\mathbf{C}$ are tiny, inputs and outputs now actually have non-negligible size, especially since those operands have $4\times$ higher bit-width than our weights. Hence, it is quite easy to end up with a kernel where activation loading significantly impacts runtime: e.g., assume that for every 16×64 tile from $\mathbf{B}$, we reload one 16×16 tile from $\mathbf{A}$ (in global memory), then both weight and activation loading will contribute equally to the runtime. The remedy to this is to widen $\mathbf{A}$ tiles, but doing so by too much will partition n only into very few segments. This in turn means that we also need to split our problem across the k dimension in order to produce sufficiently many sub-problems to saturate most of the GPU's SMs. Unfortunately, if there are too many k splits, we will require quite a few reduction steps, involving repeated global reads and writes, noticeably increasing the number of overall memory accesses and consequently the kernel runtime.

Fortunately, there is one mechanism we can exploit to work around the above problems: the *L2 cache*. This cache usually has 10s of MBs capacity [NVI20] and $2 - 4\times$ higher bandwidth than global memory; for simplicity of discussion, we will assume $4\times$. The size of the cache is

easily enough to store (significant parts of) $\mathbf{A}$, and so loading $\mathbf{A}$ fragments will take only $1/4$ of the time it takes to load $\mathbf{B}$ fragments in the 16×16 and 16×64 example from above. Further, since a GPU can load from L2 to L1 and from global to L2 simultaneously (any global load must pass through the entire cache hierarchy), we can pipeline these loads and essentially hide the bandwidth cost of the $\mathbf{A}$ fragment access completely, as long as the overall required memory traffic (for weights + activations) does not exceed the L2 bandwidth.

Consequently, we will proceed by partitioning $\mathbf{C}$ into tiles of size $m \times n_c$ with $n_c \in \{64, 128, 256\}$, i.e. moderately wide tiles of full input batchsize m, and then assigning each corresponding independent matmul sub-problem to one SM (for now, we assume that this leads to good utilization, we will handle the common case where this does not happen later). At $n_c = 256$, even batchsize $m = 64$ remains bound by global weight loading since the weight/activation load ratio is 1 and the L2 bandwidth $\geq 2\times$ larger than the global one.

Maximizing Loading Bandwidth. Since our kernel will be memory-bound, its runtime will be determined by how much of the theoretical bandwidth we are actually able to achieve. Consequently, we always want to utilize the widest loads possible. On current GPUs, this is 128 bits = 16 bytes per thread. This means one warp can load $32 \times 32 = 1024$ INT4 weights with a single instruction. To reach peak efficiency, we need to have 8 threads each in a warp read 128 bytes of contiguous memory (assuming 128-byte-aligned addresses), a full cache line. Achieving this for activation tiles of shape $m \times k_a$ mandates a tile k_a of at least 64. Since the weights are static during inference and can thus be preprocessed offline, we simplify things by reshuffling 16×64 tiles (1024 weights in total) so that they are laid out contiguously in memory and are thus trivial to load optimally (this also simplifies the corresponding index calculations).

While we continuously reload $\mathbf{A}$ fragments from L2 cache, each element of $\mathbf{B}$ is accessed exactly once. Nevertheless, every read will be put into the L2 cache as well, potentially evicting parts of $\mathbf{A}$ that are still needed by some SMs. To avoid such cache pollution, we can supply the `cp.async` instruction with an `evict_first` cache-hint, ensuring that unnecessarily stored $\mathbf{B}$ data is dropped before any other cache line, which may actually still be required.

Shared Memory Layouts. Overall, we will load both weights and activations asynchronously via Ampere's `cp.async` instruction from global (or L2) to shared memory; this requires no temporary registers and also makes overlapping these loads with computation much easier. Due to our offline preprocessing of $\mathbf{B}$, we can simply copy to shared memory in contiguous fashion, avoiding any bank conflicts. In contrast, handling the $\mathbf{A}$ fragments requires a lot more care: specifically, we need to ensure that the 16-byte vectors corresponding to indices ij, $(i+8)j$, $i(j+1)$ and $(i+8)(j+1)$ of each 16×16 FP16 activation block are stored in different memory banks. Only then can `ldmatrix.sync` instructions, which load $\mathbf{A}$ operand data and distribute it across warp threads to prepare for tensor core use, execute in conflict-free manner. Note that this is not the case for, e.g., ij and $(i+1)j$ since the offset between those indices is exactly 128 bytes (there are 32 banks of 4-byte width). This can be achieved by storing 16-byte element ij in an activation tile at location $i(i \oplus j)$ in the corresponding shared memory tile, where $\oplus$ denotes the XOR operation [NVI24a]. Another key aspect of this index transformation is that if a warp reads a contiguous sub-tile of the global $\mathbf{A}$ tile (e.g., the first 4 rows), then it will be written permuted but still overall contiguously into shared memory. Although undocumented, this appears to be necessary in order to avoid bank conflicts on writing, judging by the NVIDIA profiler. These index calculations are somewhat complex and

potentially slow to take care of dynamically; however, as they only affect a relatively small number of shared memory locations, which remain static throughout the main loop, we can precompute them in registers, accompanied by appropriate unrolling (see below).

Memory Load Pipelining. The key to simultaneously reaching close to maximum bandwidth and close to maximum compute is to fully overlap memory loading and tensor core math. For global to shared memory loads, this can be achieved via `cp.async` operations, in every iteration prefetching the $\mathbf{A}$ and $\mathbf{B}$ tiles which will be used $d-1$ steps in the future, where d is the pipeline depth (we need one more buffer for the current tile). Additionally, we can prefetch the next sub-tile from shared memory (most GPU operations do not block until they hit a dependency) before accumulating the current partial matmul, for which the operands were already fetched to registers in the previous iteration—this technique is also called *double buffering* [NVI24b]. We pick a pipeline depth of $d = 4$ for two reasons: (a) this seemed sufficient in all of our tests to completely hide latency while fitting into shared memory even for $m = 64$, and (b) because it is evenly divisible by 2. The latter is crucial as it allows us to smoothly unroll across the full pipeline since after d iterations both the pipeline and the register buffer index will always have the same value of 0. This unrolling makes all shared memory addressing completely static, avoiding slow transformed index calculations (see above) by using some of the extra registers that we have available. Finally, we would like to note that this also seemed to be the most reliable way to make the CUDA compiler correctly order instructions to enable actual double buffering. Figure 4.1 visualizes the several layers of pipelining used by the Marlin kernel.

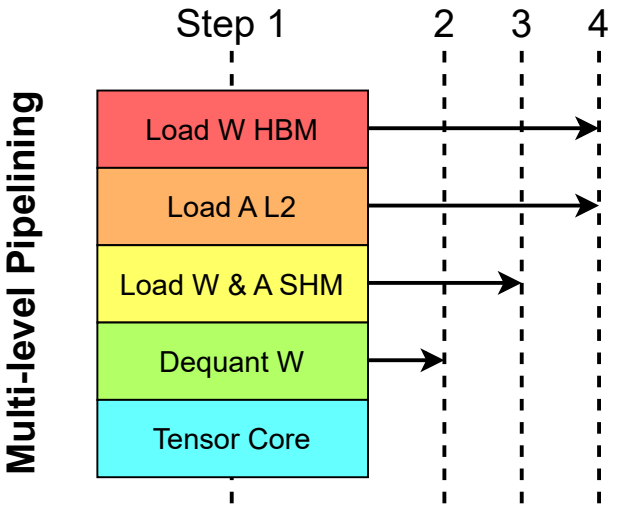


Figure 4.1: Several levels of pipelining in the Marlin kernel.

Warp Layout. As discussed previously, each SM will compute an $m \times n_c$ fragment of the output $\mathbf{C}$ by repeatedly fetching $m \times k_a$ blocks from $\mathbf{A}$ and $k_a \times n_c$ tiles from $\mathbf{B}$ and then accumulating those partial matmuls into the result, stored in registers—only after the entire column slice in $\mathbf{B}$ is processed, it will be written back out into global memory. This computation must further be subdivided across warps: in the most straight-forward fashion, each warp would compute an $m \times (n_c/\#\text{warps})$ tile of the output. In order to reach peak

compute throughput, we would like to use at least 4 (as Ampere GPUs have 4 warp schedulers) and ideally 8 warps (to have a bit more latency hiding) [SLG+22]. However, this leads to rather small tile sizes, especially at smaller n_c. This is not just problematic for our memory reshuffling discussed above but also hinders tensor core throughput since a small tile-width brings more sequential dependencies (as those consecutive operations must use the same accumulators) into tensor-core operations, which can cause stalls. Instead, we fix the sub-tile width of each warp to 64 and further split the computation across k_a; Figure 4.2 illustrates such a warp layout and Algorithm 4.1 provides corresponding pseudo-code. Consequently, multiple warps will accumulate partial results of the same output sub-tile in registers. These must then eventually be reduced in shared memory before the final write-out, but this can be done via a logarithmic parallel reduction [H+07], and thus typically causes minimal overhead.

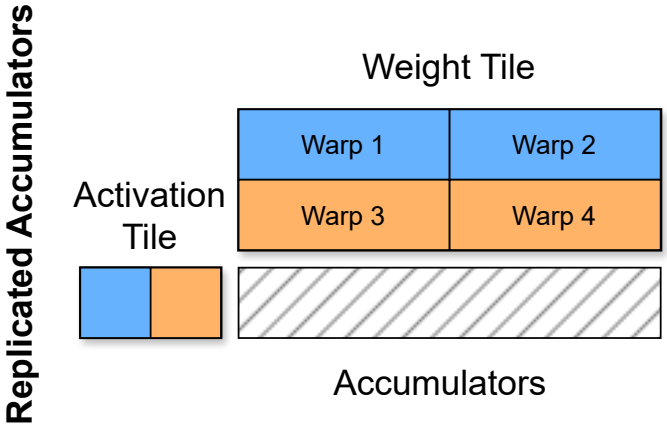


Figure 4.2: Illustration of Marlin's warp layout. Multiple warps accumulate partial results of the same output tile; see also Algorithm 4.1 for corresponding pseudocode.

Algorithm 4.1: Warp reduction within a corresponding $\mathbf{A}$ and $\mathbf{B}$ tile. The following pseudo-code is executed by all warps, identified via "warp_idx".

split $\mathbf{A}$ tile into $1 \times k$ sub-tiles, indexed via $[i, j]$
split $\mathbf{B}$ tile into $k \times n$ sub-tiles, indexed via $[i, j]$
$i \leftarrow \lfloor \text{warp\_idx}/n \rfloor$
$j \leftarrow \text{warp\_idx} \bmod n$
$\mathbf{C} \leftarrow$ all zeros (if $\mathbf{A}$ and $\mathbf{B}$ tiles are the first)
while $i < k$ **do**
  $\mathbf{C} \leftarrow \mathbf{C} + \mathbf{A}[0, i]\mathbf{B}[i, j]$
  $i \leftarrow i + (\#\text{warps}/n)$
end while
parallel reduce $\mathbf{C}$ across j (if $\mathbf{A}$ and $\mathbf{B}$ tiles are the last)

Dequantization and Tensor Cores. Another critical aspect of any mixed-precision kernel is datatype conversion, in our case, converting INT4 to FP16. Doing this naively via type-casts is very slow. Fortunately, we can achieve the same result with clever binary manipulations—we follow a modified version of the approach suggested by [KHFA22]. We now illustrate this

procedure at the hand of the most simple case: converting the INT4 located at positions $12-15$ in an INT16 to a signed FP16 value. First, we extract just the bits corresponding to our INT4 (via an AND of a mask) and turn bits $1-7$ of the result into 0110010 (with an OR); this can be accomplished in a single lop3 instruction, which we however seemingly need to emit explicitly. Now, we have an FP16 number with an exponent of 50 and the last 4 mantissa bits corresponding to our conversion target. Consequently, subtracting the FP16 value with exponent 50 and mantissa 0, will give us the floating point representation of exactly our 4 target bits, unsigned. To make this value signed, we further have to subtract 8, which we can however fuse directly into the last 3 bits of the total value we subtract. A similar strategy also works for decoding the INT4 located at positions $8-11$; for the values at $0-3$ and $4-7$, we need to execute a shift, in order to bring those values into the mantissa bits of the reinterpreted float. Modern GPUs can simultaneously compute with two separate 16-bit operands packed into a single 32-bit register. Hence, we can efficiently dequantize two INT4s in an INT32 at the same time, using the just described procedure. Finally, we want to dequantize directly into the right register layout for subsequent tensor core calls. To do this, we again take advantage of the fact that $\mathbf{B}$ can be preprocessed offline and reorganize weights such that the 16-byte vector read by each thread contains precisely its necessary 8 quantized weights of 4 separate 16×16 tensor core blocks. Additionally, within an INT32, weights are stored interleaved, according to the pattern 64207531, to power the previously mentioned parallel decoding.

At the innermost level, we accumulate the results of an $m \times 16$ times 16×64 matmul. We execute this accumulation column-wise, emitting 16×16 times 16×8 tensor core `mma.sync` instructions. This has the advantage over row-wise execution that we can pipeline the dequantization of the next $\mathbf{B}$ operand with the tensor core math of the current column; note that the tensor core and int/float math used for dequantization feature different GPU pipelines that can be engaged at the same time. While proper ordering of instructions is important here for good performance, this part makes a simple enough loop construction that the compiler seems to be able to figure this out well without too much extra care.

Groups and Instruction Ordering. So far, we have completely disregarded the quantization scale factors. For per-output quantization, we can simply scale the final output once before the global write-out. An interesting observation we have made in this context is that despite these loads not being asynchronous to any computation, it is still critical to perform them via `cp.async` followed by an immediate `wait_group` instruction to avoid unfavorable main loop instruction reordering by the compiler.

With grouped quantization, which is crucial to maintain the best possible accuracy, we have to load and apply scaling during the main loop. First, we reorganize scale storage in a similar way as quantized weights (see above), such that the scales required by the same type of thread, for different 16×16 blocks are packed together and can be loaded from shared memory as a single 16-byte vector (we note that multiple threads will need to load the same scale package due to the tensor core layout). In principle, for group-size 128 and a $\mathbf{B}$ tile shape of 64×256 (a common setting), we only need to global and shared memory load new scales *once every other tile* (and here only once during the first sub-tile). However, it appears that the compiler is rather brittle to such irregularities in perhaps the most critical section of the code, leading to unfavorable instructions orderings with $10-20\%$ overall slow-down in some shape settings. Instead, we find that reloading scales from shared memory for *every sub-tile* (but there are no redundant global loads) maintains peak performance. Doing this adds some technically unnecessary shared memory loads, but there is sufficient extra bandwidth to support this at

no overhead, while it otherwise preserves the compiler's well pipelined instruction ordering for non-grouped quantization.

Striped Partitioning. With all the techniques explained in the previous several paragraphs, we can reach near optimal compute and bandwidth performance, provided matrices are large and can be *perfectly partitioned* across all SMs over the n axis. In practice, this is rarely the case. The standard remedy in such a situation is to also partition across the k dimension, but for many popular layer shape and GPU combinations we would need a lot of additional splits to reach an even distribution without significant wave quantization. This in turn adds many global reduction steps with additional overhead. Instead, we opt for a *striped* partitioning scheme where the "stripes" of $\mathbf{B}$ processed by an SM may span across multiple $\mathbf{C}$ tiles (see also Figure 4.3). Concretely, we first determine the number of $\mathbf{B}$ tiles to be processed by each SM $T = \lceil \#\text{tiles}/\#\text{SMs} \rceil$ and then assign (up to) T tiles column-wise starting top-left. Crucially, if we reach the bottom of a tile column but the current SM does not yet own T tiles, we proceed by assigning tiles from the top of the next tile column; in other words, stripes can span across multiple columns. This ensures a roughly uniform distribution of tiles across all SMs, while minimizing the number of required global reduction steps. Overall, this strategy is similar to stream-k partitioning [OMC+23].

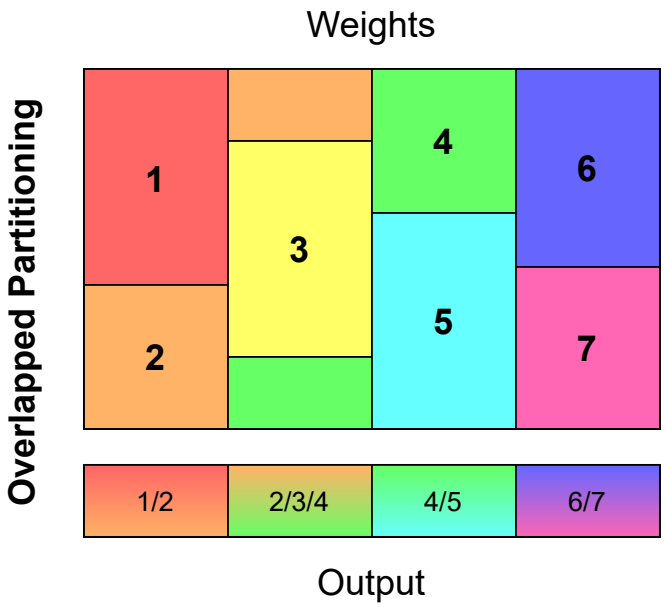


Figure 4.3: Illustration of Marlin's striped partitioning scheme.

We implement the global reduction between stripes of the same tile column serially, from bottom to top. The latter approach is most efficient since the bottom-most SM will have its results fastest and the top-most slowest in the presence of any column spill-over. We perform the reduction in FP16 directly in the output buffer to maximize L2 cache hits and thus minimize any global read overheads. This also keeps the operation essentially in-place, requiring only a small extra lock buffer for synchronization.

Finally, we note that for batchsizes $\gg 64$, we can virtually replicate $\mathbf{B}$ for the striped index calculations, followed by a modulo operation to move back into the original matrix, and

advance the $\mathbf{A}$ pointer to the corresponding size-64 input batch segment. This results in significantly less global reductions for large input batchsizes (as occur during LLM prefills) and noticeably improves compute throughput in this setting.

4.1.3 Benchmarks

We first compare the performance of Marlin with other popular 4-bit inference kernels, on a large matrix that can be ideally partioned on an NVIDIA A10 GPU. This allows all kernels to reach pretty much their best possible performance. All kernels are executed at groupsize 128 (however, we note that scale formats are not 100% identical).

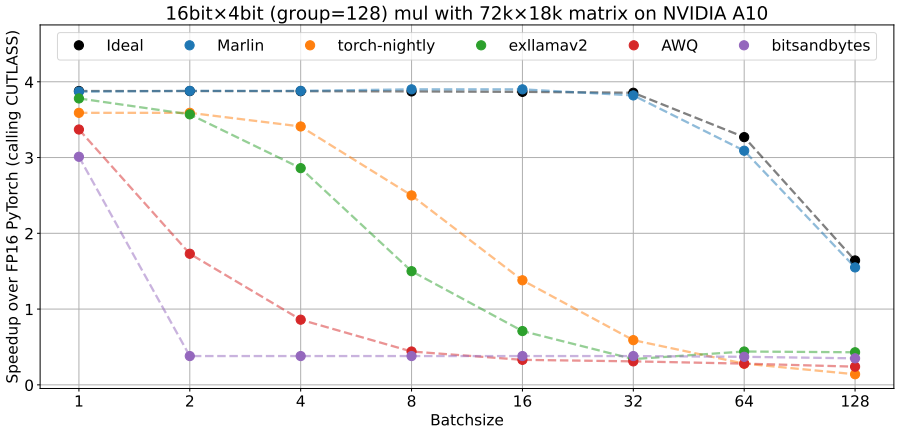


Figure 4.4: Peak performance of Marlin compared with other popular open-source kernels.

While existing kernels achieve relatively close to the optimal 3.87x (note the 0.125 bits storage overhead of the group scales) speedup at batchsize 1, their performance degrades quickly as the number of inputs is increased. In contrast, as can be seen in Figure 4.4, Marlin delivers essentially ideal speedups at all batchsizes, enabling the maximum possible 3.87x speedup up to batchsizes around 16-32.

Due to its striped partioning scheme, Marlin brings strong performance also on real (smaller) matrices and various GPUs. This is demonstrated by the results in Figure 4.5, where we benchmark, at batchsize 16, the overall runtime across all linear layers in Transformer blocks of popular open-source models.

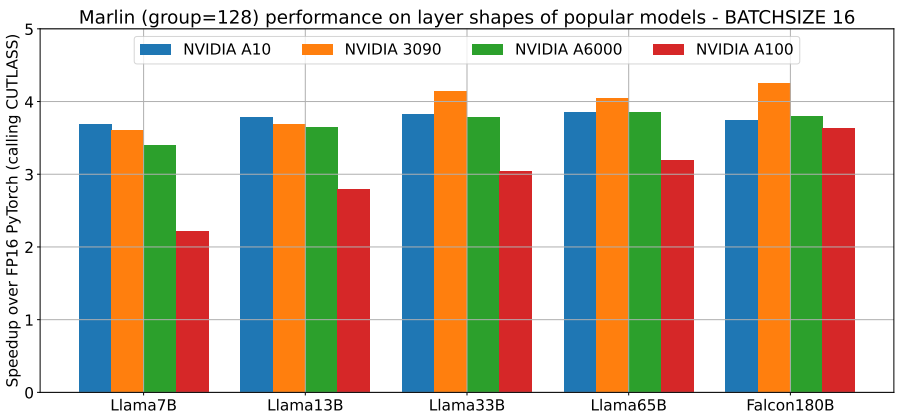


Figure 4.5: Marlin performance across real layer shapes of popular models.

Next, we also study what performance can be sustained over longer periods of time, at locked base GPU clock. Interestingly, we find that reduced clock speeds significantly harm the relative speedups of prior kernels, but have no effect on Marlin's virtually optimal performance (relative to the lower clock setting). This can be observed in Figure 4.6.

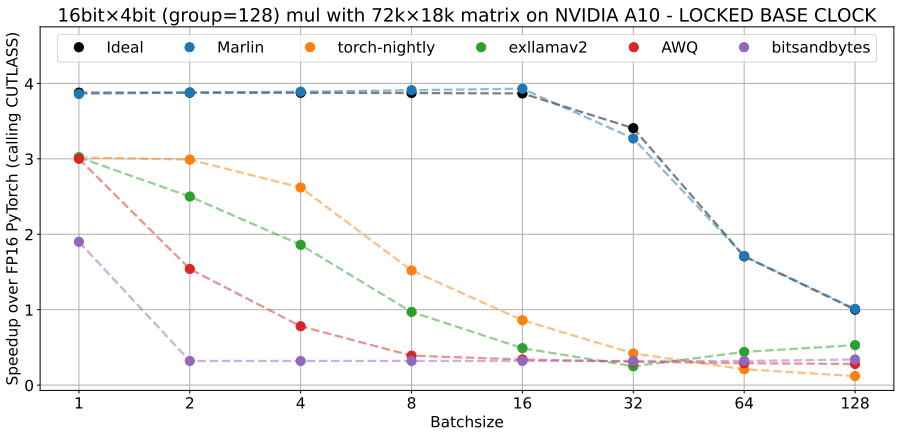


Figure 4.6: Sustained performance of Marlin compared with other popular open-source kernels.

Finally, we test how Marlin performs on very large inputs while running on a powerful GPU like the A100. While our primary goal was achieving the best possible generation performance at medium batchsizes, good prefill performance is still relevant in practice. As can be seen in Table 4.1, Marlin performs essentially identical to an uncompressed compute-bound matmul up to batchsize 1024, with only $\approx 10\%$ slow-down at even larger input shapes. At these massive m values, some of the optimizations designed specifically for our main medium m setting become slightly suboptimal; we leave optimizing for this scenario to future work.

batchsize	1024	2048	4096	8192
speedup	0.99	0.91	0.91	0.88

Table 4.1: Marlin performance on an 8K $\times$ 8K matrix for very large batchsizes.

GPTQ Modifications

The quantization format used by Marlin, designed for peak inference efficiency, is slightly different than the one used in Section 3.2. Hence, we quickly validate that GPTQ still produces highly accurate models with Marlin settings. For that purpose, we also integrate two small improvements into GPTQ: (a) picking group scales by searching for optimal group-wise clipping thresholds similar to [LTT+24], and (b) supporting calibration sequences of variable length. Figure 4.7 shows that Marlin-quantized models are $\approx 3.33\times$ smaller at the same perplexity as the uncompressed baseline. While this is not lossless (the ideal gain would be $3.87\times$ in this setup), it is an extremely practical improvement, especially given Marlin's highly efficient inference performance.

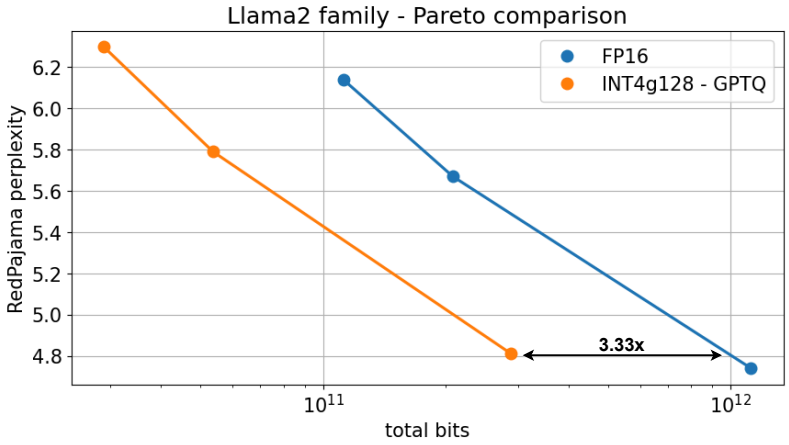


Figure 4.7: Pareto-curve of Llama2 models quantized to the Marlin format via GPTQ.

4.2 QMoE: Practical Sub-1-Bit Compression of Trillion Parameter Models

4.2.1 Motivation & Overview

Generative large language models (LLMs), e.g. [RWC+19, BMR+20, TLI+23, TMS+23], have garnered significant industrial and popular attention due to their surprising performance across many practical language and reasoning tasks. Yet, a major obstacle to broad deployment is given by their extremely high inference costs. One particularly promising approach for reducing these costs is the use of *Mixture-of-Experts (MoE)* architectures, e.g. [CDLCG+22, DHD+22, ZBK+22], whose general idea is to replicate certain model components many times while routing each input *only to a small subset of those replicas*. Through expert "specialization" to input subsets, MoEs achieve faster inference for the same model quality, but with significantly higher memory requirements due to components being replicated hundreds or even thousands of times, for the largest and best-performing models.

For example, the popular SwitchTransformer family [FZS22], on which we focus in this study, uses between 128 and 2048 experts (layer replicas) to significantly outperform standard dense T5 models [RSR+20b] in terms of inference and training costs, at equivalent model accuracy. Artetxe et al. [ABG+22] report similar improvements, on different tasks, for 512 experts. However, these results come at the cost of dramatic increases in model size: the largest SwitchTransformer has 1.6 trillion parameters, requiring 3.2TB of storage in standard half-precision, and correspondingly requires a hundred or more expensive (GPU or TPU) accelerators for efficient usage. This not only makes practical deployment costly and challenging, but also strongly limits research on such models.

Challenges. It is natural to ask whether the truly massive memory costs of such MoEs can be reduced via techniques for *model compression*, such as quantization [GKD+21] or sparsity [HABN+21], without significant accuracy loss. Achieving this would require overcoming conceptual and technical barriers:

1. Conceptually, existing *post-training/one-shot* compression methods, whose costs would be low enough to execute on such models, are only able to reduce precision to 3 or 4 bits per parameter [FAHA23, DZ23, WYH23] or around 50% sparsity [FA23], before

significant accuracy loss occurs. Yet, making trillion-parameter MoEs practical would require compression rates between $10\times$ and $20\times$ relative to 16-bit precision, i.e., on average *less than 1 bit per parameter*.

2. A key practical issue is *scaling*: applying state-of-the-art compression methods, designed for large dense models, to MoEs that are an order of magnitude larger, while maintaining affordability, runs into a plethora of memory, performance and reliability roadblocks.

3. Actually achieving *sub-1-bit* compression in practice would require a novel customized compression format. Such a format would also need to come with decoding algorithms that are highly-efficient on accelerators such as GPUs, in order to run inference on compressed models without major processing slowdowns.

Contribution. In this section, we overcome these challenges, and introduce QMoE, a framework for accurate compression and fast inference over massive MoEs, reducing model sizes by 10–20$\times$, to less than 1 bit per parameter. QMoE is specifically designed to compress and subsequently inference with models like the 1.6 trillion parameter SwitchTransformer-c2048, using only modest computational resources.

Our key technical contributions are a highly scalable compression algorithm implementation and a customized compression format designed together with bespoke GPU-kernels for fast on-the-fly decoding. Further, we show for the first time that accurate sub-1-bit compression of trillion parameter MoEs is feasible and can be achieved via affordable retraining-free compression techniques.

Concretely, we reduce the size of SwitchTransformer-c2048, the largest openly-available model, from 3.2TB in bfloat16 to less than 160GB in our customized compressed format, that is, ≈ 0.8 bits per parameter, at only a minor increase in loss on pretraining validation and zero-shot data. Using our QMoE kernels, this compressed model can then be executed fully, without any slow offloading, on commodity hardware such as $8\times$ NVIDIA RTX 3090 or $4\times$ NVIDIA A6000 GPUs, with $< 5\%$ runtime overhead relative to an idealized version of uncompressed execution, which would require $\approx 20\times$ more GPUs.

In summary, our work enables, for the first time, the performant execution of massive-scale MoE models on commodity hardware. This is illustrated by the fact that we are able to efficiently run the trillion-parameter SwitchTransformer-c2048 model on a single commodity GPU server, with minor accuracy loss. This addresses one of the key limitations behind MoE architectures, and should improve their practical adoption, as well as facilitate further research on understanding and improving such models.

4.2.2 Background

Mixture of Expert Models (MoEs)

The core idea behind Mixture of Expert models (MoEs) is to increase the number of parameters, and thus the network's modelling power, while at the same time keeping compute costs near-constant, relative to a standard feed-forward architecture. This is typically achieved by creating many copies of certain model components, each of which is responsible for processing only a subset of all input tokens. The corresponding input-to-component assignments are generally decided by a "router" layer. Probably the most common MoE design [FZS22, ABG+22], which we also focus on here, is to replicate the fully-connected module of a Transformer and route

tokens to the replica, referred to as an *expert*, with the highest assignment score predicted by a linear routing layer; see Figure 4.8 for an illustration. This design enables efficient training and inference of extremely large models, using 100s or even 1000s of experts, since each token is processed only by a small subset of the massive overall network.

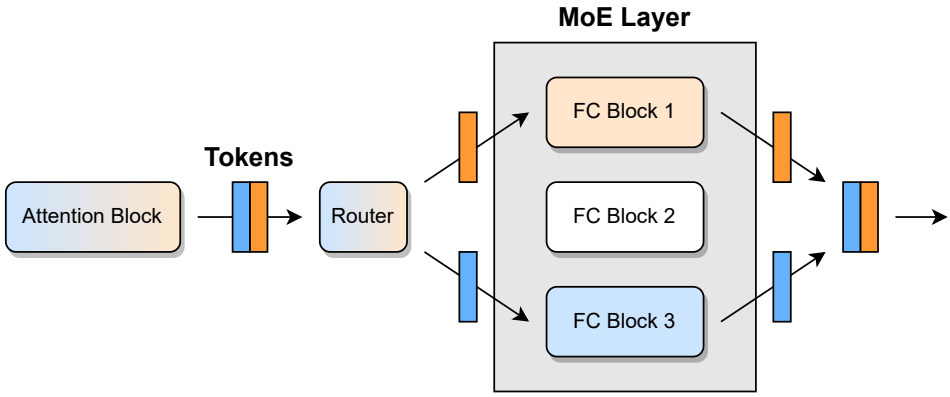


Figure 4.8: Example of an MoE Transformer block. Each token is routed to a different fully-connected (FC) block.

Data-Dependent Quantization

The currently most effective strategy for reducing model size and corresponding memory costs is *quantization*, i.e., converting model weights to lower numerical precision. While simple rounding can suffice for compression to 8 or even 4 bits, accurately quantizing models to extremely low precision (e.g., lower than 3 bits per parameter) typically requires more sophisticated *data-dependent* methods [NAVB+20, WCHC20, HNH+21].

Such data-dependent quantization methods use a small set of calibration data, which is passed through the model. As this happens, for each linear layer ℓ with weights W_ℓ, quantized weights Q_ℓ are determined one-by-one. Specifically, one approach to do this is by solving a layer-wise quantization problem, stated with respect to W_ℓ and the observed calibration data inputs X_ℓ at the current layer:

$$\text{argmin}_{Q_\ell} ||Q_\ell X_\ell - W_\ell X_\ell||. \tag{4.1}$$

Various solvers for Equation (4.1) have been proposed, with some optimized, in terms of speed and accuracy, particularly for extremely large models, like GPTQ (see Section 3.2) or ZeroQuant [YAZ+22, WYH23]. The former performs quantization using second-order information in the layer-wise Hessian matrix $X_\ell X_\ell^\top$, while the latter applies SGD-optimization with straight-through gradient estimation [BLC13].

Another noteworthy characteristic of many such methods is that per-layer quantization can be performed *sequentially*, using the input from the already partially quantized model up to layer $\ell - 1$, when quantizing layer ℓ, serving to reduce error accumulation. Concretely, this can be efficiently implemented by using X_ℓ to find Q_ℓ before passing on $X_{\ell+1} = Q_\ell X_\ell$ to the next layer.

MoE Quantization

There are several aspects which make very-low-bit, e.g. ternary (3 values) quantization promising for MoE models:

- In many architectures, almost all parameters are located in the experts, as they are 1000s of them. This means that, for size reduction, it suffices to focus on compressing just those experts and leave other layers in standard precision. This reduces error accumulation since only a subset of modules involved in a forward pass are actually quantized.

- Previous work has observed that extremely large dense models are more resistant to quantization noise than smaller ones [FAHA23, CCKDS23]. Large MoEs can be much larger than some of these massive dense models, and are thus a prime target for accurate quantization.

- MoE training involves additional stochasticity through routing instabilities and strategies like token dropping [LLX+20], which may inherently encourage high resistance to noise. Finetuning is also often performed with high dropout [FZS22].

Our experiments in Section 4.2.6 confirm that MoEs are indeed highly robust to extreme levels of quantization.

4.2.3 Scaling Up Data-dependent Quantization to MoEs

Challenges

While data-dependent quantization techniques have already been used to successfully compress large dense models up to 176 billion parameters [FAHA23, WYH23], applying them to *sparse mixture-of-expert models another order of magnitude larger* brings several new challenges.

Memory Costs. The first major problem we encounter is a large increase in the memory required to apply such techniques. Not only are the original model weights nearly $10\times$ larger, but the quantization process itself also needs $> 100\times$ more data. The latter constraint is because accurate data-dependent quantization methods require a sufficient number of input samples for each layer that is being compressed. For very large dense models, a few hundreds of thousands of "calibration tokens" typically suffice [FAHA23, YAZ+22]. However, in MoEs with thousands of layers, a single expert processes only a small subset of all inputs, hence we need much more tokens overall to achieve good coverage of all experts. Further, in encoder-decoder architecture models, like SwitchTransformers, each token is processed only by half of the model, again increasing data requirements. For *fast* compression, we must maintain intermediate results for the full calibration dataset, which requires 100s of GBs of memory for the largest models.

GPU Utilization. The next significant challenge is that existing large-scale quantization implementations, in particular for GPTQ and related methods [FAHA23, CCKDS23], are designed to be fast and memory efficient for the massive individual layers occurring in dense models. Meanwhile, MoEs typically have smaller layers, but $100\times$ to $1000\times$ more of them. Current implementations have poor GPU utilization in this case, and consequently bad performance. A similar issue occurs if activations and weights have to be transferred between CPU and GPU with high frequency, which may be required to cope with the massive memory requirements discussed previously.

Reliability Requirements. Finally, another issue when compressing models with tens of thousands of layers is that running into rare edge cases, which may break the process, is highly likely. This is includes numerical problems like non-invertible layer-wise Hessians, as well as model-specific ones, e.g., extreme routing patterns on particular layers.

System Design & Optimizations

In this section, we describe system-level design and optimizations to address the challenges in Section 4.2.3. This allows us to apply data-dependent compression to massive MoEs, while preserving the key feature of post-training compression techniques: the ability to perform effective compression using only modest computational resources, e.g., a single NVIDIA A6000 GPU and less than one day of compute. Although we focus on scaling the popular GPTQ method, most techniques described below will generalize to other approaches, like ZeroQuant [YAZ+22], as well.

Optimized Activation Offloading. As discussed before, a key challenge in compressing MoEs is that we need to maintain massive activation sets. Yet, it is possible to carefully orchestrate model execution in such a way that we only ever need to perform computation on a small subset of the intermediate data. This allows us to offload main storage from GPU, to much less expensive and plentiful CPU memory.

Concretely, we maintain a single large buffer B which we update as follows, for the dense part of a Transformer block:

1. Fetch one "sample" X, containing a few hundreds of tokens, from CPU to GPU.
2. Pass it through the corresponding dense layers to obtain the result Y.
3. Calculate and store expert assignment for tokens in Y.
4. Send Y back to CPU and overwrite X in B.

and respectively for the sparse part, looping over experts:

1. Fetch all individual tokens in B that have been assigned to expert E, denoted by X_E, from CPU to GPU.
2. Use them to produce compressed expert E' (for example, with GPTQ).
3. Run X_E through E' to get $Y_{E'}$.
4. Send $Y_{E'}$ back to CPU and overwrite X_E in B.

This process, which is visualized in Figure 4.9, minimizes both memory consumption and transfer cost: we need only a single copy of B and each token is only read and written twice per Transformer block.

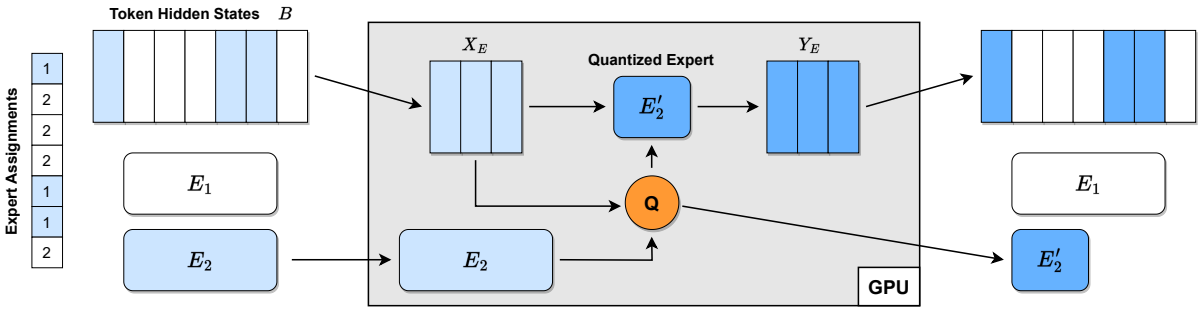


Figure 4.9: Illustration of the offloading execution for the sparse part of a Transformer block. An expert E_2 and its corresponding input tokens X_E are fetched to GPU memory to produce E_2', which together with the corresponding outputs Y_E are written back to CPU again.

List Buffer. To efficiently support per-sample access for evaluating dense model components, as well as fully-vectorized querying of expert tokens, we store B as a *list buffer* data structure. This can be seen as a huge contiguous buffer of all token hidden states, together with delimiter indices denoting boundaries between individual samples. Figure 4.10 illustrates this storage format. This datastructure is crucial for efficiency; naively iterating over samples and fetching relevant tokens via masking is unusably slow for large sample counts.

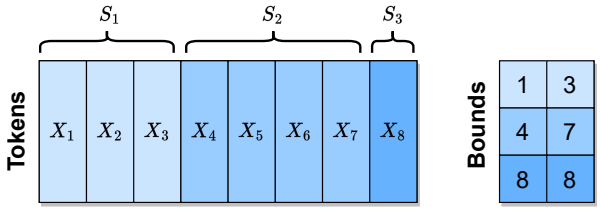


Figure 4.10: List buffer example with 3 samples, indicated by hue.

Lazy Weight Fetching. Since the weights of the 1.6 trillion parameter model consume 3.2 TB of storage, they cannot even be stored in CPU RAM. Thus, we lazily fetch them directly from disk storage as they are required. If we follow the inference procedure outlined previously, this would be exactly once. Afterwards, their memory is released again.

Expert Grouping. Additionally, in order to avoid GPU underutilization (see Section 4.2.3), we group multiple experts together and apply a joint *batched variant* of the GPTQ algorithm. Concretely, we extract the inputs X_E corresponding to all experts $E \in \mathcal{E}$ in group $\mathcal{E}$ (the X_E will generally have different sizes) and compute Hessians H_E. These matrices, together with the weight matrices W_E, are then stacked to 3-dimensional tensors, on which our modified GPTQ algorithm operates, compressing all experts simultaneously. We can also compute $H_E = X_E X_E^\top$ directly with a single matmul as the X_E are generally small enough, avoiding the slow per-sample accumulation employed by prior implementations. Our default expert groupsize $|\mathcal{E}|$ is 16, which we find to bring a good trade-off between GPU memory consumption and utilization.

Table 4.2 demonstrates the impact of expert grouping via GPTQ batching, when compressing a sparse encoder layer of switch-base-128 using 10k samples; $|\mathcal{E}| = 16$ yields about $\approx 6\times$ speedup over standard per-expert computation.

$\lvert\mathcal{E}\rvert = 1$	$\lvert\mathcal{E}\rvert = 4$	$\lvert\mathcal{E}\rvert = 16$
174.1s	54.4s	**28.8s**

Table 4.2: Sparse layer compression time for different $|\mathcal{E}|$.

Robustness Modifications. To achieve sufficiently high robustness for successfully quantizing trillion parameter models with tens of thousands of layers, we need to employ various numerical and memory adjustments. The most important are listed below:

- We use $10\times$ higher relative Hessian dampening $\delta = 0.1$, avoiding breakdowns with inf-values.
- Very few layer Hessians are not invertible even after high dampening; we skip GPTQ for those and simply perform vanilla rounding.
- Sometimes an expert receives a number of tokens that is much larger than average, leading to out-of-memory situations when these are fetched to GPU. We avoid this by capping the maximum number of tokens used for compression at $4\times$ the mean and use multiple iterations for computing and updating Y_E in such cases.

Accuracy Improvements

In addition to implementing a highly efficient compression system, we also make new discoveries about applying GPTQ in our particular context, i.e., for models trained for masked-language-modelling, MoEs and ternary quantization.

Premasking Special Tokens. First, we find that results can be improved if the various special separator tokens inserted by the masked-language-modelling task [RSR+20b] are excluded from the calibration data used for compression. Concretely, in the encoder, we mask out those "mask-tokens" during the Hessian computation. Meanwhile, in the decoder, we skip the token directly *before* such a special token as this is the one used to predict the latter.

As shown in Table 4.3 for switch-base-128 with 10k samples, this brings noticeably lower loss at no additional compute cost. We think that because those tokens are very common during training, the model is so robust in their prediction that any error compensation on them during quantization is unnecessary, while worsening correction for other tokens.

mask	BF16	2bit	tern
no	1.73	1.86	2.16
yes	1.73	**1.76**	**1.99**

Table 4.3: Impact of special token masking; validation loss.

4.2.4 Realizing Sub-1-Bit Compression

Using our system discussed in Section 4.2.3, we can accurately quantize extremely large SwitchTransformers to very low bit-widths: 2-bit and even ternary (3 possible values). Yet, in practice, this falls still short of our compression goal of less than 1 bit per parameter. We

find that compression rates can be pushed significantly further by taking advantage of the *low entropy in the quantized weights*. Next, we co-design an encoding scheme and a CUDA kernel which realize sub-1-bit per weight compression in practice, at minimal cost in terms of GPU execution overhead for inference.

Natural Sparsity

We pick quantization grids in standard fashion: row-wise around the min and max weights values [DLBZ22, FAHA23], e.g., for ternary: $\{w_{\text{min}}, 0, w_{\text{max}}\}$. These rather wide grids combined with the fact that weights are typically close to normally distributed, *naturally* lead to high sparsity after quantization, i.e., a large number of zeros. We demonstrate this in Table 4.4, averaged over all layers. For ternary weights, the largest model achieves close to *90% natural sparsity*; the standard deviation is also quite low, at $< 5\%$. Seen another way, the quantized weights have low entropy, meaning that, on average, significantly less bits per weight should be required for lossless storage.

model	2-bit	ternary
base128	72.2%	85.7%
large128	73.1%	86.4%
c2048	76.5%	88.6%

Table 4.4: Natural sparsity for different compressed models.

From Sparsity to Entropy

The direct way of utilizing these high zero proportions would be in form of a joint sparse & quantized representation [KCN+22, YCG23]: storing only the quantized values of non-zero weights, together with necessary position metadata. However, as our base quantization levels are already very low, standard sparsity metadata formats [EDGS20, LZW+23] would only allow limited additional compression. A bitmask indicating non-zero locations requires 1 bit per weight, while 10-13 bit (depending on layer size) column indices are even less memory efficient at the sparsity levels we encounter. Therefore, we take a different approach: we do not utilize sparsity directly but rather the *low entropy*, which is implied by the fact that a single value (0) occurs very frequently.

Fast GPU Decoding Challenges

In principle, we could group multiple consecutive ternary weights into super-symbols and then apply a code which assigns *variable length codewords* to those super-symbols, based on their probability of occurrence, for example, via a Huffman approach [Huf52]. If the quantized weight values were close to independent, this would achieve strong compression rates; in fact, for actual independence, they would be essentially Shannon-optimal [Mac03].

At the same time, our primary goal is to use compressed models for *fast and space-efficient inference*. Thus, it is critical not only that our encoding scheme achieves good compression, but also that it can be decoded fast on GPU hardware. This is challenging for a number of reasons:

Challenge 1: Entropy-based codes generally possess sequential decoding dependencies: symbol i can only be determined if the length, which is variable, of all $(i-1)$ prior symbols is known. Hence, processing consecutive symbols simultaneously leads to high synchronization overhead.

Challenge 2: Binary words in storage (e.g., INT32 blobs) may contain different numbers of decoded symbols. Consequently, even if rows/blocks are encoded independently, parallel decoding will happen non-uniformly, while all threads in a GPU-warp must always execute the same instruction. This would result in many wasted operations.

Challenge 3: Variable-length low-bit decoding involves a large number of binary operations like shifts, which are not particularly efficient on GPUs.

Challenge 4: Individual matrices of MoEs are typically not very large, making it difficult to split them into enough separately decoded segments to achieve good GPU utilization without having to store additional data to break sequential dependencies, which would harm compression rates.

In contrast, uncompressed half-precision matrix-vector products, which are the primary operation underlying generative inference, easily achieve close to ideal memory-bandwidth utilization and thus present a very strong baseline.

4.2.5 Compression Scheme & Kernel Co-design

To achieve our goal, we need to design a compression scheme and its GPU decoding kernel *jointly*, and potentially trade off compression for faster decoding. We begin with an overview of the main ideas behind our approach, followed by an in-depth discussion of key details.

Overview

Instead of a code with variable length codewords (see Section 4.2.4) mapping to fixed length data, we will use a *dictionary-based* code with fixed length codewords mapping to a variable number of symbols. Such LZW-based schemes [Wel84] are popular for general purpose compression like ZIP, as they are particularly effective for text data with long repeated segments. While a dictionary code is not ideal in terms of compression rate for the case of almost-random data in our application, it will be key for fast GPU decoding.

First, our kernel design uses one warp, that is 32 consecutive threads, to handle a row of a weight matrix, each of which is encoded independently. This addresses Challenge 4 in Section 4.2.4, yielding reasonable GPU utilization for relevant matrix sizes, with negligible metadata overhead. Further, we use a fixed-to-variable code with a large dictionary. This allows us to use a full warp to process one codeword at-a-time, extracting all data, while maintaining good efficiency, thus working around Challenges 1 and 2. This way, slow bit and base-3 operations (for ternary) can also be kept at a minimum, resolving Challenge 3.

Dictionary Design and Implementation

In general, assume that the values of a ternary weight matrix (denoted by 0, 1, 2) are distributed close to independently according to the distribution:

$$P(0) = p_0, \quad P(1) = P(2) = \frac{1-p_0}{2}, \tag{4.2}$$

where p_0 denotes the probability of sampling 0, e.g., 0.885 as per Table 4.4. As we plan to use a rather large dictionary, it should be shared between many weight matrices to not cause substantial storage overheads. We find that such a static dictionary works well enough, while simplifying memory efficient compression (see Section 4.2.3) as we do not have to collect statistics over many yet uncompressed experts.

Next, we consider pairs of ternary values $t = (t_1, t_2)$, whose corresponding probability is $P(t) = P(t_1)P(t_2)$. We generate the 2^{16} highest probability sequences containing at most 14 such pairs. This dictionary can be generated using a max-priority queue on probability, as shown by Algorithm 4.2.

Algorithm 4.2: Generate decoding dictionary sequences.

$Q \leftarrow$ max priority queue containing $(1.0, ())$
while $|D| < 2^{16}$ **do**
  $p, s \leftarrow \text{pop}(Q)$
  append s to dictionary if $0 < |s| < 28$
  for $t \in \{(t_1, t_2) \,|\, t_1, t_2 \in \{0, 1, 2\}\}$ **do**
    $\text{push}((p \cdot P(t), \text{cat}(s, t)), Q)$
  end for
end while

To briefly understand the procedure, notice that upon the first iteration, it will push all individual pairs $t = (t_1, t_2)$ to the priority queue, sorting them by decreasing probability, after which they will be expanded in this order.

We have exactly 2^{16} codewords as this allows us to store them in the native UINT16 datatype, avoiding any slow bit-extractions at this decoding level. Each of those codewords maps to two consecutive UINT32 values containing up to 7 pairs each, stored using 2 bits per ternary value, followed by the total number of pairs in the sequence; see also Figure 4.11. This format dictates our maximum chosen pair count of 14. Further, we consider pairs, rather than individual weights, to fit the maximum count into 4 bits. The 2-bit-per-weight format is used as there is enough space, while a more compact ternary encoding would involve slow modulo and division operations for extraction. We store the pair-count twice so that each thread can work with only half of the data, stored in a fast INT32 type.

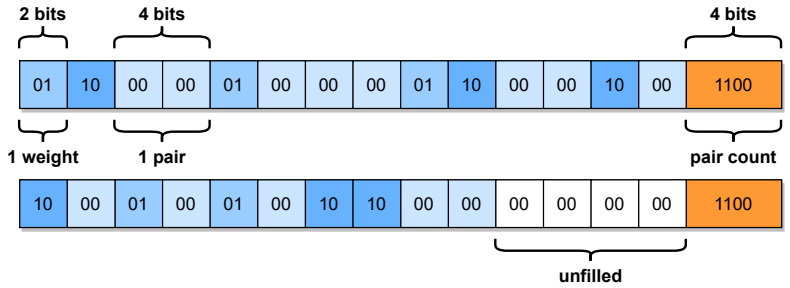


Figure 4.11: Data format of a dictionary entry; here of 24 weights.

Overall, mapping 16-bit codewords to 64-bit data blobs strikes a good balance between several goals: (a) Having codewords map to, on average, more uncompressed values than their bitwidth, a necessary condition for achieving < 1-bit compression. (b) Minimizing the overall

storage cost of the dictionary to fit into the L2-cache of the GPU, which is critical for good decoding performance. (c) Utilizing as many threads in a warp as possible for simultaneously extracting plain weights from the decoded data; usually, > 16 will do useful work and only 4 out of 32 threads are never active in this step. (d) Avoiding as many conditionals and extra operations necessary for dealing with non-uniform data storage as possible, which slow down parallelization.

Finally, we note that while dictionary lookups are in principle random access, keeping it sorted from highest to lowest probability ensures very favorable caching behavior. Since each lookup also automatically prefetches several subsequent elements, and most lookups are for frequently occurring codewords, there are many fast L1-cache hits.

Validation. To assess the effectiveness of our scheme, we compute achieved compression rates, both on a real ternary quantized c2048 model as well as on weight matrices sampled directly from distribution (4.2), yielding $20.07\times$ and $21.11\times$, respectively. This gap of only $\approx 5\%$ suggests that our simplifying independence assumption is indeed quite close for large models. We also note that our rates are only $\approx 20\%$ away from the distribution's (with $p = 0.885$) *theoretical* compression limit of $25.40\times$, which we consider a reasonable trade-off for enabling fast GPU decoding.

GPU Kernel

Having defined the dictionary format, we now discuss the design of the actual decoding kernel. We focus on the most important operation for inference, decompression fused with a matrix-vector-product. However, our techniques can easily be adapted to other use-cases, e.g., pure decompression.

Listing 4.1 provides CUDA-like pseudocode for our kernel, computing the matrix-vector-product of compressed matrix `w_comp` (with metadata `row_off` and `ter_minmax`, using dictionary `dec`) and BF16 vector `x`, into output buffer `y`. The handling of various edge cases and some index calculations have been removed for readability. Please see our source code for the fully functional implementation.

Parallelization. Overall, each threadblock will handle multiple consecutive rows, each of which is processed by a single warp. We use exactly one thread-block per GPU Streaming Multiprocessor (SM) with $\min(\#\text{rows\_in\_block}, 32)$ warps; if there are more than 32 rows in a block, (some) warps sequentially process multiple rows (note that this part is omitted in Listing 4.1 for simplicity). This avoids any bad wave quantization effects. We find this strategy to be an effective heuristic that yields good performance for all matrix shapes we consider.

Execution. Our kernel starts by loading the entire input vector to shared memory (`x_shared`, lines 7-9), using all warps in a threadblock. This enables fast element access in the subsequent per-row product-sum accumulations.

Next, each warp processes its corresponding row by first fetching (up to) 32 codewords into shared memory (`w_comp_block`, line 23) using a single coalesced transaction. It then loops over those symbols, processing one-at-a-time (lines 26-33). First, using 28 of its 32 threads (line 25), it fetches the corresponding decoding data from the dictionary where the first UINT32 is assigned to threads 0-13 and the second to threads 14-27 (`wx14`, line 27). Then,

each thread extracts its corresponding ternary weight (lines 29-30) and adds the corresponding input product into its own partial result accumulator (`res`, line 31). We note that the input reads from shared memory are contiguous and do not cause bank conflicts. Afterwards, each thread advances the offset index (`idx`, line 32) into the input vector by the total number of weights encoded in the current symbol.

Finally, after the full row has been scanned, a warp-reduction (lines 37-38) over the partial results of each thread yields the output (`y`, lines 39-40).

Ternary Decoding. Another relevant detail is that ternary weights are stored as $0, 1, 2$ (line 29) but need to be dequantized to $0, w_{\text{min}}, w_{\text{max}}$ for multiplication with inputs. We found that the most efficient way of performing this conversion is via a shared memory lookup table (lines 11-14). Crucially, this table needs to be replicated 32 times across the column-dimension to avoid very frequent bank conflicts, which would otherwise occur every time not all 28 threads dequantize the same value (line 30). Fortunately, there are only 3 input values and so its overall size is tolerable.

```
1   template <int num_warps, int w_width>
2   __global__ void Sub1MatVec(
3     int* dec,
4     ushort* w_comp, int* row_off, __nv_bfloat162* ter_minmax,
5     __nv_bfloat16* x, __nv_bfloat16* y
6   ) {
7     __shared__ float x_shared[w_width];
8     for (int i = thread; i < w_width; i += 32 * num_warps)
9       x_shared[i] = __bfloat162float(x[i]);
10
11    __shared__ float deq[3][32 * num_warps];
12    deq[0][thread] = 0;
13    deq[1][thread] = __bfloat162float(ter_minmax[row].x);
14    deq[2][thread] = __bfloat162float(ter_minmax[row].y);
15
16    __syncthreads();
17    __shared__ w_comp_block[32][num_warps];
18
19    float res = 0;
20    int idx = 0;
21
22    for (int i = 0; i < row_off[row + 1] - row_off[row]; i += 32) {
23      w_comp_block[warp][lane] = w_comp[i + lane];
24
25      if (lane < 28) {
26        for (int j = 0; j < 32; j++) {
27          int enc = w_comp_block[warp][j];
28          int wx14 = dec[2 * enc + (lane / 14)];
29          int ter = (wx14 >> (4 + 2 * (lane % 14))) & 0x3;
30          float w = deq[ter][thread];
31          res += w * x_shared[idx + lane];
32          idx += 2 * (wx14 & 0xf);
33        }
34      }
35    }
36
37    for (int i = 16; i > 0; i /= 2)
38      res += __shfl_down_sync(0xffffffff, res, i);
39    if (lane == 0)
40      y[row] += __float2bfloat16(res);
41  }
42
```

Listing 4.1: Simplified kernel pseudocode for a fused decompress + matrix-vector-product operation.

4.2.6 Experiments

General Setup

Models. We focus our experiments on the SwitchTransformer [FZS22] family of models. Our primary target is the very largest variant, c2048, with around 1.6 trillion parameters, but we also consider the comparatively small base128 (7B params) and large128 (26B params) versions for testing and ablations. We chose the SwitchTransformer family as it contains the largest publicly-available model, which also features a similar or higher number of training tokens to parameters ratio than potential alternatives like Artetxe et al. [ABG+22]. Further, those models are also among the most popular massive MoEs, with several implementations across frameworks [WDS+19, SCP+18, Goo23b].

Framework. As accessibility is a major goal of our work, we build our code-base around the PyTorch-backend of the highly popular HuggingFace [WDS+19] framework, which brings a number of additional challenges. First, we find that the largest model variants require a handful of bugfixes, primarily configuration and model setup changes, in order to run properly. We suspect that this is because their enormous sizes have rendered extensive testing very difficult. Second, we observed a major inefficiency in the context of generative inference for models with a large number of experts: the HuggingFace implementation will perform several (empty) CUDA calls for potentially 1000s of experts to which no token is routed, accumulating large overheads. We modify the implementation (also for baselines) to skip such unnecessary calls, leading to $> 10\times$ speedup for large models. We apply all changes to the HuggingFace framework only dynamically at runtime, so that our code can be run directly with an official installation.

Datasets. SwitchTransformers have been trained for a Masked-Language-Modelling (MLM) objective [RSR+20b] on the C4 dataset [RSR+20a]. Similar to most works in the area of LLM quantization [YAZ+22, FAHA23, DZ23], we focus on general *upstream* compression directly on this pretraining task/dataset combination. Consequently, our evaluation focuses on validation performance for C4/MLM, where we use the public reproduction of C4 on HuggingFace as well as their replication of the original masking procedure. Calibration data for compression is taken, in order, from the first two shards of the training set. For efficiency, we primarily evaluate on 128 samples (corresponding to the average loss over $>$ 10K tokens, which is quite stable) from the first shard of the validation set, but we also perform some evaluations other datasets.

Hardware. All compression experiments, including those for the very largest models, can be performed in less than a day on a single NVIDIA A6000 with 48GB of GPU memory. However, efficiently compressing trillion parameter models using a large number of calibration samples requires a few 100GBs of (CPU) RAM; the original 1.6T model itself also occupies > 3 TB disk storage.

Compression Results

Accuracy. We begin by quantizing all SwitchTransformer models to 2-bit and ternary precision, and evaluating their validation loss. Our default number of calibration samples is 10K for 128 experts and 160K for 2048, but we also consider using $0.5\times$ and $2\times$ as many samples. In addition to using our efficient QMoE framework discussed in Section 4.2.3, we also

consider a standard round-to-nearest (RTN) baseline [DLBZ22]. We simulate the latter by fixing Hessians to the identity matrix, thus applying precisely the same quantization settings and evaluation protocol. Table 4.5 summarizes our results.

Perhaps surprisingly, vanilla rounding (RTN) does not lead to a complete model collapse even at ternary precision, emphasizing the high robustness of large MoEs to quantization. Nevertheless, the loss increases are quite significant for smaller models at 2-bit and far too large to be useful at ternary precision. In contrast, using data-dependent quantization, 2-bit is achievable at minimal loss (1.7% relative on c2048) and ternary at only a small increase (6.7% relative on c2048). This demonstrates not only the effectiveness of such advanced quantization methods in this context, but also shows that extremely low-bit compression is indeed practical for massive MoEs.

method	base128		large128		c2048	
	2bit	tern	2bit	tern	2bit	tern
BF16	1.73		1.55		1.18	
RTN	2.27	4.54	1.96	2.79	1.33	2.15
QMoE 0.5x	1.78	2.11	1.54	1.70	1.22	1.27
QMoE 1.0x	**1.76**	1.99	**1.56**	1.69	**1.20**	**1.26**
QMoE 2.0x	**1.76**	**1.93**	1.57	**1.64**	1.21	**1.26**

Table 4.5: Comparing C4 validation losses for 2-bit and ternary (tern) quantized SwitchTransformers. "QMoE 0.5x" indicates that only half of the default number of calibration samples are used.

Additionally, we conduct evaluations on Arxiv, GitHub, StackeEchange and Wikipedia data sampled from RedPajama [Com23]. Even though only $< 0.01\%$ of our C4 calibration data originates from those websites, the compressed model still preserves performance almost as well as on the core of the distribution (see Table 4.6).

bits	arxiv	github	stackexch.	wiki
BF16	1.31	0.99	1.15	1.20
2-bit	1.34	1.05	1.17	1.24
tern	1.42	1.13	1.22	1.32

Table 4.6: Additional evaluations for the c2048 model.

In terms of calibration data, we see that increasing the amount of samples generally improves performance slightly, most noticeably for ternary quantization, but there is also some noise in the process, especially at 2-bit.

Compression. Next, we investigate the actual compression rates that are achieved by further compressing ternary models using our scheme introduced in Section 4.2.4. We consider both compression relative to just the MoE modules (the model parts we quantize) as well as to the full model and all its metadata. The compression rates and overall checkpoint sizes are listed in Table 4.7.

In general, measuring only relative to parts we compress (moe-only), all sizes achieve $> 16\times$ compression rate and thus < 1 bits per parameter storage. On c2048, even the overall rate,

model	moe-only	full	size [GB] bf16	ours
base128	$17.06\times$	$11.76\times$	14.9	1.27
large128	$18.34\times$	$13.32\times$	52.7	3.96
c2048	$20.07\times$	$19.81\times$	3142	158.6

Table 4.7: Compression rates and sizes for ternary models.

including all uncompressed dense layers, remains at $19.81\times$, corresponding to *0.807 bits per parameter*, reducing the checkpoint size from 3142GB to 158.6GB. One can also observe that compression rates increase with model size, which is for two reasons: (a) natural sparsity increases while our encoding dictionary is also optimized for c2048 (see Section 4.2.4), and (b) weight distributions become closer to independent for larger layer sizes.

Runtime. Finally, we evaluate how long it takes to produce compressed models on a single A6000 GPU, for different amounts of calibration data. The results are shown in Table 4.8. Smaller models can be compressed in less than an hour and even c2048 in less than a day, confirming the high efficiency of QMoE. The runtime increase from large128 to c2048 is roughly proportional to the difference in size, despite the latter using $16\times$ more samples. This is because the number of samples per expert stays constant and the expert size increases only slightly. Finally, we note that simply (iteratively) loading the original 1.6T model into RAM takes close to 5 hours on our slow disk storage.

model	5K/80K	10K/160K	20K/320K
base128	8.4min	14.0min	21.6min
large128	22.0min	30.2min	45.2min
c2048	13.3h	16.0h	20.8h

Table 4.8: Compression runtime for different calibration data size.

Runtime Results

Individual Layers. Our kernel performance evaluation starts with a direct (isolated) comparison of our compressed matrix-vector product kernels (see Section 4.2.4) against PyTorch's standard (uncompressed) bfloat16 cuBLAS kernels. Figure 4.12 (Left) shows the time taken by our compressed kernels relative to bfloat16, for the matrix shapes found in our MoEs, on two different GPUs. While our kernels have to perform a lot less slow (global) memory reads than the bfloat16 baseline due to lower storage costs, they need to spend much more compute for complex unpacking of the heavily-compressed weights. Nevertheless, executing our compressed kernels takes less time than the close to ideal bfloat16 baseline in all cases, with up to 35% speedup on specific matrix shapes. We note that these are very low-latency operations, with the smallest matrix taking < 0.02 milliseconds and the largest < 0.05.

End-to-End Execution. Finally, we also benchmark our kernels end-to-end in HuggingFace on the real weights of our compressed MoE models. We consider an individual user application, like [FAHA23, LKM23, PPK+22], where a single prompt (sampled from C4) should be processed to generate a 128-token response. As actually running the bfloat16 version of the c2048 model

would require > 65 A6000 and > 130 3090 GPUs (versus 4 and 8, respectively, for sub-1-bit compressed weights) we have to estimate its runtime. We do this by having all experts in a layer point to the same weight data (resolving memory issues), which allows us to collect timings with precisely the same overheads as for our compressed models. However, this is a highly optimistic estimate since real execution would require close to $20\times$ more GPUs, with corresponding communication overheads, and our numbers should thus be viewed as a lower bound.

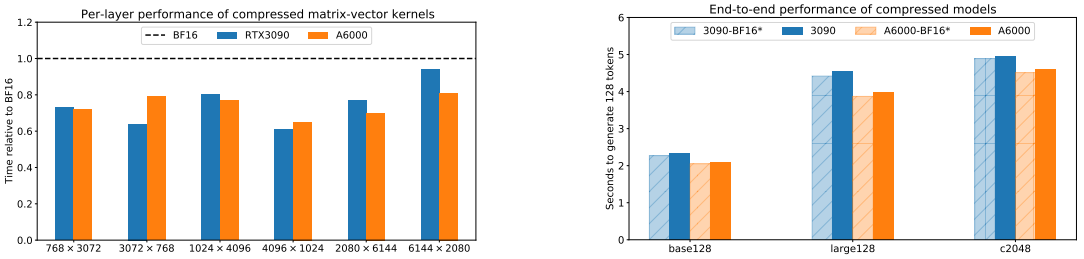


Figure 4.12: (Left) Per-layer compressed kernel performance relative to uncompressed execution. (Right) End-to-end runtimes of compressed models and estimates (*, would require 65/130 GPUs) for bloat16 baselines. c2048 is run on 4×A6000 and 8×3090 GPUs, respectively.

The results, shown in Figure 4.12 (Right), demonstrate that end-to-end execution of compressed models is only $< 5\%$ slower than standard (uncompressed) execution. This slight slow-down despite faster per-layer timings is due to the fact that the encoder may sometimes route multiple tokens to the same expert. Our current implementation naively executes a separate matrix-vector product for each token, while the baseline performs a much more efficient joint matrix multiplication. For applications where this is a significant bottleneck, one could easily introduce an inner loop over tokens into our kernel (Listing 4.1, line 30), or fully decompress first, followed by a standard matmul, for large token counts.

4.2.7 Related Work

Mixture-of-Expert (MoE) Models. Mixture-of-expert models are a popular approach for creating large-scale models that are more efficient for inference [FZS22, ABG+22, CDLCG+22]. At the core of MoEs lie (sparse) routing mechanisms, of which many variants have been proposed. Those range from static assignment based on input token IDs [RSW+21], over dynamic token-to-expert matching [ZLL+22], to "soft" routing of linear input combinations [PRMH24]. Since MoEs can feature rather different computational profiles from standard dense models, there is also significant research on optimizing inference and training systems [BCD+22, GNYZ23, HCX+23]. Among the most critical problems in this area are data-exchanges between accelerators during routing and dealing with uneven compute-loads for different experts.

LLM Quantization. Quantization is a very popular compression technique, which has seen a vast amount of work [GKD+21], especially in the context of LLMs. Specifically, the ability to perform accurate weight quantization for billion-parameter models has greatly boosted their accessibility: it has been shown that extremely large dense models can be quantized to 8- or even 4-bit precision at little accuracy loss [DLBZ22, YAZ+22, FAHA23, DZ23]. Pushing towards even lower bitwidths via more sophisticated compression formats, like multi-level grouping coupled with higher-precision outliers [DSE+24, AMF+23], or new quantization techniques, like incoherence preprocessing [CCKDS23], is an active area of research. Currently, accurate

quantization to 2 or less bits appears to be a major barrier for post-training quantization of standard LLMs. By contrast, in this work we show that massive MoE models appear to be significantly more compressible, as we achieve sub-1-bit compression at comparable loss increases to 3-bit or 4-bit quantization of standard LLMs.

MoE Compression. There has also been work on compressing MoE models in particular. Chen et al. [CHX+22] and Koishekenov et al. [KNB23] perform compression via specialization of MoEs to specific "downstream" finetuning datasets by pruning components not relevant to the particular task. In contrast, we focus on general "upstream" compression of the pretrained model, via extremely low-bit quantization. Other works [KHFA22, YGW+23, KHFA23] also perform MoE quantization, but focus on noticeably higher bit-widths, like 8 or 4 bits per weight. This is accomplished primarily via simple rounding, which, as shown by our experiments, is not accurate enough for full 2-bit or lower compression. Kim et al. [KFA22] achieve 2-bit quantization on a 5 billion parameter MoE, which is considered relatively small in this area, by further optimization of the model via Quantization-Aware Training [NFA+21]. Applying such an approach for trillion-scale models would be extremely resource intensive. They also do not provide any mechansims for exploiting low-bit quantization and its corresponding natural sparsity in practice, which is challenging and constitutes a key contribution of our work.

Relative to prior work, we are particularly focused on scalabilty and practicalty. While existing works study models with at most tens of billions of parameters, we demonstrate all our techniques at trillion parameter scale.

4.2.8 Discussion and Limitations

We have presented QMoE, an end-to-end compression and inference framework for massive MoEs. We showed, for the first time, that models like the trillion-parameter SwitchTransformer-c2048 can be accurately compressed to less than 1 bit per parameter, close to $20\times$ compression rate, in a custom format that enables the first efficient execution of such a model on a single commodity GPU server. QMoE is open-source and built around the popular HuggingFace framework, making deployment and research for massive MoEs significantly cheaper and more accessible.

Our study is limited in terms of models, as only very few massive and accurate MoEs are available publicly. Additionaly, due to their size, most MoEs are trained and deployed in different bespoke framework, requiring complex manual integrations to use for further research. A natural extension of our work would be to apply our QMoE techniques to other MoE models or variants, such as Artetxe et al. artetxe2021efficient or SoftMoEs [PRMH24]. It would also be interesting to further finetune a compressed model for specialized down-stream tasks. Zoph et al. [ZBK+22] report strong results when finetuning only non-expert layers, which QMoE leaves uncompressed, suggesting that this could be a promising direction for future work.

CHAPTER 5

Scaling Laws

In Chapter 3, we studied compression primarily from the perspective of a broad audience, that is researchers or practioners who would like to take an existing model and make it more efficient, in order to run it on their available hardware, which may not be particularly powerful. Hence, the compression process itself being cheap enough was absolutely critical. In this chapter, we consider a very different use-case: a large company able to train its own massive models completely from scratch, which would like to reduce the cost of serving millions of users by obtaining more compressed networks. In this scenario, the cost of compression is secondary (as huge compute clusters are easily available), what really matters are the efficiency gains achieved.

Concretely, we will focus on much more expensive training-based (as opposed to post-training) compression techniques. Here, especially in the context of modern models which scale exceptionally well in both size and amounts of training data, studying individual models is not always very meaningful. Instead, the most critical questions evolve around the *joint scaling behavior* of compression, model size and training data. Section 5.1, which is based on the ICLR 2024 paper "Scaling Laws for Sparsely-Connected Foundation Models" [FRH+24] takes the first key steps in that direction, identifying a corresponding *scaling law* and defining the concept of *optimal sparsity*.

5.1 Scaling Laws for Sparsely-Connected Foundation Models

5.1.1 Motivation & Overview

Foundation models [BHA+21], loosely defined as large (often Transformer-based [VSP+17]) networks that are trained on massive quantities of highly general data, have driven significant progress in deep learning, for both natural language [BMR+20] and vision tasks [DBK+21]. One key property of such models is the predictability of their performance when scaling various model attributes, such as the number of parameters and the amount of data or computation used [KMH+20]. This is encapsulated by *scaling laws*, which make it possible to accurately predict the final performance of a model given its size and training budget (i.e., amount of training data).

A parallel trend, motivated by computational costs, has been the focus towards increased efficiency for large models. This is usually achieved by employing compressed parameterizations via quantization [GKD+21] or sparsification [HABN+21], during inference and/or training, which can lead to reduced run-time via both software and hardware support [EDGS20, YAZ+22]. Despite major community interest in efficiency, the impact of these compressed representations, in particular of parameter/weight sparsity, on the scaling behavior of foundation models is not well understood; especially, when applying powerful but expensive training-based compression methods [JKC+18, ZG17].

In this paper, we address this gap by studying the relationship between sparsity and scaling laws for foundation models. We focus on *weight sparsity*, that is, on networks whose individual connections are pruned, and on Transformer [VSP+17] models for both vision [DBK+21] and language [RSR+20b] domains. We use the massive JFT-4B [Goo23a] and C4 [RSR+20a] datasets, which are several orders of magnitude larger than what has been employed so far by the vast majority of work on sparsity. In this massive dataset regime, dense models continue to improve with prolonged training and it is unclear if sparse models can be competitive in a fair comparison, using equal amounts of training compute and data. This is in contrast to popular pruning benchmarks, e.g., ImageNet [DDS+09], executing for many training epochs, where dense models tend to saturate [KKI+23], allowing sparse models to achieve major gains relative to dense models with a comparable number of parameters.

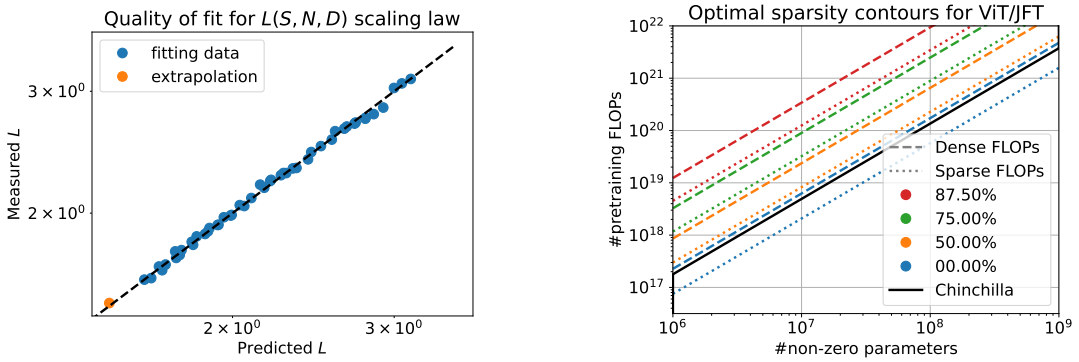


Figure 5.1: (Left) Fit and extrapolation quality of the $L(S, N, D)$ scaling law on T5/C4. (Right) Optimal sparsity S_{opt} contours fitted on ViT/JFT, for sparse and dense costs (details in Section 5.1.4).

In order to quantify the benefits of sparsity, or the lack thereof, in this large-dataset regime we develop joint scaling laws that relate the sparsity of a network, its effective size and the amount of data used for training. We show that, for sparsity S, number of non-zero parameters N and amount of training data/steps D, the validation loss L approximately satisfies the following law, for both vision and language tasks:

$$L(S, N, D) = \left(a_S(1-S)^{b_S} + c_S\right) \cdot \left(\frac{1}{N}\right)^{b_N} + \left(\frac{a_D}{D}\right)^{b_D} + c, \tag{5.1}$$

Intuitively, the first two summands capture the power law scaling in terms of capacity, i.e. sparsity and non-zero parameters, and data, respectively, while c is a lower bound on the achievable task loss. In more detail, the first multiplicative term captures the impact of sparsity, here expressed as remaining density $(1-S)$, which itself follows a saturating power-law with coefficient a_S, exponent b_S and limit constant c_S. The exponents b_N and b_D scale the (non-zero) parameter count N, and the data D term, respectively, as is common in classical scaling laws [KMH+20].

We validate this formula empirically using large vision and language datasets, several model sizes, amounts of training data and sparsity levels. Please see Figure 5.1 (Left) for an illustration of the scaling law fit and extrapolation quality. In turn, this law allows us to obtain several new insights for sparsely connected foundation models:

- First, the sparse scaling law suggests that sparsity affects each model size in a similar way, i.e., as a multiplicative constant to the size scaling. At the same time, sparsification does not appear to interact significantly with the data scaling; the original dense term in D is preserved.
- Second, we can use our scaling law in Equation (5.1) to analytically derive the *optimal sparsity* S_{opt} for a given inference size and training budget, allowing us to predict the regime where sparsity could actually provide benefits over simple dense model rescaling and extended training.
- Our analysis of optimal sparsity S_{opt}, demonstrated in Figure 5.1 (Right), shows that its iso-contours run parallel to the dense compute optimal Chinchilla line [HBM+22] of the respective model and task. Importantly, the optimal sparsity increases with longer training. Further, while optimal dense models define a line on the parameter-FLOPs surface, optimal sparse models form a half-plane (with different sparsities unlocking multiple optimal sizes for a fixed training cost).
- In addition, we find that the main conclusions of our law hold also for hardware-friendly n:m sparsity patterns [MLP+21], that relative capacity gains through sparsity are consistent across domains, and that pruning well-trained dense models is more efficient than training from scratch (while sparsifying), if dense checkpoints already exist, but is significantly slower otherwise.

In sum, our results provide the first scaling law for characterizing the impact of sparsity on the performance of Transformers trained on massive datasets. From the conceptual perspective, this provides a simple tool to understand the power–but also the limitations–of sparsity for a given task/model combination. From the practical side, this can be used to determine whether sparsity can be a reasonable option for inference or training speedups, in settings where specific software/hardware support for such compressed representations is available.

5.1.2 Scaling Laws for Parameter-Sparse Transformers

Fair evaluation in the presence of strong scaling. In the context of modern Transformers trained on massive datasets, popular evaluation approaches for pruning [GEH19, SA20, SWR20, SJP+21, BCM+23] that have been reasonable for standard benchmarks like ResNet50/ImageNet [SA20, SJP+21] or BERT/GLUE [SWR20, KCN+22], require careful reconsideration to ensure meaningful comparisons. Specifically, it is critical to adopt the resource-equivalent setting of Liu et al. [LSZ+18] and Jin et al. [JCR+22], which we illustrate below.

For example, assume that a model pretrained for 100k steps is pruned to 50% sparsity over another 100k steps (a standard setup for ResNet50/ImageNet). The resulting network should not be compared to the original one, as it has had $2\times$ more training overall. Further, even a comparison against a $2\times$ smaller model (same non-zero parameter count) trained for 200k steps (same amount of training) is not necessarily fair, as training this smaller dense model

requires less overall compute than producing the larger sparse one (as we perform sparsification only gradually). In both cases, due to the strong scaling properties of Transformers trained on massive quantities of data [KMH+20, HBM+22], the respective baseline would have most likely improved significantly with more data/compute as well. Thus, the proper comparison point for a sparse network is a *dense model with the same number of parameters trained for equivalent compute*. This resource normalization, required by strong scaling and no overfitting, renders this setting very challenging.

Experimental setup overview. We execute extensive training sweeps across sparsity, size and data, which we then subsequently use to develop scaling laws. Now follows a very brief summary of our main setup; a detailed discussion of all our choices, including the experiment grid and hyper-parameters, can be found in Appendix B.1.1. In terms of models and datasets, we focus on Vision Transformers [DBK+21] trained for multi-label image classification on the JFT-4B dataset [DDM+23], consisting of 4 billion images, as well as encoder-decoder T5 models [RSR+20b] (improved 1.1 version [Goo23b]) trained for masked-language-modelling on C4 [RSR+20b], consisting of 150+ billion tokens. We follow the model's respective original training recipes [ZKHB22, RSR+20b] and carry out sparsification *during* training via gradual magnitude pruning [ZG17], using a cubic schedule starting at 25% of training and ending at 75%. Our setup is optimized for robustness and consistency across scales rather than maximizing pruning performance on one particular setting (see also Appendix B.1.3).

5.1.3 Deriving the Core Law

Dense scaling. It is well established [KMH+20, HBM+22] that the pretraining validation loss of *dense* Transformers can be approximately modeled, in terms of parameter count N and amount of training data D, by functions of the following form:

$$L(N, D) = \left(\frac{a_N}{N}\right)^{b_N} + \left(\frac{a_D}{D}\right)^{b_D} + c. \tag{5.2}$$

The first two summands capture the power law scaling in terms of size and data, respectively. Meanwhile, c represents the inherent stochasticity of the modelling problem as a lower bound on the loss. The scaling exponents b_N and b_D are usually quite stable for a particular task, whereas the constant coefficients a_N and a_D vary with minor process changes like a different architecture or optimizer.

Scaling laws usually assume an ideal training setup with no data repetition and focus on modelling the non-bottlenecked regime (e.g., with sufficient steps/data/batchsize/etc.) rather than on edge cases [KMH+20, HBM+22]; we follow suit. Further, we deliberately consider the pretraining loss and infinite data setting to assess the effectiveness of sparsity in its most challenging (one essentially needs to fit the data as well as possible) yet also most useful application (all further post-processing would directly benefit from a compressed base model; see also Appendix B.1.6).

Preliminary observations. The key question we hope to address is how parameter sparsity S enters this core scaling relationship; understanding this will enable studying other interesting aspects like optimal sparsity or limit performance. A priori, it is not obvious how S should enter into Equation (5.2) to form $L(S, N, D)$, where N denotes the number of *non-zero parameters*. Are larger models easier to sparsify, does longer training help highly sparse models more, or is sparsity mostly independent of other parameters? Therefore, to identify the functional form of

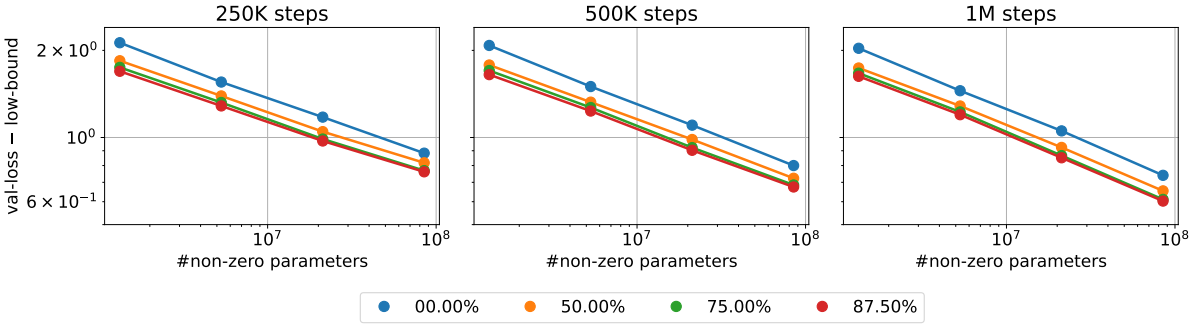


Figure 5.2: Visualization of T5/C4 sweep results for all sizes and sparsities, grouped by training steps.

the scaling law, we run a T5 training sweep over sparsity, size and steps. Figure 5.2 shows validation loss (with a lower bound $c = 1$ subtracted to account for power law saturation against the inherent uncertainty limit) versus model size for all sparsity levels, grouped by the number of training steps. Please observe that the scaling of this plot, as well as most other visualizations in this paper, is log-log.

We make three major observations from these graphs:

1. The loss vs. #non-zero curves for all sparsity levels seem to form almost parallel lines, differing primarily in the intercept.
2. The higher the sparsity the lower the loss, but gains are quickly diminishing.
3. The overall shape of all curves is very similar for each training duration, the y-values just tend to shift a bit downwards with more training steps.

Sparse scaling law. We now use the previous insights to construct our $L(S, N, D)$ formula. Observation 1 suggests that the model size power law scaling for all sparsity levels differs primarily by a constant factor (intercept in a log-log plot); b_N (the slope) stays fairly consistent. Based on observation 2, we model this sparsity factor as a (quickly) saturating power law. Finally, observation 3 indicates that sparsity and data scaling are mostly independent, hence we simply keep the original D-term. In summary, these observations lead us to the following joint scaling law:

$$L(S, N, D) = \left(a_S(1 - S)^{b_S} + c_S\right) \cdot \left(\frac{1}{N}\right)^{b_N} + \left(\frac{a_D}{D}\right)^{b_D} + c. \tag{5.3}$$

To properly model that 0.75 is twice as sparse as 0.5, we define the sparsity power-law part via the corresponding compression rate $1/(1 - S)$. Further, a_N is subsumed by a_S and c_S, leaving 7 free parameters. On a high level, our scaling law combines a *capacity limit* term, comprised of size and sparsity (which can encode extra information via its zero pattern), with the standard data limit term. Lastly, $S = 0$ recovers the established $L(N, D)$ form.

T5/C4 results. Next, we fit the coefficients of $L(S, N, D)$ to our entire T5 sweep data. This is accomplished, following [HBM+22], by minimizing the Huber-loss of $\log L$ with $\delta = 0.001$ (for robustness against outliers) using BFGS, for multiple random starting points. We plot actual vs. predictions in Figure 5.1 (Right) to judge the quality of our final fit (see Appendix B.1.4 for coefficient values). All in all, the predictions match the observed data quite closely (despite

having ≈ 7 datapoints per free parameter), demonstrating the compatibility of the law in (5.3) with the observations.

Furthermore, we evaluate extrapolation performance by pruning a 2.3 billion parameter model to 75% sparsity. This constitutes an $\approx 6.75\times$ *larger* target number of non-zero parameters than the maximum in our fitting data, which is a similar level of extrapolation as was done in the Chinchilla study [HBM+22]. To avoid any architecture bottlenecks and achieve better training utilization, we use the T5-XL architecture (rather than a rescaled T5-base) and train with batchsize 256 for 250k steps (rather than 500k with batchsize 128). Despite these changes to our setup, the prediction of our fitted scaling law is quite close to the actual validation loss; see Figure 5.1 (Right).

ViT/JFT-4B results. Lastly, we execute a ViT training sweep and also fit a scaling law of the same (5.3) form as for the T5 data. Here we use $\delta = 0.01$ and do not take the log of L as we find the NLP-optimized settings from before to exclude outliers too aggressively for ViT data (which gives a poor fit for smaller models). We note that this sweep contains $> 2\times$ more datapoints, leading to more robust coefficient estimates. We qualitatively compare predicted and actual loss-vs-data curves in Figure 5.3, organized by sparsity level. We strongly emphasize that the predictions in all subplots here are produced by *a single joint law* with the same parameters (*not* one fit per image). As can be seen, for the most part, our law appears to match the collected datapoints very well. Only at the lowest amount of training, some points are a bit off the prediction curve; we suspect that this may be related to the fact that these runs only involve comparatively few training steps, which may be a slight bottleneck for the optimization process. Finally, we train a 75% ViT-L for 3.6 billion images to validate extrapolation, ending up only 0.09 better than the corresponding prediction, in line with being a slightly better base architecture than the family used for our sweep (see Appendix B.1.1).

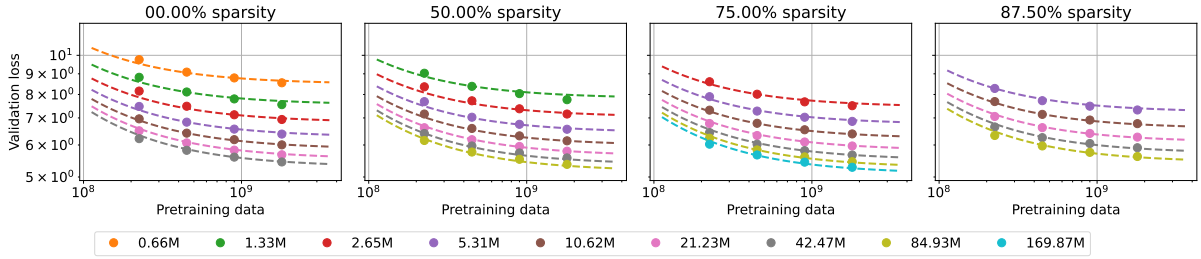


Figure 5.3: Visual comparison of the ViT scaling sweep data and the corresponding fitted scaling law.

5.1.4 Optimal Sparsity

One particularly interesting feature of the joint scaling law just derived is that it allows easily comparing models with different sparsities but the same number of non-zero parameters and training cost. Thus, we can determine in which situations sparse models are better than dense ones, according to all criteria discussed in Section 5.1.2. Specifically, we can define the following quantity:

Optimal sparsity. *The sparsity value $S_{opt}(N, C)$ which yields the lowest validation loss for a fixed number of non-zero parameters N and training cost C.*[1]

[1]We note that it is common in the literature [HBM+22] to define scaling laws in terms of parameters N and data D, but switch to expressing scaling in terms of computational cost C whenever relevant.

There are two ways of defining training costs in this context: (a) *densely*, as the cost of training a dense base model of size $N/(1-S)$ for the same amount of training steps, or (b) *sparsely*, as the actual FLOPs spent to produce the sparse model, assuming that sparsity can be perfectly exploited during training as soon as it appears. For our particular sparsification schedule, (b) can be calculated by multiplying the training costs of a dense model, approximated as $6ND$ [KMH+20] (or half for encoder-decoder architecture models), by (see Appendix B.1.5 for this and other derivations):

$$c_{\text{mul}}(S) = (0.25 + 0.50 \cdot (1 - 0.75 \cdot S))/(1 - S) + 0.25. \tag{5.4}$$

As we have assumed that the amount of training equals the amount of new data, we can determine the performance of a sparsity S model trained for compute $C = 6ND \cdot c_{\text{mul}}(S)$ by querying L with $D_S = (C/6N)/c_{\text{mul}}(S)$, i.e., scaling down the D corresponding to C by the increase in training costs of the sparse model. Inserting D_S and then differentiating with respect to S gives the contour line for which sparsity S is optimal, i.e., achieves the lowest loss among all possible sparsity choices, when training for the same compute:

$$b_D \cdot \frac{c'_{\text{mul}}(S)}{c_{\text{mul}}(S)} \cdot \left(c_{\text{mul}}(S) \cdot \frac{a_D}{D_S}\right)^{b_D} = a_s b_s (1 - S)^{b_S - 1} \cdot N^{-b_N}. \tag{5.5}$$

An interesting property about this contour is that it implies $D_S = O(N^{b_N/b_D})$, meaning that if data- is stronger than size-scaling, then the same sparsity is optimal for a smaller data-to-size ratio on larger models. This is sensible as a process bottlenecked more by capacity than by data will benefit more from increasing the former, e.g., by adding sparsity. Finally, we want to point out that S_{opt} can often also be determined explicitly by solving (5.4) for S, e.g., here for dense training costs with $c_{\text{mul}}(S) = 1/(1 - S)$:

$$S_{\text{opt}}(N, C) = \max\left\{1 - \exp\left[\left(\log\frac{b_D}{a_S b_S} + b_N \log N\right)/(b_D + b_S)\right] \cdot \left(\frac{6a_D N}{C}\right)^{b_D/(b_D+b_S)}, 0\right\}. \tag{5.6}$$

Empirical results. We now compute optimal sparsity curves for our experimental T5 and ViT data, for which we fit scaling laws in the previous subsection. Figure 5.1 (Right) and 5.4 show the optimal sparsity contours, both for dense and sparse costs. An interesting feature of Equation (5.5) is that all sparsity contours are, by construction, parallel to the Chinchilla compute optimal line [HBM+22], which denotes ideal utilization of training FLOPs for fully dense models; this can be clearly observed in the plots as well. However, we note that the Chinchilla line does not necessarily correspond to the $S = 0$ case since non-zero sparsity may be optimal in this regime (this is the case for sparse-FLOPs).

The key take-away from these results is that as one trains significantly longer than Chinchilla (dense compute optimal), more and more sparse models start to become optimal in terms of loss for the same number of non-zero parameters. This is because the gains of further training dense models start to slow down significantly at some point, allowing sparse models to overtake them. We further illustrate this effect on a subset of our actual ViT data in Figure 5.5.

The practical question now is how much longer training is necessary? In terms of sparse FLOPs, 50% sparsity is already optimal for $< 2\times$ (ViT) and $< 3\times$ (T5) longer training than Chinchilla; for dense FLOPs it is $\approx 5\times$ and $\approx 70\times$, respectively. While the latter number seems quite high at first glance, we note that language models of the sizes we consider here are already typically trained for $> 100\times$ longer than Chinchilla [BMR+20]. Additionally, larger

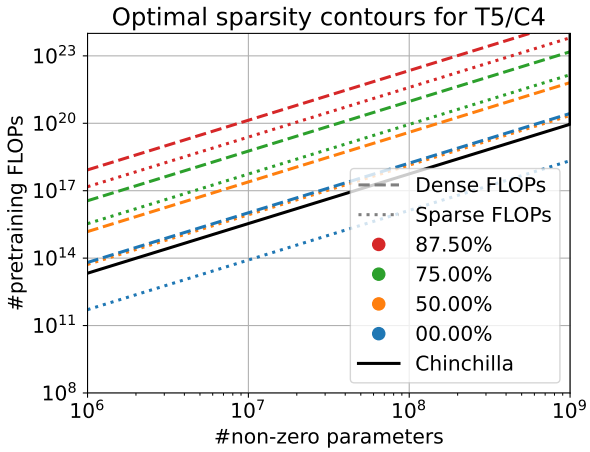


Figure 5.4: Optimal T5 sparsity contours.

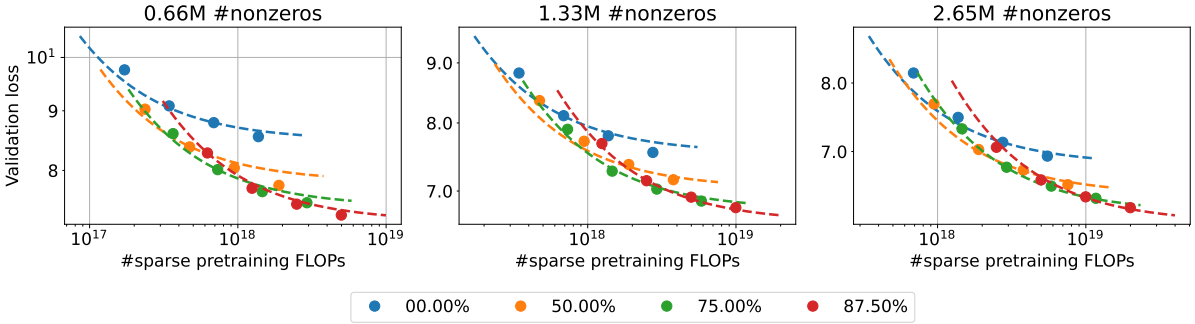


Figure 5.5: Loss vs. sparse pretraining FLOPs for ViT models of varying sparsity.

models are being trained with more and more data as well, e.g., Llama2-7B with $\approx 14\times$ Chinchilla [TMS+23]. In general, the optimal sparsity at a given point (N, C) is lower for dense than sparse FLOPs since the former assumes that sparsity provides no benefits *during* training.

Limit Performance

In the previous section, we have focused only on *when* sparse models become optimal but not *how much better* they can be compared to dense models. In this section, we study the following question: How much larger, and thus computationally more expensive, does a dense model need to be in order to match the loss of a smaller sparse model with very long training? Since we have found the scaling term in D to not interact with sparsity in Section 5.1.3, it suffices to compute the increase in N required to lower the loss by the same factor as the increase in S via:

$$\text{gain}(S) = \left(\frac{a_S(1-S)^{b_S} + c_S}{a_S + c_S}\right)^{-1/b_N}. \tag{5.7}$$

The gains for our particular scaling coefficients are shown in Table 5.1 (Left). They are to be interpreted in the following way: for example, a 75% sparse ViT with N non-zeros will perform similar to a dense one with $\approx 2.17N$ parameters, when both are trained with *the same amount of data*. Crucially, this holds for *any* amount of data and thus also in the infinite limit when training is purely capacity bound. Hence, this expresses an equivalence between dense capacity and sparse capacity. We find that sparsity gains are very similar across vision and text domains, with the sweet-spot being around 75% sparsity at $\approx 2.15\times$ gain. This is in

alignment with the view of sparsity as a general increase in modeling power; we note that the gain is defined *relatively* to domain-specific improvements through increases in model size.

Family	0.500	0.750	0.875
ViT/JFT	1.60×	2.17×	2.63×
T5/C4	1.59×	2.16×	2.63×

Pattern	0.50	0.75
n:4	1.56×	1.62×
n:8	1.67×	1.81×

Table 5.1: (Left) Equivalent dense size multiplier to match performance of a sparse model. (Right) Dense size multipliers for n:m sparsity on T5/C4.

N:M sparsity. Complementing the *unstructured* sparsity exploration, we now also consider *structured* n:m sparsity, which can be accelerated on hardware [PY21, HCI+21]. Similar to how minor changes in the process (optimizer, model shape) generally only affect the multiplicative constants in dense scaling laws [KMH+20], we also expect minor changes in the sparsification process to only affect the sparsity term in (5.3). This can be exploited to fit laws based on significantly less runs: if the dense base scaling is known, one only has to fit a_S, b_S and c_S (just 3 rather than 7 parameters) to find the corresponding $L(S, N, D)$. We utilize this in the context of n:m sparsity by fitting new laws for 2:4 and 1:4 as well as 4:8 and 2:8 patterns, respectively, based only on a subset of our full grid in Appendix B.1.1. Concretely, we execute all runs involving either the least amount of steps, or the smallest model.

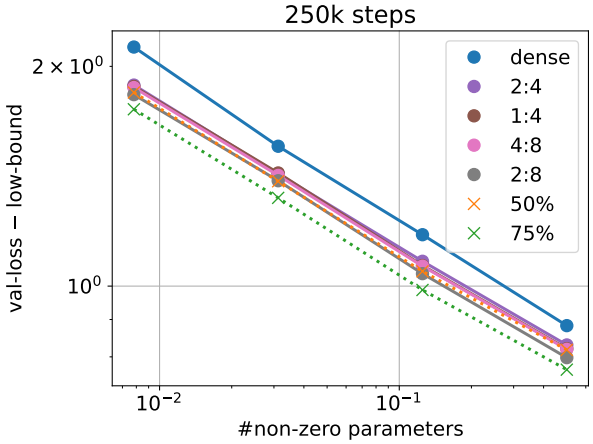


Figure 5.6: Loss vs. size plot for a subset of T5/C4 n:m sparsity data.

Figure 5.6 visualizes a subset of the collected data, displaying a very similar form to 5.2, which indicates that the general scaling law shape also holds for n:m sparsity. We also fit scaling laws (with Huber $\delta = 0.01$ as 0.75 patterns will otherwise be treated as an outlier) and calculate sparsity gains as in Section 5.1.4; see Table 5.1 (Right). In general, 2:4 and 4:8 perform both very similar to 50% (see Table 5.1 and also Figure 5.6), although the n:m estimates are slightly more noisy due to less data used in fitting the curves. Meanwhile, 1:4 brings almost no advantage and 2:8 only a slight improvement, which is contrary to our unstructured results. We suspect that the 75% patterns may simply be too stringent to significantly increase capacity beyond their 50% variants.

Decoder-only models. While we have focused on encoder-decoder models for language to cover a wide range of architectures, we now also validate our sparsity scaling laws in the context of standard decoder-only Transformers trained for auto-regressive language modeling.

We follow the same setup as in our T5/C4 experiments, changing only model architecture and loss function. We again execute the experiment grid defined in Table B.1 for 250K and 500K training steps, as well as a subset of the more expensive 1M step runs. Figure 5.7 demonstrates that all key sparse scaling properties still hold.

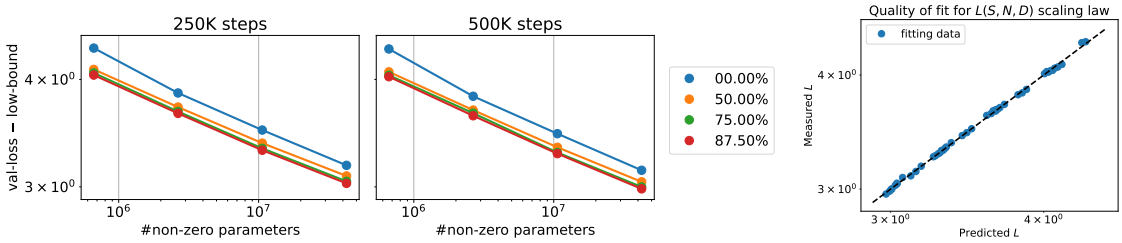


Figure 5.7: Visualization of decoder-only experiment and corresponding scaling fit quality.

Pruning pretrained models. Lastly, we consider a practical scenario where a set of existing *very well-trained* dense models should be made more efficient via pruning, using *a small fraction* of the compute spent for the initial pretraining. Our main interest here is to compare the efficiency of sparsifying from scratch and sparsifying from a pretrained checkpoint. For that purpose, we train ViT S/16, M/16 and B/16 models for 4 full epochs on JFT (i.e., 16 billion images) and then start the same gradual sparsification procedure we used before from these checkpoints, for 5.6% of the pretraining budget (as the model is already pretrained, we start to sparsify immediately rather than after 25% of training). Finally, we use our scaling laws from Section 5.1.3 to determine the amount of training necessary to produce equivalent models of the same quality when starting from scratch. Table 5.2 shows how much more/less data is required to achieve equivalent performance for sparsifying from scratch, when excluding/including the pretraining cost, respectively.

Model	0.500		0.750		0.875	
	exc.	inc.	exc.	inc.	exc.	inc.
S/16	4.90×	0.25×	4.27×	0.23×	2.45×	0.13×
M/16	4.76×	0.25×	4.18×	0.22×	2.57×	0.14×
B/16	4.35×	0.23×	4.00×	0.21×	2.72×	0.14×

Table 5.2: Relative amount of data required for sparsifying from scratch to match the validation loss of pruning from a pretrained model, when pretraining cost is excluded (exc.) and included (inc.).

If the model already exists and there is thus no pretraining cost, then starting from such a checkpoint is $> 4\times$ more efficient then sparsifying from scratch for 0.5/0.75, and $> 2\times$ for 0.875 sparsity, respectively. The reason why the efficiency gains are decreasing with higher sparsity is most likely the increased divergence from the initial starting point. At the same time, when the pretraining cost is counted as well, pruning throughout the whole training process appears to be $\geq 4\times$ more efficient, relative to the $\approx 5\%$ pruning of pretraining budget. Overall, these results clearly demonstrate that, while the sparsification process benefits significantly from a better trained initial model, it only does so to a limited extent.

5.1.5 Related Work

Sparsity & pruning. Sparsity has a long history [LDS89, HSW93] and a large number of works have been published on this topic [HABN+21]. State-of-the-art methods range from simple gradual removal of the smallest weights [ZG17], to partial or full sparse training [MMS+18, JPR+20, PIVA21], approximate Hessian-based metrics [SA20, FKA21, KKFA23] and "soft" sparse optimization [KRS+20, SWR20]. Sparsity can lead to substantial practical speedups with specialized inference algorithms [KKG+20, EDGS20]. Yet, most of those works focus on relatively simple tasks like ResNet50/ImageNet or BERT/GLUE.

In contrast, much less is known when it comes to sparsifying Transformers trained on massive datasets: The Appendix of Gopher [RBC+21] conducts pruning experiments for a generative language modelling task and finds that, when trained for the same amount of steps, sparse models can outperform dense ones, but leaves open whether this is also possible when accounting for the significantly increased compute spent for producing those sparse models, relative to equivalently sized dense ones. Similarly, [Cer22] prunes a GPT-like model, also using significantly more data than its dense baseline. Recently, SparseGPT [FA23] showed that it is possible to impose weight-sparsity on extremely large language models, even without retraining; yet, it remains unclear if this can also be done on smaller networks that are trained on more data.

Scaling laws. The key behind the success of Transformers is their exceptional scaling: increasing model size and/or data brings consistent performance improvements, even at huge scale. Further, this scaling behavior is predictable, following simple power-law curves [KMH+20]. This can be utilized to construct a family of compute optimal models [HBM+22]. More recently, scaling laws are being extended to more specialized applications, e.g.: optimizing model shapes [AZKB23], routing mechanisms [CDLCG+22], repeating training data multiple times [MRB+23] and several downstream tasks [CGRK23]. However, not much is known about the scaling of weight sparsity for such models.

[RFCS21] studies the relationship between width, depth and weight density for pruning pretrained ResNets trained primarily on the CIFAR dataset [KH+09], which is nowadays considered very small. Contrarily, we consider modern Transformers trained on datasets many orders of magnitude larger and focus particularly on the data/compute dimension that is crucial in this context, but not relevant in the setting of [RFCS21].

Transformer efficiency. Making (large) Transformers more efficient is currently a highly active area of research. Probably the currently most popular and practical approach is quantization, that is reducing the numerical precision [FAHA23, DZ23, XLS+23]. Further, there are also many works on Mixture-of-Expert (MoE) models, which bound the overall computation cost per sample [DHD+22, FZS22, ABG+22, RPM+21]. MoEs are a form of *dynamic activation* sparsity, which is very different from the *static weight* sparsity that we study; the former trades off increased memory for faster inference, whereas the latter reduces both inference *and* memory costs. In general, quantization, MoEs and weight sparsity are complementary and may be stacked for compound gains [KCN+22].

5.1.6 Discussion

Limitations. While our study is based on extensive experiments across models and domains, it also has limitations, which we discuss here and plan to address in future work. First, our

sparsification recipe was optimized for robustness and scalability across a wide range of setups, rather than to fully maximize sparsity versus accuracy in a particular setting. We believe that specific coefficient values can be improved with extensive setting-specific tuning and better sparsification techniques; however, the general nature of our scaling law will remain consistent.

We focus on settings where pruning is applied to pretraining tasks with massive amounts of data and compute. This is ideal in terms of usability, as down-stream (finetuning) applications directly benefit from the pruned model, but it also makes compression quite challenging. We think higher sparsity rates can be achieved if pruning is applied directly for specialized applications that only require a subset of the model's capabilities. Similarly, our study focuses on the infinite data setting, which essentially eliminates overfitting, as only a single pass over the data is performed. Sparsity could be particularly effective when data is limited and thus multiple epochs are performed.

Our proposed sparse scaling law suggests that higher sparsity is always better (but with potentially quite quickly saturating improvements), which may not be true in extremes. For very high sparsity (e.g., $64\times$ compression) we sometimes see slightly worse performance, presumably due to imperfections in the pruning and optimization process. This phenomenon could potentially be modelled by a quadratic, but the present study treats it as a bottleneck-case that is not necessarily captured.

Finally, the main goal of this study is understanding core scaling relationships. Thus, we focused on the most fundamental cost metric, non-zero parameter count. However, in practice, sparsity acceleration can be complex: current software/hardware may not provide ideal speedups and models generally also contain operations (e.g., layer-norms, attention) which do not benefit from weight sparsity. Extending our scaling results to more target metrics is an interesting topic for future work.

Compatibility with other works. We will now briefly discuss how our scaling insights line up with existing sparsification results on similar models/datasets. First, the results in the Appendix of Rae et al. [RBC+21], for a decoder-only text-generation model, are consistent with our scaling laws; the improvement through sparsity appears to be similar for each model size and their maximum size advantage of $2.5\times$ observed at 0.9 sparsity is quite close to our limit gains in Section 5.1.4.

In contrast, Cerebras [Cer22] reports a significantly better gain of $\approx 5\times$, but in a quite different setting where the baseline is training (not inference) compute optimal and sparsification uses $> 5\times$ more data than the dense comparison point. This is not inconsistent to our results: if we query our fitted T5 scaling law (see Section 5.1.3) with this setup, we predict 1.54 loss (dense 1B params, 20B tokens) vs. 1.48 loss (80% sparse & 200M non-zeros, 100B tokens), in favor of the sparse model.

Finally, SparseGPT [FA23] notes that post-training pruning becomes easier as the model size increases. However, they do not perform any retraining, and observe this effect primarily relative to the respective unpruned base model, not in terms of improvements over the Pareto size-vs-loss frontier that we focus on. Hence, we believe that this is likely more related to the pretrained models' initial robustness to pertubations rather than the architecture's inherent sparsifiability.

Practical implications. Our scaling insights lead to a number of practical consequences: Sparsity seems to affect each model size in approximately the same way, while remaining

mostly independent of the amount of training data used. This provides evidence that good pruning performance in less expensive settings should generalize to performance at scale, which will hopefully accelerate research on new sparsification recipes and algorithms. Additionally, we have shown that optimal sparsity levels continuously increase with longer training. Sparsity thus provides a means to further improve model performance for a fixed final parameter cost. In particular, when training beyond Chinchilla optimality, where simple dense training starts to run into diminishing returns, sparsity can provide a clear alternative. Thus, our findings can be interpreted as providing practical motivation for further developing sparsity support.

CHAPTER 6

Conclusion

In summary, this thesis studied the problem of making massive machine learning models more efficient from three major directions: developing new fast and accurate compression algorithms, implementing highly efficient systems solutions to translate theoretical gains into practical benefits and empirically understanding the scaling behavior of compression during training on a more fundamental level.

We started in Section 3.1 by developing OBC, a new post-training (that is free of any retraining) compression algorithm with state-of-the-art accuracy on smaller vision and text models, for both sparsity and quantization. A particularly interesting aspect of OBC is that it approached two very different forms of compression with the same algorithmic strategy, thus unifying them. Next, we heavily modified this algorithm in a way that yields several orders of magnitude speedup, while performing only slightly worse in terms of accuracy. This lead to GPTQ (see Section 3.2) and SparseGPT (see Section 3.3), the first methods fast and accurate enough to compress 100+ billion parameter models to 4- or even 3-bit precision and 50% sparsity, respectively. GPTQ and SparseGPT are highly practical, taking only $\approx$ 4 hours on a single GPU to compress such massive models, while coming with optimized GPU inference kernels that deliver up $4.5\times$ real-world generation speedup for single-user applications. In fact, our GPTQ kernel was the first practical (open-source) demonstration that standard weight-only quantization does not just bring memory savings but also latency reduction.

Next, we revisited the problem of writing highly efficient GPU kernels for mixed-precision inference, though this time in the context of a much more practical, but also much more challenging, multi-user setting. Section 4.1 described how to implement Marlin, an extremely optimized FP16 times INT4 matrix multiplication kernel, which for the first time delivers essentially ideal performance in the medium batchsize regime, simultaneously maximizing memory bandwidth and compute throughput. In Section 4.2, we presented additional systems optimizations to scale GPTQ, and similar methods, to even larger trillion-parameter models. In this context, we also co-designed a bespoke extreme compression format together with an efficient GPU kernel. This makes it possible to achieve an average size of 0.8 bits per parameter, on highly overparameterized models, while maintaining uncompressed inference speed.

Finally, Section 5.1 moved to a slightly different compression setting, where huge amounts of compute resources are available and maximizing accuracy is the exclusive goal. We conducted large amounts of pruning-during-training experiments to identify the first scaling law accurately

modeling the joint interaction of weight-sparsity, model-size and amount of training data used. Interestingly, we observed that sparsity seems to behave very regularly across scales. This allowed us to characterize the optimal sparsity level for every combination of target inference and training cost. Crucially, we found that the more and more budget is spent on training a fixed cost model, the higher the optimal sparsity becomes. This is particularly relevant for developing mobile models, where a small memory footprint is critical, while current recipes are already heavily overtraining such networks.

Overall, this thesis makes practical contributions on various angles of large model efficiency. Our fast and accurate compression algorithms are particularly useful for open-source, academia, or any individuals with only limited compute resources. Meanwhile, our scaling law insights are very relevant to corporations or other entities able to train massive models from scratch, which should then be served to large user-bases. Lastly, our highly efficient GPU inference kernels bring significant practical acceleration both in local (single-user) and large-scale serving settings. Nevertheless, there is still a lot of potential for future work in the area of this thesis; we outline a few especially interesting research directions below.

6.1 Future Work

Activation Compression. In this thesis, we have primarily focused on compressing model *weights*, leaving *activations* in standard precision. As we have shown, this is sufficient to accomplish significant speedup in memory-bound applications. However, not every LLM use-case is memory-bound: for example, prefilling a large prompt is completely compute-bound. If we were able to compress also the activations to lower bitwidth such operations could be accelerated as well. One major difference between activations and weights is that the former are *dynamic* whereas the latter are *static*. This means that any quantization must be applied *online*, directly during inference. Hence, it may not be possible to employ advanced rounding schemes like GPTQ, as their runtime may negate any benefits of the additional compression. Most existing activation compression schemes [EMB+19, JCM+21] utilize extensive (re-)training which would likely be far too expensive for larger models, suggesting that the development of entirely new approaches might be necessary. Another related problem is the compression of the KV-cache, which is comprised of activation vectors. This is probably a bit easier since the KV-cache is mostly involved in memory-bound operations [Sha19], where it is, as we have shown, feasible to efficiently support much more flexible and complex compression formats via custom CUDA kernels.

(Semi-)Automated Kernel Generation. As we have demonstrated with Marlin, it is possible to write near peak-performing kernels for compressed inference. However, doing so requires a lot of effort, with many meticulous low-level optimizations. It will be very difficult to sustain the development of such well optimized kernels across various compression formats and GPU architectures, due the amount of manual labor involved. For standard (uncompressed) kernels, (semi-)automated frameworks like Triton [TKC19] and JAX Pallas [BFH+18] are gaining in popularity. These make it possible to write many different types of accelerator kernels at a higher block-based level, while low-level detailed are automatically figured out by the compiler. Unfortunately, such libraries are currently not particularly well suited for compression use-cases. This is because reaching peak decompression performance often involves various tricks at the very lowest thread and/or register level, which is completely abstracted away and thus very hard to control explicitly. Figuring out either appropriate

extensions to current systems or completely new kinds of abstractions seems like a necessary step towards significantly speeding up compressed kernel development. Alternatively, it would be even better, yet likely much more difficult, if those low-level tricks could be automatically determined by the compiler.

Compression Scaling Laws. This thesis introduced scaling laws for one particular type of compression, weight-sparsity. A natural follow-up question to ask is whether the scaling phenomena we observed in this context also translate to other forms of compression, like weight quantization or various forms of activation compression. Additionally, this would be interesting from the perspective of comparing the practical efficiency of different compression schemes. Expanding on that, it might also be possible to jointly model finer-grained aspects of compression types (group-sizes, mantissa bits, sparsity pattern restrictions, etc.) in order to eventually discover new compression formats with better efficiency vs. accuracy trade-offs than conventional schemes. Another related direction could be exploring the scaling of compression with impact on training speed, like fully-quantized training [MSB+22]. In our specific weight-sparsity setting, compression only started to become optimal (and thus useful in practice) once a model is being "overtrained" [TLI+23]. However, if compression accelerates not just inference but also training, this may have potential to provide benefits in "undertrained" or "training-compute-optimal" [HBM+22] regimes as well.

Bibliography

[AAA+23] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[ABG+22] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. Efficient large scale language modeling with mixtures of experts. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[AGL+17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Randomized quantization for communication-efficient stochastic gradient descent. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[AMF+23] Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, and Dan Alistarh. Towards end-to-end 4-bit inference on generative large language models. *arXiv preprint arXiv:2310.09259*, 2023.

[AZKB23] Ibrahim Alabdulmohsin, Xiaohua Zhai, Alexander Kolesnikov, and Lucas Beyer. Getting ViT in shape: Scaling laws for compute-optimal model design. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[BCD+22] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ML. In *Conference on Machine Learning and Systems (MLSys)*, 2022.

[BCM+23] Riade Benbaki, Wenyu Chen, Xiang Meng, Hussein Hazimeh, Natalia Ponomareva, Zhe Zhao, and Rahul Mazumder. Fast as CHITA: Neural network pruning with combinatorial optimization. In *International Conference on Machine Learning (ICML)*, 2023.

[BD08] Thomas Blumensath and Mike E Davies. Iterative thresholding for sparse approximations. *Journal of Fourier Analysis and Applications*, 14(5-6):629–654, 2008.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. `http://github.com/google/jax`, 2018.

[BHA+21] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[BLC13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[BNS19] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[BPM+18] Michael Boratko, Harshit Padigela, Divyendra Mikkilineni, Pritish Yuvraj, Rajarshi Das, Andrew McCallum, Maria Chang, Achille Fokoue-Nkoutche, Pavan Kapanipathi, Nicholas Mattei, et al. A systematic classification of knowledge, reasoning, and context within the ARC dataset. *arXiv preprint arXiv:1806.00358*, 2018.

[CCKDS23] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. QuIP: 2-bit quantization of large language models with guarantees. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[CDLCG+22] Aidan Clark, Diego De Las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake Hechtman, Trevor Cai, Sebastian Borgeaud, et al. Unified scaling laws for routed language models. In *International Conference on Machine Learning (ICML)*, 2022.

[Cer22] Cerebras. Creating Sparse GPT-3 Models with Iterative Pruning. `https://www.cerebras.net/blog/creating-sparse-gpt-3-models-with-iterative-pruning`, 2022.

[CGRK23] Ethan Caballero, Kshitij Gupta, Irina Rish, and David Krueger. Broken neural scaling laws. In *International Conference on Learning Representations (ICLR)*, 2023.

[CHBS23] Brian Chmiel, Itay Hubara, Ron Banner, and Daniel Soudry. Optimal fine-grained N:M sparsity for activations and neural gradients. In *International Conference on Learning Representations (ICLR)*, 2023.

[CHX+22] Tianyu Chen, Shaohan Huang, Yuan Xie, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. Task-specific expert pruning for sparse mixture-of-experts. *arXiv preprint arXiv:2206.00277*, 2022.

[CKYK19] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *International Conference on Computer Vision Workshop (ICCVW)*, 2019.

[CND+23] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[Com23] Together Computer. RedPajama: An open source recipe to reproduce llama training dataset. `https://github.com/togethercomputer/RedPajama-Data`, 2023.

[DBK+21] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[DCP17] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[DDM+23] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Peter Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning (ICML)*, 2023.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[DFE+22] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[DHD+22] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. GLaM: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning (ICML)*, 2022.

[DLBZ22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[DPHZ23] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized llms. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[DSE+24] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh.

SpQR: A sparse-quantized representation for near-lossless llm weight compression. In *International Conference on Learning Representations (ICLR)*, 2024.

[DZ23] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning (ICML)*, 2023.

[EDGS20] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[EFS+23] Murali Emani, Sam Foreman, Varuni Sastry, Zhen Xie, Siddhisanket Raskar, William Arnold, Rajeev Thakur, Venkatram Vishwanath, and Michael E Papka. A comprehensive performance study of large language models on novel AI accelerators. *arXiv preprint arXiv:2310.04607*, 2023.

[EGM+20] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning (ICML)*, 2020.

[Ele22] EleutherAI. EleutherAI LM Evaluation Harness. `https://github.com/EleutherAI/lm-evaluation-harness`, 2022.

[ELRCB18] Utku Evci, Nicolas Le Roux, Pablo Castro, and Leon Bottou. Mean replacement pruning. *OpenReview*, 2018.

[EMB+19] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. In *International Conference on Learning Representations (ICLR)*, 2019.

[EPK+24] Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *International Conference on Machine Learning (ICML)*, 2024.

[FA22] Elias Frantar and Dan Alistarh. SPDY: Accurate pruning with speedup guarantees. In *International Conference on Machine Learning (ICML)*, 2022.

[FA23] Elias Frantar and Dan Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning (ICML)*, 2023.

[FA24a] Elias Frantar and Dan Alistarh. Marlin. `https://github.com/IST-DASLab/marlin/tree/master`, 2024.

[FA24b] Elias Frantar and Dan Alistarh. QMoE: Sub-1-bit compression of trillion parameter models. In *Machine Learning and Systems*, 2024.

[FAHA23] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training compression for generative pretrained transformers. In *International Conference on Learning Representations (ICLR)*, 2023.

[FCC+24] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. MARLIN: mixed-precision auto-regressive parallel inference on large language models. *arXiv preprint arXiv:2408.11743*, 2024.

[FKA21] Elias Frantar, Eldar Kurtic, and Dan Alistarh. M-FAC: Efficient matrix-free approximations of second-order information. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[FRH+24] Elias Frantar, Carlos Riquelme, Neil Houlsby, Dan Alistarh, and Utku Evci. Scaling laws for sparsely-connected foundation models. In *International Conference on Learning Representations (ICLR)*, 2024.

[FSA22] Elias Frantar, Sidak Pal Singh, and Dan Alistarh. Optimal Brain Compression: A framework for accurate post-training quantization and pruning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[FZS22] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

[GEH19] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. In *International Conference on Machine Learning (ICML)*, 2019.

[GKD+21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[GM16] Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning (ICML)*, 2016.

[GNYZ23] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. MegaBlocks: Efficient sparse training with mixture-of-experts. In *Conference on Machine Learning and Systems (MLSys)*, 2023.

[Goo23a] Google. Big vision. `https://github.com/google-research/big_vision`, 2023.

[Goo23b] Google. T5X. `https://github.com/google-research/t5x`, 2023.

[GZYE20] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[H+07] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2007.

[HABN+21] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

[Hag94] Masafumi Hagiwara. A simple and effective method for removal of hidden units and weights. *Neurocomputing*, 6(2):207 – 218, 1994.

[HBM+22] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[HCI+21] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Seffi Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find N:M transposable masks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[HCX+23] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. In *Conference on Machine Learning and Systems (MLSys)*, 2023.

[HLL+18] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *European Conference on Computer Vision (ECCV)*, 2018.

[HMD16] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

[HNH+20] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Improving post training neural quantization: Layer-wise calibration and integer programming. *arXiv preprint arXiv:2006.10518*, 2020.

[HNH+21] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Accurate post training quantization with small calibration sets. In *International Conference on Machine Learning (ICML)*, 2021.

[HPTD15] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[HSW93] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, 1993.

[Huf52] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[Hug22] HuggingFace. HuggingFace Perplexity Calculation. `https://huggingface.co/docs/transformers/perplexity`, 2022.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[HZS17] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, 2017.

[IPKA22] Eugenia Iofinova, Alexandra Peste, Mark Kurtz, and Dan Alistarh. How well do sparse ImageNet models transfer? In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[JCM+21] Sebastian Jaszczur, Aakanksha Chowdhery, Afroz Mohiuddin, Lukasz Kaiser, Wojciech Gajewski, Henryk Michalewski, and Jonni Kanerva. Sparse is enough in scaling transformers. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[JCR+22] Tian Jin, Michael Carbin, Dan Roy, Jonathan Frankle, and Gintare Karolina Dziugaite. Pruning's effect on generalization through the lens of training and regularization. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[JKC+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[Joc22] Glenn Jocher. YOLOv5. `https://github.com/ultralytics/yolov5`, 2022.

[JPR+20] Siddhant M Jayakumar, Razvan Pascanu, Jack W Rae, Simon Osindero, and Erich Elsen. Top-KAST: Top-K always sparse training. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[JYP+17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[KA22] Eldar Kurtic and Dan Alistarh. GMP*: Well-tuned global magnitude pruning can outperform most bert-pruning methods. *arXiv preprint arXiv:2210.06384*, 2022.

[KCN+22] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The Optimal BERT Surgeon: Scalable and accurate second-order pruning for large language models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[KFA22] Young Jin Kim, Raffy Fahim, and Hany Hassan Awadalla. Mixture of quantized experts (MoQE): Complementary effect of low-bit quantization and robustness. *OpenReview*, 2022.

[KFA24] Eldar Kurtic, Elias Frantar, and Dan Alistarh. Ziplm: Inference-aware structured pruning of language models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[KH+09] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. *CiteSeer*, 2009.

[KHFA22] Young Jin Kim, Rawn Henry, Raffy Fahim, and Hany Hassan Awadalla. Who says elephants can't run: Bringing large scale moe models into cloud scale production. *arXiv preprint arXiv:2211.10017*, 2022.

[KHFA23] Young Jin Kim, Rawn Henry, Raffy Fahim, and Hany Hassan Awadalla. Finequant: Unlocking efficiency with fine-grained weight-only quantization for llms. *arXiv preprint arXiv:2308.09723*, 2023.

[Kin97] Jason Kingdon. *Hypothesising Neural Nets*, pages 81–106. Springer London, 1997.

[KKF+23] Eldar Kurtic, Denis Kuznedelev, Elias Frantar, Michael Goin, and Dan Alistarh. Sparse finetuning for inference acceleration of large language models. *arXiv preprint arXiv:2310.06927*, 2023.

[KKFA23] Denis Kuznedelev, Eldar Kurtic, Elias Frantar, and Dan Alistarh. CAP: correlation-aware pruning for highly-accurate sparse vision models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[KKG+20] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Bill Nell, Nir Shavit, and Dan Alistarh. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *International Conference on Machine Learning (ICML)*, 2020.

[KKI+23] Denis Kuznedelev, Eldar Kurtic, Eugenia Iofinova, Elias Frantar, Alexandra Peste, and Dan Alistarh. Accurate neural network pruning requires rethinking sparse optimization. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[KKM+22] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[KMH+20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[KNB23] Yeskendir Koishekenov, Vassilina Nikoulina, and Alexandre Berard. Memory-efficient NLLB-200: Language-specific expert pruning of a massively multilingual machine translation model. In *Annual Meeting of the Association for Computational Linguistics*, 2023.

[KRS+20] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity. In *International Conference on Machine Learning (ICML)*, 2020.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2012.

[LAS+23] Yucheng Lu, Shivani Agrawal, Suvinay Subramanian, Oleg Rybakov, Christopher De Sa, and Amir Yazdanbakhsh. STEP: Learning n: M structured sparsity masks from scratch with precondition. In *International Conference on Machine Learning (ICML)*, 2023.

[LDS89] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Conference on Neural Information Processing Systems (NeurIPS)*, 1989.

[LGT+21] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. BRECQ: Pushing the limit of post-training quantization by block reconstruction. In *International Conference on Learning Representations (ICLR)*, 2021.

[LGW+21] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[LKM23] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*, 2023.

[LLX+20] D Lepikhin, H Lee, Y Xu, D Chen, O Firat, Y Huang, M Krikun, and N Shazeer. GShard: scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision (ECCV)*, 2014.

[LPM+23] Joo Hyung Lee, Wonpyo Park, Nicole Mitchell, Jonathan Pilault, Johan Obando-Ceron, Han-Byul Kim, Namhoon Lee, Elias Frantar, Yun Long, Amir Yazdanbakhsh, et al. JaxPruner: A concise library for sparsity research. In *Conference on Parsimony and Learning (CPAL)*, 2023.

[LSB+20] Tao Lin, Sebastian U Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. In *International Conference on Learning Representations (ICLR)*, 2020.

[LSW+22] Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. The BigScience corpus: A 1.6 TB composite multilingual dataset. *arXiv preprint arXiv:2303.03915*, 2022.

[LSZ+18] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations (ICLR)*, 2018.

[LTT+24] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: Activation-aware weight quantization for llm compression and acceleration. In *Conference on Machine Learning and Systems (MLSys)*, 2024.

[LZK+21] Liyang Liu, Shilong Zhang, Zhanghui Kuang, Aojun Zhou, Jing-Hao Xue, Xinjiang Wang, Yimin Chen, Wenming Yang, Qingmin Liao, and Wayne Zhang. Group fisher pruning for practical network compression. In *International Conference on Machine Learning (ICML)*, 2021.

[LZW+23] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, et al. Efficient GPU kernels for n:m-sparse weights in deep learning. In *Conference on Machine Learning and Systems (MLSys)*, 2023.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

[MAFA24] Ilia Markov, Kaveh Alimohammadi, Elias Frantar, and Dan Alistarh. L-GreCo: Layerwise-adaptive gradient compression for efficient data-parallel deep learning. In *Conference on Machine Learning and Systems (MLSys)*, 2024.

[MG15] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International Conference on Machine Learning (ICML)*, 2015.

[MIFA24] Arshia Soltani Moakhar, Eugenia Iofinova, Elias Frantar, and Dan Alistarh. SPADE: Sparsity-guided debugging for deep neural networks. In *International Conference on Machine Learning (ICML)*, 2024.

[MKK+24] Ionut-Vlad Modoranu, Aleksei Kalinov, Eldar Kurtic, Elias Frantar, and Dan Alistarh. Error feedback can accurately compress preconditioners. In *International Conference on Machine Learning (ICML)*, 2024.

[MKM+94] Mitch Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The Penn treebank: Annotating predicate argument structure. In *Human Language Technology, New Jersey, 1994*, 1994.

[MLP+21] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

[MMS+18] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12, 2018.

[MR10] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *ACM International Conference on Multimedia*, 2010.

[MRB+23] Niklas Muennighoff, Alexander M Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. Scaling data-constrained language models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[MRL+17] Nasrin Mostafazadeh, Michael Roth, Annie Louis, Nathanael Chambers, and James Allen. Lsdsem 2017 shared task: The story cloze test. In *Proceedings of the 2nd Workshop on Linking Models of Lexical, Sentential and Discourse-level Semantics*, pages 46–51, 2017.

[MSB+22] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. FP8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[MXBS17] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017.

[NAVB+20] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? Adaptive rounding for post-training quantization. In *International Conference on Machine Learning (ICML)*, 2020.

[NBBW19] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *International Conference on Computer Vision (ICCV)*, 2019.

[NCB+21] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson. Loss aware post-training quantization. *Machine Learning*, 110(11):3245–3262, 2021.

[Neu22] NeuralMagic. DeepSparse. `https://github.com/neuralmagic/deepsparse`, 2022.

[NFA+21] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[NPI+23] Mahdi Nikdan, Tommaso Pegolotti, Eugenia Iofinova, Eldar Kurtic, and Dan Alistarh. SparseProp: efficient sparse backpropagation for faster training of neural networks. In *International Conference on Machine Learning (ICML)*, 2023.

[NVI20] NVIDIA. NVIDIA A100 tensor core GPU architecture. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020.

[NVI22] NVIDIA. Nvidia a10 datasheet. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a10/pdf/datasheet-new/nvidia-a10-datasheet.pdf`, 2022.

[NVI24a] NVIDIA. CUTLASS convolution. `https://github.com/NVIDIA/cutlass/blob/main/media/docs/implicit_gemm_convolution.md`, 2024.

[NVI24b] NVIDIA. Efficient GEMM in CUDA. `https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md`, 2024.

[OMC+23] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the GPU. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[PIVA21] Alexandra Peste, Eugenia Iofinova, Adrian Vladu, and Dan Alistarh. AC/DC: Alternating compressed/decompressed training of deep neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[PKL+16] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

[PPK+22] Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuQmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.

[PRMH24] Joan Puigcerver, Carlos Riquelme, Basil Mustafa, and Neil Houlsby. From sparse to soft mixtures of experts. In *International Conference on Learning Representations (ICLR)*, 2024.

[PY21] Jeff Pool and Chong Yu. Channel permutations for N:M sparsity. *Conference on Neural Information Processing Systems (NeurIPS)*, 34, 2021.

[PYNC22] Minseop Park, Jaeseong You, Markus Nagel, and Simyung Chang. Quadapter: Adapter for GPT-2 quantization. *arXiv preprint arXiv:2211.16912*, 2022.

[RBC+21] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training Gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[RCK+20] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[RFCS21] Jonathan S Rosenfeld, Jonathan Frankle, Michael Carbin, and Nir Shavit. On the predictability of pruning across scales. In *International Conference on Machine Learning (ICML)*, 2021.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[RKX+23] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning (ICML)*, 2023.

[RPM+21] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[RSR+20a] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[RSR+20b] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)*, 21(1):5485–5551, 2020.

[RST+24] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[RSW+21] Stephen Roller, Sainbayar Sukhbaatar, Jason Weston, et al. Hash layers for large sparse models. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[RWC+19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019.

[RZLL16] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[SA20] Sidak Pal Singh and Dan Alistarh. WoodFisher: Efficient second-order approximation for neural network compression. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[SCP+18] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[SFA+22] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[Sha19] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Findings of the Association for Computational Linguistics (ACL)*, 2016.

[SJP+21] Jonathan Schwarz, Siddhant Jayakumar, Razvan Pascanu, Peter Latham, and Yee Teh. Powerpropagation: A sparsity inducing weight reparameterisation. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[SLG+22] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, 2022.

[SS18] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning (ICML)*, 2018.

[SWR20] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[TGM+23] Vithursan Thangarasa, Abhay Gupta, William Marshall, Tianda Li, Kevin Leong, Dennis DeCoste, Sean Lie, and Shreyas Saxena. SPDF: Sparse pre-training and dense fine-tuning for large language models. *arXiv preprint arXiv:2303.10464*, 2023.

[TKC19] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMa: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[TMS+23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[TP03] Sandeep Tata and Jignesh M Patel. PiQA: An algebra for querying protein data sets. In *International Conference on Scientific and Statistical Database Management*, 2003.

[VDOVK17] Aaron Van Den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[WCHC20] Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. Towards accurate post-training network quantization via bit-split and stitching. In *International Conference on Machine Learning (ICML)*, 2020.

[WDS+19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[Wel84] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(06):8–19, 1984.

[WGFZ19] Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. Eigendamage: Structured pruning in the Kronecker-factored eigenbasis. In *International Conference on Machine Learning (ICML)*, 2019.

[WLL+19] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-aware automated quantization with mixed precision. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[WYH23] Xiaoxia Wu, Zhewei Yao, and Yuxiong He. ZeroQuant-FP: A leap forward in llms post-training w4a8 quantization using floating-point formats. *arXiv preprint arXiv:2307.09782*, 2023.

[WYZ+22] Xiaoxia Wu, Zhewei Yao, Minjia Zhang, Conglong Li, and Yuxiong He. XTC: Extreme compression for pre-trained transformers made simple and efficient. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[WZG20] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations (ICLR)*, 2020.

[XLS+23] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*, 2023.

[XZC22] Mengzhou Xia, Zexuan Zhong, and Danqi Chen. Structured pruning learns compact and accurate models. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2022.

[XZL+23] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-LLM: enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. In *VLDB Endowment*, 2023.

[YAZ+22] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[YCG23] Chong Yu, Tao Chen, and Zhongxue Gan. Boost transformer-based language models with GPU-friendly sparsity and quantization. In *Findings of the Association for Computational Linguistics: ACL 2023*, 2023.

[YDZ+21] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al. HAWQ-v3: Dyadic neural network quantization. In *International Conference on Machine Learning (ICML)*, 2021.

[YGW+23] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. EdgeMoE: Fast on-device inference of MoE-based large language models. *arXiv preprint arXiv:2308.14352*, 2023.

[YGZL20] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[ZBK+22] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. ST-MoE: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*, 2022.

[ZG17] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[ZKHB22] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[ZLL+22] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. Mixture-of-experts with expert choice routing. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[ZLZ+22] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[ZMZ+21] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning N:M fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations (ICLR)*, 2021.

[ZRG+22] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[ZS19] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

APPENDIX A

Algorithms

A.1 Optimal Brain Compression: Accurate Post-Training Quantization and Pruning

A.1.1 Proof of Row & Column Removal Lemma

Proof. First, we observe that element j in row i, i.e. $[\mathbf{A}]_{ij}$, is set to 0 by the equivalent matrix transformation of subtracting $[\mathbf{A}]_{ij}$ times column i denoted by $\mathbf{A}_{:,i}$ divided by the corresponding diagonal element $[\mathbf{A}]_{ii}$ (similarly, elements in column i can be set to 0 by subtracting row i). Thus, Lemma 1 corresponds to zeroing $\mathbf{H}^{-1}_{pi}$ and $\mathbf{H}^{-1}_{ip}$ for $i \neq p$ via equivalent matrix transformations, or in other words, Gaussian elimination of one row and column.

Next, we apply these equivalent matrix transformations to both sides of the obvious equality $\mathbf{H}^{-1}\mathbf{H} = \mathbf{I}$, which ultimately gives an equation of the following $\mathbf{AB} = \mathbf{C}$ form:

$$\begin{bmatrix} \mathbf{A_1} & \mathbf{0} & \mathbf{A_2} \\ \mathbf{0}^\top & a & \mathbf{0}^\top \\ \mathbf{A_4} & \mathbf{0} & \mathbf{A_3} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B_1} & \mathbf{b_1} & \mathbf{B_2} \\ \mathbf{b_4}^\top & b & \mathbf{b_2}^\top \\ \mathbf{B_4} & \mathbf{b_3} & \mathbf{B_3} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{c_1} & \mathbf{0} \\ \mathbf{c_4}^\top & c & \mathbf{c_2}^\top \\ \mathbf{0} & \mathbf{c_3} & \mathbf{I} \end{bmatrix}. \tag{A.1}$$

Notice now that the entries of $\mathbf{B}$ corresponding to the eliminated row and column in $\mathbf{A}$ do not affect the $\mathbf{I}$ and $\mathbf{0}$ blocks in $\mathbf{C}$ since they are always multipled by 0. Thus, the matrix of the $\mathbf{A_i}$ blocks must be the inverse of the $\mathbf{B_i}$ block matrix, which is exactly what we wanted to calculate. □

A.1.2 OBQ-ExactOBS Algorithm Pseudocode

The OBQ version of the ExactOBS algorithm is given below; we emphasize the similarity to the pruning variant of ExactOBS shown in Algorithm 3.1.

Algorithm A.1: Quantize $k \leq d_{\text{col}}$ weights from row $\mathbf{w}$ with inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top)^{-1}$ according to OBS in $O(k \cdot d_{\text{col}}^2)$ time.

$M = \{1, \ldots, d_{\text{col}}\}$
for $i = 1, \ldots, k$ **do**
  $p \leftarrow \text{argmin}_{p \in M} \frac{1}{[\mathbf{H}^{-1}]_{pp}} \cdot (q(w_p) - w_p)^2$
  $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{H}^{-1}_{:,p} \frac{1}{[\mathbf{H}^{-1}]_{pp}} \cdot (w_p - q(w_p))$
  $\mathbf{H}^{-1} \leftarrow \mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{pp}} \mathbf{H}^{-1}_{:,p} \mathbf{H}^{-1}_{p,:}$
  $M \leftarrow M - \{p\}$
end for

A.1.3 Timing Information

In this section, we provide detailed information about the runtime of our method. All numbers reported here are for the execution on a single NVIDIA RTX 3090 GPU using our PyTorch implementations. Pruning runs with a global step are performed with the "less compute" variant described in Figure 3.1. Hence an entire database of *many* pruning levels can be generated in approximately the time shown for unstructured and block pruning runs here.

PTQ Runtime Comparison. We begin with a runtime comparison of existing state-of-the-art post-training methods at the task of quantizating the weights of all layers of a ResNet50 to 4 bits. All timings were collected by executing the authors' open-source implementations on the same hardware, the results are shown in Table A.1.

Model	BitSplit	AdaRound	AdaQuant	BRECQ	OBQ
ResNet50	124m	55m	17m	53m	65m

Table A.1: Runtimes of post-training quantization methods in minutes (m).

BRECQ, AdaRound and our method OBQ all take around one hour to fully quantize ResNet50, the former two slightly less and the latter slightly more. Meanwhile, BitSplit takes about twice as long, whereas AdaQuant is $3\times$ faster. However, as shown in Table 3.5 in the main text, AdaQuant is also considerably less accurate than the other methods. In summary, the runtime of ExactOBS is in line with existing post-training methods. Additional optimizations, like periodically shrinking the Hessian by omitting rows/columns of pruned/quantized weights, can likely improve the practical speed further.

Different Compression Types. Next, we study the runtime of ExactOBS applied to different types of compression problems. We consider a smaller model (YOLOv5s), a medium model (ResNet50) and a larger one (BERT). The corresponding runtimes for all compression types featured in this work are listed in Table A.2.

In general, we can see that quantization and unstructured pruning take about the same time, which matches with the fact that the corresponding algorithms are very similar. Correspondingly, 2:4 pruning and quantizing a 2:4 pruned model are only approximately half as expensive, which is again expected as they perform half the work. For YOLO and BERT, blocked pruning is the most expensive compression type due to the overheads incurred by handling the additional

Model	Quant	Unstr	4-block	2:4	Quant 2:4
ResNet50	65m	64m	61m	31m	35m
YOLOv5s	7m	6m	10m	3m	4m
BERT	111m	103m	142m	51m	56m

Table A.2: Runtimes of ExactOBS for different models and compression types in minutes (m).

$c \times c$ block matrices (see Section 3.1.4). Interestingly, for ResNet50, this is not the case, which is probably related to the highly non-uniform compute distribution that is discussed in more detail in the next paragraph. Overall, these results show that our techniques are quick for small models and still reasonably efficient even for bigger models like BERT, taking less than 2 hours on a single GPU. Finally, we note that ExactOBS is essentially perfectly parallelizable and its runtime can thus scale linearly with the number of available GPUs.

Per-Layer Runtimes. Finally, we note that as the time complexity of OBQ implemented via ExactOBS is $O(d_{\text{row}} \cdot d_{\text{col}}^3)$, i.e. cubic in the column dimension, the overall runtime can often be dominated by a few particularly large layers. This is illustrated e.g. by ResNet50 where, as shown in Figure A.1, about 75% of the overall runtime is spent in the 3×3 convolutions of the last block (which have $d_{\text{col}} \approx 4500$ when unfolded), of which there are just 3 in total. Meanwhile, most of the earlier layers are quantized within seconds. This means that one could, in many cases, reduce the overall compression runtime significantly by applying a faster but less accurate method to just those few bottleneck layers while still achieving more accurate compression on all the others through our techniques.

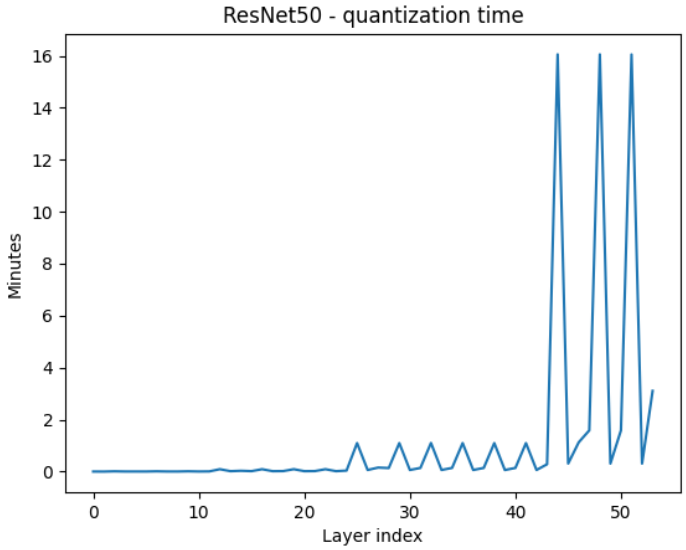


Figure A.1: Runtime of OBQ for each layer of ResNet50.

A.2 GPTQ: Post-Training Quantization for Generative Pre-Trained Transformers

A.2.1 Additional Language Generation Results

Tables A.3, A.4, A.5 and A.6 show additional results for language generation tasks.

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	38.99	31.08	20.29	17.97	15.77	14.52	14.04	13.36	12.01
RTN	4	53.89	36.79	57.30	31.05	18.84	16.51	15.40	225.66	14.22
GPTQ	4	**45.17**	**34.52**	**21.85**	**19.14**	**16.56**	**14.94**	**14.26**	**13.81**	**12.26**
RTN	3	1.4e3	88.04	1.3e4	1.4e4	5.7e3	2.8e3	1.2e3	5.0e3	8.0e3
GPTQ	3	**73.19**	**47.08**	**32.10**	**24.81**	**21.88**	**16.68**	**15.36**	**28.12**	**12.86**

Table A.3: OPT perplexity results on PTB.

BLOOM	Bits	560M	1.1B	1.7B	3B	7.1B	176B
full	16	43.69	57.96	30.00	25.34	20.83	14.59
RTN	4	51.10	66.85	33.58	27.68	22.42	15.00
GPTQ	4	**46.97**	**62.47**	**31.84**	**26.49**	**21.67**	**14.75**
RTN	3	126.	185.	106.	66.78	35.04	107.
GPTQ	3	**70.35**	**87.04**	**46.11**	**34.02**	**26.14**	**15.57**

Table A.4: BLOOM perplexity results for PTB.

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	26.56	22.59	16.07	14.34	12.71	12.06	11.44	10.99	10.13
RTN	4	33.91	26.21	24.51	18.43	14.36	13.36	13.46	309.	11.61
GPTQ	4	**29.22**	**24.63**	**16.97**	**15.00**	**13.18**	**12.26**	**11.57**	**11.23**	**10.28**
RTN	3	834	55.49	5.2e3	1.1e4	5.3e3	3.1e3	1.4e3	3.5e3	4.6e3
GPTQ	3	**42.41**	**31.33**	**21.63**	**18.17**	**17.14**	**13.34**	**12.23**	**14.59**	**10.67**

Table A.5: OPT perplexity results on C4. We note that the calibration data used by GPTQ is sampled from the C4 training set, this task is thus not fully zero-shot.

A.2.2 Timing Experiment Setup

Our timing experiments are performed following the standard HuggingFace/accelerate[1] setup also used by the recent work LLM.int8() [DLBZ22]. In this setting, the model is split by distributing chunks of consecutive layers across GPUs. Importantly, in this setup the communication costs are minimal, $< 5\%$ of the total runtime even when working with 8 GPUs. This means almost all of the reported speedups are due to our quantized-matrix full-precision

[1] `https://huggingface.co/docs/accelerate/index`

BLOOM	Bits	560M	1.1B	1.7B	3B	7.1B	176B
full	16	26.60	22.05	19.49	17.49	15.20	11.71
RTN	4	29.89	24.44	21.26	18.76	16.06	12.04
GPTQ	4	**28.00**	**23.25**	**20.55**	**18.10**	**15.60**	**11.81**
RTN	3	67.49	60.71	113.	80.49	22.59	598.
GPTQ	3	**35.78**	**28.83**	**25.34**	**21.25**	**17.67**	**12.27**

Table A.6: BLOOM perplexity results for C4. We note that the calibration data used by GPTQ is sampled from the C4 training set, this task is thus not fully zero-shot.

vector product kernels. We emphasize that the only difference between the FP16 baseline and our quantized models are the kernels used to perform the underlying matrix-vector products.

This means all overheads due to HuggingFace, attention or non-quantized operations like residuals or LayerNorms are exactly the same. Consequently, our quantized models should benefit from more advanced distribution strategies [ZLZ+22] or more efficient attention kernels [DFE+22] just as much as our baseline.

In general, our kernels target generative inference in the low batch-size setting (for simplicity, we consider only batchsize 1) where the underlying (close to) matrix-vector products are memory-bound. For non-generative and large-batch applications, operations may be compute- rather than memory-bound and our kernels thus not directly applicable. Instead, one could simply decompress the matrix before performing the corresponding matrix-matrix calculations: this takes $<$ 1.5ms on an A100 and $<$ 3ms on an A6000 compared to 76ms/365ms for the subsequent OPT-175B FC2 layer computation with batchsize 16×1024 tokens. Hence, for such applications our methods significantly reduce the required number of GPUs at very little computational overhead. This is similar to recent work [DLBZ22], but we achieve a $2.5\times$ higher compression rate.

A.3 SparseGPT: Massive Language Models Can Be Accurately Pruned

A.3.1 Ablation Studies

In this section, we conduct ablations studies with respect to several of the main parameters of `SparseGPT`. For a fast iteration time and making it possible to also explore more compute and memory intensive settings, we focus on the OPT-2.7B model here. Unless stated otherwise, we always prune uniformly to the default 50% sparsity. For brevity we only show raw-WikiText2 results here, but would like to note that the behavior on other datasets is very similar.

Amount of Calibration Data. First, we investigate how the accuracy of `SparseGPT` scales with the number calibration data samples, which we vary in powers of two. The results are shown in Figure A.2. Curiously, `SparseGPT` is already able to achieve decent results even with just a few 2048-token segments; using more samples however yields significant further improvements, but only up to a certain point as the curve flattens quite quickly. Thus, since using more samples also increases compute and memory costs, we stick to 128 samples in all our experiments.

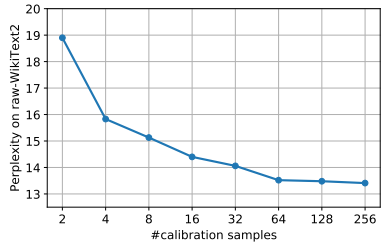


Figure A.2: Calibration samples ablation.

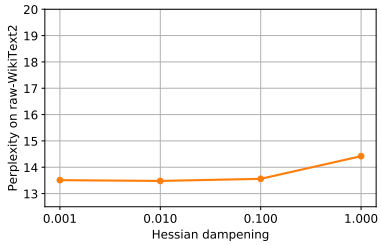


Figure A.3: Hessian dampening ablation.

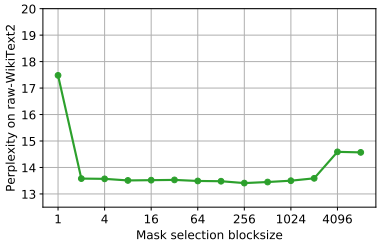


Figure A.4: Mask select. blocksize ablation.

Hessian Dampening. Next, we study the impact of Hessian dampening by testing values varying as powers of ten (see Figure A.3) which are multiplied by the average diagonal value, following [FAHA23]. Overall, this parameter does not seem to be too sensitive, 0.001 to 0.1 appear to perform quite similar; only when the dampening is very high, the solution quality decreases significantly. We choose 1% (i.e. 0.01) dampening to be on the safe side with respect to inverse calculations also for the very largest models.

Mask Selection Blocksize. Another important component of our method is the adaptive mask selection as shown in Figure A.4 where we vary the corresponding blocksize parameter with powers of two. Both column-wise (blocksize 1) as well as near full blocksize (4096 and 8192) perform significantly worse than reasonable blocking. Interestingly, a wide range of block-sizes appear to work well, with ones around a few hundred being very slightly more accurate. We thus choose blocksize 128 which lies in that range while also slightly simplifying the algorithm implementation as it matches the default lazy weight update batchsize.

Sensitivity to Random Seeds. Finally, we determine how sensitive the results of our algorithm are with respect to randomness; specifically, relative to the random sampling of the calibration data. We repeat a standard 50% pruning run 5 times with different random seeds for data sampling and get 13.52 ± 0.075 (mean/std) suggesting that `SparseGPT` is quite robust to the precise calibration data being used, which is in line with the observations in other post-training works [NAVB+20, HNH+21, FSA22].

A.3.2 Approximation Quality

In this section we investigate how much is lost by the partial-update approximation employed by `SparseGPT`, relative to (much more expensive) exact reconstruction. We again consider the OPT-2.7B model at 50% sparsity and plot the layer-wise squared error of `SparseGPT` relative to the error of exact reconstruction (with the same mask and Hessian) for the first half of the model in Figure A.5. Apart from some outliers in form of the early attention out-projection layers, the final reconstruction errors of `SparseGPT` seem to be on average only around 20% worse than exact reconstruction; on the later fully-connected-2 layers, the approximation error even gets close to only 10%, presumably because these layers have a very large number of total inputs and thus losses by considering only correlations within subsets are less severe than on smaller layers. Overall, these results suggest that, despite its dramatic speedup, `SparseGPT` also remains quite accurate.

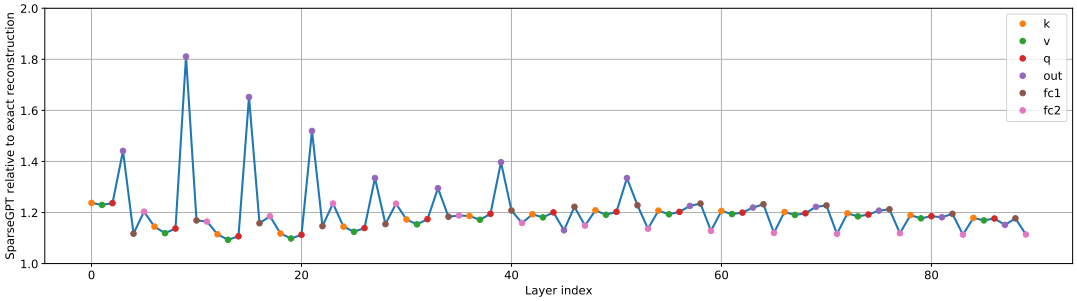


Figure A.5: Error of SparseGPT reconstruction relative to exact reconstruction for the first half of OPT-2.7B at 50% sparsity.

A.3.3 Evaluation Details

Perplexity. As mentioned in the main text, our perplexity calculation is carried out in standard fashion, following exactly the description of [Hug22]. Concretely, that means we concatenate all samples in the test/validation dataset, encode the result with the model's matching tokenizer and then split it into non-overlapping segments of 2048 tokens (the maximum history of the models we study). Those are run through the model to calculate the corresponding average language modelling loss. The exponentiated number is the perplexity we report.

Datasets. In terms of datasets, we use the raw version of the WikiText2 test-set and concatenate samples, as recommended by the HuggingFace description referenced above, with "\n\n" to produce properly formatted markdown. For PTB, we use the test-set of HuggingFace's "ptb_text_only" version and concatenate samples directly, without separators, as PTB is not supposed to contain any punctuation. Our C4 subset consists of the starting (the dataset comes in random order) 256 times 2048 encoded tokens in the first shard of the directly concatenated validation set; this choice is made to keep evaluation costs manageable.

A.3.4 Additional Results

Pruning Difficulty Scaling on PTB & C4. Tables A.7 and A.8 present the equivalent results to Table 3.9 in the main text, but on PTB and our C4 subset, respectively. Overall, they follow very similar trends to those discussed in Section 3.3.4. The main notable difference is that no slight perplexity decrease relative to the dense baseline is observed at 50% sparsity for the largest models, hence we have labelled this as a dataset specific phenomenon.

50% Sparse + 3-bit. The main paper only presents near loss-less results for 50% + 4-bit joint sparsification and quantization, corresponding to 3-bit quantization in terms of storage. For 50% + 3-bit (corresponding to 2.5-bit), OPT-175B achieves 8.60 PPL on raw-WikiText2, which is also more accurate than GPTQ's [FAHA23] 8.94 state-of-the-art 2.5-bit result. SparseGPT scores the same 8.93 for 4:8 + 3-bit. Based on these initial investigations, we believe that combining sparsity + quantization is a promising direction towards even more extreme compression of very large language models.

Table A.7: OPT perplexity results on PTB.

OPT - 50%	125M	350M	1.3B
Dense	38.99	31.07	20.29
Magnitude	276.	126.	3.1e3
AdaPrune	92.14	64.64	41.60
SparseGPT	**55.06**	**43.80**	**25.80**

OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	17.97	15.77	14.52	14.04	13.36	12.01
Magnitude	50%	262.	613.	1.8e4	221.	4.0e3	2.3e3
SparseGPT	50%	**20.45**	**17.44**	**15.97**	**14.98**	**14.15**	**12.37**
SparseGPT	4:8	23.02	18.84	17.23	15.68	14.68	12.78
SparseGPT	2:4	26.88	21.57	18.71	16.62	15.41	13.24

OPT - 50%	125M	350M	1.3B
Dense	26.56	22.59	16.07
Magnitude	141.	77.04	403.
AdaPrune	48.84	39.15	28.56
SparseGPT	**33.42**	**29.18**	**19.36**

OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	14.32	12.71	12.06	11.45	10.99	10.13
Magnitude	50%	63.43	334.	1.1e4	98.49	2.9e3	1.7e3
SparseGPT	50%	**15.78**	**13.73**	**12.97**	**11.97**	**11.41**	**10.36**
SparseGPT	4:8	17.21	14.77	13.76	12.48	11.77	10.61
SparseGPT	2:4	19.36	16.40	14.85	13.17	12.25	10.92

Table A.8: OPT perplexity results on a C4 subset.

A.3.5 Partial 2:4 Results

Tables A.9 and A.10 show the performance of a sequence of partially 2:4 sparse models on three different language modelling datasets. The first fraction of layers is fully sparsified while the remainder is kept dense. In this way, speedup and accuracy can be traded off also from binary compression choices, such as n:m-pruning.

OPT-175B – 2:4	dense	1/2	2/3	3/4	4/5	full
raw-WikiText2	8.34	8.22	8.38	8.49	8.52	8.74
PTB	12.01	12.15	12.80	13.02	13.12	13.25
C4-subset	10.13	10.22	10.41	10.52	10.59	10.92

Table A.9: Pruning different fractions (as consecutive segments from the beginning) of OPT-175B layers to the 2:4 pattern.

BLOOM-176B – 2:4	dense	1/2	2/3	3/4	4/5	full
raw-WikiText2	8.11	8.20	8.50	8.67	8.74	9.20
PTB	14.58	14.78	15.44	15.84	15.96	16.42
C4-subset	11.71	11.81	12.06	12.23	12.32	12.67

Table A.10: Pruning different fractions (as consecutive segments from the beginning) of BLOOM-176B layers to the 2:4 pattern.

A.3.6 Sparsity Acceleration

Lastly, we perform a preliminary study of how well sparse language models can already be accelerated in practice with off-the-shelf tools, for both CPU and GPU inference. We think that these results can likely be improved significantly with more model specific optimization, which we think is an important topic for future work.

CPU Speedups. First, we investigate acceleration of *unstructured* sparsity for CPU inference. For that we utilize the state-of-the-art DeepSparse engine [Neu22] and run end-to-end inference on OPT-2.7B (support for larger variants appears to be still under development) for a single batch of 400 tokens, on an Intel(R) Core(TM) i9-7980XE CPU @ 2.60GHz using 18 cores. Table A.11 shows the end-to-end speedups of running sparse models over the dense one, executed in the same engine/environment. (For reference, dense DeepSparse is $1.5\times$ faster than the standard ONNXRuntime.) The achieved speedups are close to the theoretical optimum, which suggests that unstructured sparsity acceleration for LLM inference on CPUs is already quite practical.

Sparsity	40%	50%	60%
Speedup	$1.57\times$	$1.82\times$	$2.16\times$

Table A.11: Speedup over dense version when running sparsified OPT-2.7 models in DeepSparse.

Weight	Q/K/V/Out	FC1	FC2
Dense	2.84ms	10.26ms	10.23ms
2:4 Sparse	1.59ms	6.15ms	6.64ms
Speedup	$1.79\times$	$1.67\times$	$1.54\times$

Table A.12: Runtime and speedup for the different layer shapes occuring in OPT-175B using 2048 tokens.

GPU Speedups. 2:4 sparsity as supported by NVIDIA GPUs of generation Ampere and newer theoretically offers $2\times$ acceleration of matrix multiplications. We now evaluate how big those speedups are in practice for the matmul problem sizes that occur in our specific models of interest. We use NVIDIA's official CUTLASS library (selecting the optimal kernel configuration returned by the corresponding profiler) and compare against the highly optimized dense cuBLAS numbers (also used by PyTorch). We assume a batch-size of 2048 tokens and benchmark the three matrix shapes that occur in OPT-175B; the results are shown in Table A.12. We measure very respectable speedups through 2:4 sparsity between $54 - 79\%$, for

individual layers (end-to-end speedups will likely be slightly lower due to some extra overheads from e.g. attention).

APPENDIX B

Scaling Laws

B.1 Scaling Laws for Sparsely-Connected Foundation Models

B.1.1 Experimental Setup

This section discusses our experimental choices and hyper-parameters, as well as technical details for sparsity-aware AdaFactor and iterative n:m pruning.

Models & datasets. We consider two standard deep learning applications: vision and language. For the former, we focus on Vision Transformers [DBK+21] trained for multi-label image classification on the JFT-4B dataset [DDM+23], consisting of 4 billion images; for the latter, we consider encoder-decoder T5 models [RSR+20b] (improved 1.1 version [Goo23b]) trained for masked-language-modelling on C4 [RSR+20b], consisting of 150+ billion tokens. These choices allow us to study the generality of our laws not just across vision and language but also for different kind of pretraining objectives and variations of Transformer architectures.

Training hyper-parameters. For the most part, we reuse the optimized training hyper-parameters of the original ViT-scaling [ZKHB22] and T5 paper [RSR+20b], respectively. Our only notable change is that we do not factor the second moment of the respective AdaFactor-based [SS18] optimizers (however, we still apply relative learning rate scaling and RMS clipping for T5); this is done since factorized moments for pruning and sparse optimization are not yet very well studied. Further, we train T5 models with batchsize 128 (similar to most ablation studies in the original paper [RSR+20b]) in order to perform sufficiently many optimization steps also for experiments with lower total amounts of training data, which we found important to obtain stable sparse results through model and data scaling.

Model sizes. When it comes to selecting our particular model dimensions, two things must be taken into account: (a) we are particularly interested in the inference-optimal *overtraining regime* [TLI+23] where models get close to their capacity limit, and (b) to produce a model with N non-zeros and sparsity S, we actually need to train a model that is $1/(1-S)$ times larger than a dense model of size N. In combination with the fact that we need to repeat the entire training sweep for multiple sparsity levels, this limits the size of models we can study

while keeping compute requirements feasible. Specifically, we start from the base variants of both ViT and T5 (B/16 and t5-base). Then we generate models of appropriate sizes by scaling only the Transformer's hidden dimension and keeping all other shape parameters constant. This way we can get quite precise size-matches between models and sparsities, facilitating direct comparisons (not all default family models are exactly $2\times$ steps apart and a 50% sparse model would thus not always be directly comparable to the next smallest dense variant); we did not observe any notable performance decrease for dense models using this scaling strategy, at the sizes we study.

Sparsity configurations. We focus primarily on the most fundamental sparsity type, *unstructured* sparsity, but also perform some investigations for the more practical *n:m* pruning pattern [ZMZ+21, PY21] where only n out of m consecutive weights are non-zero. We uniformly sparsify all linear layers in the Transformer backbone, which effectively avoids layer collapse [WZG20], or other edge cases that may otherwise occur in our sweeps, and generally works decently well for Transformer models. On T5 models, we also sparsify the rather large embeddings to the amount necessary for parameter matching a smaller dense version.

Preliminary experiments indicated quickly diminishing returns for very sparse models, which, as discussed previously, are in addition quite expensive to train. Thus, we focus on three medium sparsities: 50%, 75%, 87.5%, corresponding to a $2\times$, $4\times$ and $8\times$ compression rate, respectively. Our implementation is based on the recently proposed Jaxpruner library [LPM+23], which is easy to integrate into the offical ViT [Goo23a] and T5 [Goo23b] codebases.

Pruning strategy. As we intend to execute substantial sweeps across model size, training data and sparsity, it is essential that our pruning method is highly robust and does not require hyper-parameter retuning for each run. A natural candidate is gradual magnitude pruning (GMP) [ZG17], which is well studied and known to be very reliable. At the same time, GMP is usually quite competitive with more complex techniques [SA20, KA22], especially at the medium sparsity levels that we focus on. We also tested a variation of GMP which incorporates diagonal second-order information [KCN+22], but found it to perform almost identically in our setting. Further, we tried AC/DC [PIVA21], STE [LSB+20] and RigL [EGM+20] (which achieve strong results on classic benchmarks) but saw similar or worse performance, while being more sensitive to hyper-parameters as we scale (see also Appendix B.1.3). Thus, we ultimately decided to use GMP.

In terms of specific hyper-parameters, we prune using a cubic schedule starting after 25% of training and ending at 75%, updating every 100 steps. Our sparsification interval was chosen so that pruning begins with a reasonably well trained model and ends with sufficient finetuning of the final sparse structure. However, we performed ablations for frequency/start/end in Appendix B.1.3 and did not find the process to be too sensitive to those hyper-parameters (except for when the pruning interval is really short).

Sweep grids. Table B.1 lists the grid parameters that we sweep over. For ViTs, we consider 7 target models sizes in $2\times$ increments each, while we use 4 targets sizes in increments of $4\times$ for T5. Vision Transformers are trained for 4 different lengths, with the longest corresponding to ≈ 1.8 billion images; language models are trained for 3 different lengths up to ≈ 65 billion tokens. The set of sparsity targets is the same in both cases, corresponding to 2, 4 and $8\times$ compression rate. Overall, the ViT grid was designed to be more extensive whereas the T5 setup was chosen to be more efficient.

Model family	ViT	T5
#Non-zero params	0.66M, 1.33M, ..., 42.4M	1.3M, 5.3M, ..., 85M
Training steps	55K, 110K, 220K, 440K	250K, 500K, 1M
Sparsities	0.0, 0.5, 0.75, 0.875	0.0, 0.5, 0.75, 0.875
Total #runs	112	48

Table B.1: Grid definition for our main scaling sweeps.

B.1.2 Technical Details

Sparsity-aware RMS. AdaFactor [SS18] as employed by T5 [RSR+20b] defines the learning rate relatively, scaling it by the root-mean-square (RMS) of each weight tensor, respectively. We find that this does not interact well with high sparsity, as a tensor with many zeros tends to have a lower RMS, resulting in a smaller learning rate, which is especially problematic as high levels of sparsification require more recovery during the pruning process. To work around this, we always calculate the RMS only over *unpruned* weights, which effectively alleviates this problem. We also apply the same technique to AdaFactor's RMS clipping threshold, but note that this is much less critical than the learning rate scaling.

Iterative n:m pruning. Sparsifying to the n:m pattern is usually done by pruning in one-shot, followed by finetuning [ZMZ+21], or directly training with a dynamic pruning mask via straight-through gradient estimation [LAS+23]. We take a different approach in this work: we gradually remove the smallest weights while ensuring that at least n weights remain in each group of size m. This effectively generalizes the highly robust gradual pruning paradigm to the n:m setting. Not only does *gradual n:m pruning* unify our setups between unstructured and structured sparsity experiments, we also found it to work reliably across scales with the same hyper-parameters, a highly useful property for scaling studies. A simple and efficient implementation of this scheme is shown in Algorithm B.1: the key is to temporarily set the largest n items in each group to ∞, thus ensuring that they are always picked by an unstructured topk selection.

Algorithm B.1: Prune weights $\mathbf{w}$ to sparsity $s \leq 1 - n/m$ where each group of m weights contains at most n zeros.

$I_{\text{nm}} \leftarrow \text{topk-nm}(|\mathbf{w}|, n, m)$
$\mathbf{w}' \leftarrow$ copy of $\mathbf{w}$
$\mathbf{w}'_{I_{\text{nm}}} \leftarrow \infty$
$I_{\text{unstr}} \leftarrow \text{topk-unstr}(|\mathbf{w}'|, 1 - s)$
$\mathbf{w}_{-I_{\text{unstr}}} \leftarrow 0$

B.1.3 Pruning Ablations

We ablate gradual magnitude pruning hyper-parameters by sparsifying ViT-B/16 for 900M images to $S = 0.9375$ sparsity. A high S was chosen in order to amplify differences between parameter settings; we vary the (relative) start and end point of gradual pruning as well as the update frequency of the mask; the pruning schedule is always cubic. As can be seen in

Table B.2, most configurations perform very similar, except when the pruning period is too short overall. We ultimately pick the 25-75/100 setup to ensure that there is sufficient time for training a decent model before starting pruning, as well as for properly finetuning the final sparse version, which we think could be helpful for some points in our main scaling experiment grid.

start	end	freq	accuracy
0.250	0.750	100	45.28
0.250	0.750	50	45.08
0.250	0.750	200	45.13
0.125	0.875	100	45.33
0.475	0.625	100	44.65
0.125	0.625	100	45.13
0.375	0.875	100	45.04

Table B.2: Ablation study of gradual magnitude pruning hyper-parameters.

AC/DC. We also experimented with the AC/DC method [PIVA21], a sparse training approach that was recently shown to yield very strong results on standard (non-foundation model) benchmarks [KKI+23]. We use a sparse and dense cycle length of 20K steps (10K each phase) and apply AC/DC only during the same pruning period as our GMP setup to ensure the same pre- and post-sparsification finetuning. On smaller T5 models, AC/DC works well but yields very similar results to GMP. On larger models, however, AC/DC appears to require some hyper-parameter reconfiguration for higher sparsities. Since per-model hyper-parameter tuning is inconvenient for large scaling sweeps, while initial results also did not suggest clear improvements, we stuck to well established GMP. In general, we note that even for classic benchmarks, major differences between pruning methods tend to appear mostly at very high sparsities [SA20]. Nevertheless, we think that more extensively investigating advanced sparsification approaches in the context of massive pretraining datasets is an interesting topic for future work.

#nnz	0.500		0.750		0.875	
	GMP	AC/DC	GMP	AC/DC	GMP	AC/DC
1.3M	17.11	17.11	15.64	15.96	14.73	14.59
42M	6.11	6.11	5.87	6.05	5.81	6.11

Table B.3: Comparing validation perplexity of T5 models trained for 250K steps using GMP and AC/DC, respectively; we show perplexity as losses at these levels should be compared in log-scale.

B.1.4 Scaling Coefficients

Table B.4 lists the fitted coefficient values for the scaling results presented in the main paper; D is assumed to be the number of images for ViT and the number tokens for T5. The fitting errors are also shown, where we note again that they correspond to the Huber-loss with $\delta = 0.01$ for ViT/JFT and the Huber-loss of $\log L$ with $\delta = 0.001$ for T5/C4, following [HBM+22].

Model	Sparse	a_S	b_S	c_S	b_N	a_D	b_D	c	Error
ViT/JFT	unstr.	2.94e+2	0.821	4.68e+2	0.392	2.37e+8	0.890	4.517	4.93e-4
T5/C4	unstr.	1.68e+1	0.722	4.50e+1	0.245	6.90e+8	0.203	0.651	7.60e-6
T5/C4	n:m	8.64e+1	2.752	5.36e+2	–	–	–	–	2.1e-5

Table B.4: Fitted coefficients of the scaling laws presented in the main paper.

For n:m sparsity, we only refit the sparsity coefficients a_S, b_S and c_S, preserving the other values from the corresponding unstructured results. While the fitting procedure may not be guaranteed to be convex, we find the process to converge to virtually the same values from different random starting points.

B.1.5 Optimal Sparsity Derivations

Sparse costs. We now discuss how to derive the sparse cost factor $c_{\text{mul}}(S)$ given by Equation 5.4 in Section 5.1.4 for our particular pruning setup. For the first 25% of training, we use a dense model that is $1/(1-S)$ times larger, incurring cost $0.25/(1-S)$, while for the last 25% we are training a model with sparsity S of the same cost as the dense reference model, thus contributing a 0.25 term. For the middle 50%, we prune in the cubic $S - S \cdot (1-t)^3$ schedule [ZG17], where $t \in [0, 1]$. The cost spent over this period is given by 1 minus (we care about density rather than sparsity) the integral over the full range of t, which is $(1 - 0.75 \cdot S)$, further multiplied by 0.5 to cover the duration.

Contour lines. To determine the contours of $S_{\text{opt}}(N, C)$, i.e., the regions where a particular sparsity S is optimal, we first query $L(S, N, D)$ with $D_S = D/c_{\text{mul}}(S)$, where $D = C/6N$ is the amount of data corresponding to dense training FLOPs C relative to which c_{mul} is defined, giving:

$$(a_S(1-S)^{b_S} + c_S) \cdot \left(\frac{1}{N}\right)^{b_N} + \left(c_{\text{mul}}(S) \cdot \frac{a_D}{D}\right)^{b_D} + c. \tag{B.1}$$

Next, we optimize for S by differentiating and setting the result equal to 0:

$$-a_s b_s (1-S)^{b_S - 1} \cdot \left(\frac{1}{N}\right)^{b_N} + b_D \left(c_{\text{mul}}(S) \cdot \frac{a_D}{D_S}\right)^{b_D - 1} \cdot \left(c'_{\text{mul}}(S) \cdot \frac{a_D}{D_S}\right) = 0. \tag{B.2}$$

Finally, the form presented in the main paper is constructed by a few algebraic simplifications:

$$b_D \left(c_{\text{mul}}(S) \cdot \frac{a_D}{D_S}\right)^{b_D - 1} \cdot \left(c'_{\text{mul}}(S) \cdot \frac{a_D}{D_S}\right) = a_s b_s (1-S)^{b_S - 1} \cdot N^{-b_N} \tag{B.3}$$

$$b_D \cdot \frac{c'_{\text{mul}}(S)}{c_{\text{mul}}(S)} \cdot \left(c_{\text{mul}}(S) \cdot \frac{a_D}{D_S}\right)^{b_D} = a_s b_s (1-S)^{b_S - 1} \cdot N^{-b_N}. \tag{B.4}$$

Closed form solution. The closed form solution for $S_{\text{opt}}(N, C)$ assuming dense costs $c_{\text{mul}}(S) = 1/(1-S)$ can be found by solving the above contour equation for S:

$$\left(\frac{1}{1-S}\right)^{b_D+1} \cdot b_D \left(\frac{a_D}{D_S}\right)^{b_D} = (1-S)^{b_S-1} \cdot a_s b_s \cdot N^{-b_N} \tag{B.5}$$

$$\left(\frac{1}{1-S}\right)^{b_D+b_S} = \frac{a_S b_S}{b_D} \cdot N^{-b_N} \left(\frac{a_D}{D_S}\right)^{-b_D} \tag{B.6}$$

$$(b_D + b_S) \cdot \log\left(\frac{1}{1-S}\right) = \log\left[\frac{a_S b_S}{b_D} \cdot N^{-b_N} \left(\frac{a_D}{D_S}\right)^{-b_D}\right] \tag{B.7}$$

$$\frac{1}{1-S} = \exp\left\{\log\left[\frac{a_S b_S}{b_D} \cdot N^{-b_N} \left(\frac{a_D}{D_S}\right)^{-b_D}\right] / (b_D + b_S)\right\} \tag{B.8}$$

$$S = 1 - \exp\left\{-\log\left[\frac{a_S b_S}{b_D} \cdot N^{-b_N} \left(\frac{a_D}{D_S}\right)^{-b_D}\right] / (b_D + b_S)\right\}. \tag{B.9}$$

Further algebraic manipulations yield:

$$S = 1 - \exp\left\{\log\left[\frac{b_D}{a_S b_S} \cdot N^{b_N} \left(\frac{a_D}{D_S}\right)^{b_D}\right] / (b_D + b_S)\right\} \tag{B.10}$$

$$S = 1 - \exp\left[\left(\log\frac{b_D}{a_S b_S} + b_N \log N + b_D \log\frac{a_D}{D_S}\right) / (b_D + b_S)\right]. \tag{B.11}$$

$$S = 1 - \exp\left[\left(\log\frac{b_D}{a_S b_S} + b_N \log N\right) / (b_D + b_S)\right] \cdot \left(\frac{a_D}{D_S}\right)^{b_D/(b_D+b_S)}. \tag{B.12}$$

Substituting D_S by its expression in C and guaranteeing that the result is always non-negative (since sparsity < 0 does not exist) brings the formula given in Section 5.1.4.

B.1.6 Impact of Sparsity on Downstream Tasks

As discussed in the main text, the focus of this study, similar to most other works on scaling laws, lies on modeling the *pretraining task validation loss*. For dense models it is known, that this loss is strongly correlated with performance on downstream (e.g. few-shot) applications [HBM+22]. This effect has also been observed for sparse models, for example in the context of transfer learning for ImageNet models [IPKA22]. We now provide additional evidence, for our particular sparse models by evaluating them on a suite of common few-shot tasks.

Concretely, we consider all dense and sparse models from our main sweep and plot few-shot accuracy vs. pretraining loss; the results are shown in Figure B.1. As can be seen, sparse models exhibit very similar behavior to dense ones, both in terms of accuracy achieved for a fixed validation loss as well in terms of overall noise level that is typical for each task. This confirms that sparse models with better pretraining validation loss, like dense models, indeed also perform, on average, better on few-shot tasks.

B.1.7 Practical Sparsity Acceleration

Although we focus primarily on developing a general understanding of sparsity scaling, rather than on accelerating sparse model in practice, there are many works that study the latter. We now summarize some of them.

Network weights that are exactly zero must not be stored and can be safely skipped during computation. This means that, in principle, sparse models require both less memory as well as

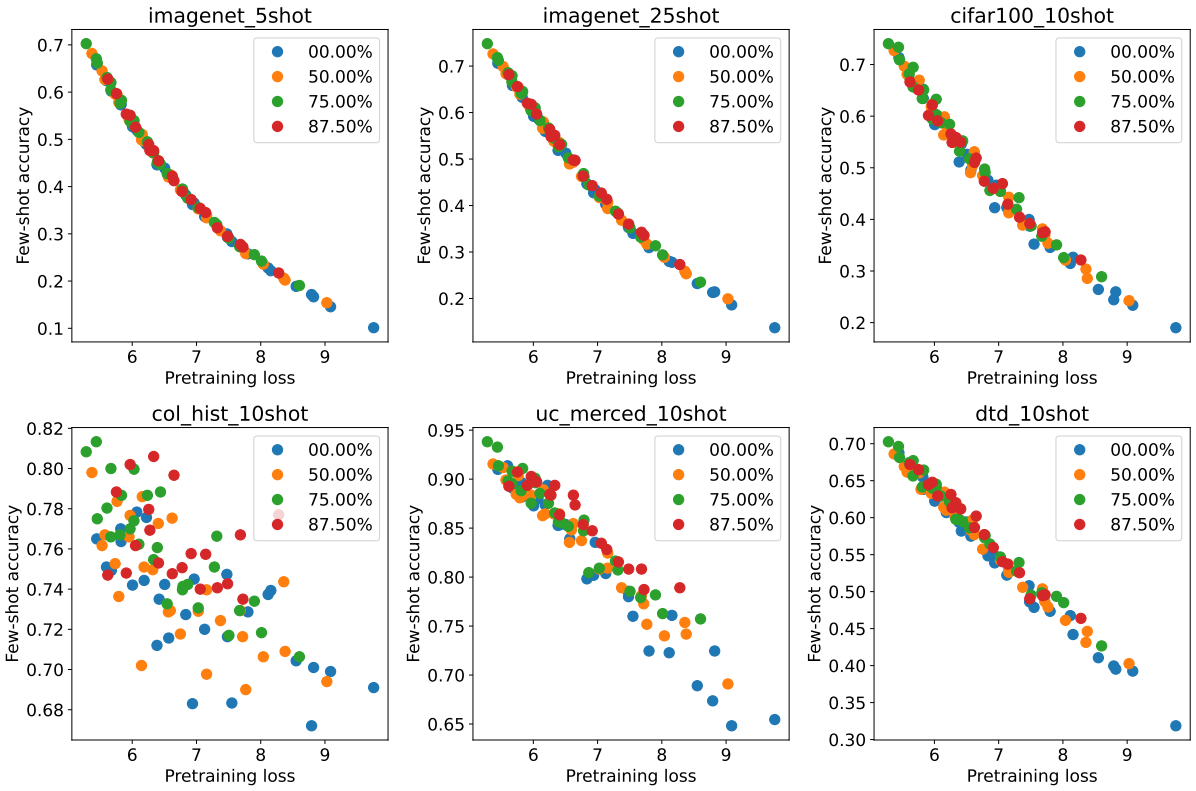


Figure B.1: Few-shot accuracy vs. pretraining loss for dense and sparse ViT/JFT models.

less compute to run than their dense counterparts. For CPU inference, this compute reduction can be effectively translated into end-to-end speedups using sparsity-aware matrix algorithms [EDGS20, KKG+20]. Custom hardware can also achieve strong throughput improvements for sparse models: for example, Thangarasa et al. [TGM+23] report almost $4\times$ speedup on a matrix multiplication with 75% sparse weights. On modern GPUs, effectively utilizing fully unstructured sparsity is challenging due to irregular computation patterns. However, newer generations of NVIDIA GPUs include dedicated hardware support for semi-structured n:m sparsity [PY21], yielding up to $2\times$ speedup over fully dense matrix multiplications.

While effectively utilizing weight sparsity *during* training is still an active of research, there are already some promising results. Emani et al. [EFS+23] report substantial end-to-end training speedups for training large, unstructured sparse, Transformer models on custom hardware. Meanwhile, [NPI+23] develop sparsity-aware backpropagation kernels for efficient CPU training.

Finally, we note a very recent line of work which uses unstructured sparsity [XZL+23, KKF+23] primarily for reducing model size, in order to fit massive Transformer models onto smaller accelerators. Further, for memory bound generative inference, this brings practical as well, by reducing transfer costs between GPU main memory and registers due.