CSE 585 Paper Summaries: November 12, 2024

Aditya Singhvi (singhvi), Alex de la Iglesia (alexdela), Ammar Ahmed (ammarah)

Summary of "AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration"

Problem and Motivation

There are advantages to running models on edge devices rather than communicating with a large scale computing server: reduced latency, offline availability, and privacy. Issues arise with edge computing now that LLMs are getting too large to be deployed on edge devices.
In order to allow edge devices to use these large models, model weights are quantized to reduce size, but reducing precision of weights significantly decreases model performance. AWQ then investigates if some parameters can be left unquantized to maintain most of the model's performance while still receiving the benefit of quantization. A feature's importance can be intuitively thought of as its activation output's magnitude, as this reflects how much influence the parameter has on the resulting output. AWQ proposes identifying the top 1% of weights by this metric, called salient weights, and quantizing all other weights, in order to reduce model size with a smaller drop in performance.

Related Works

Quantization is where parameters are reduced in precision, requiring fewer bits to represent. Because the parameters lose precision, this often results in quantization error, which can result in less accurate results using these imprecise parameters.

Approaches to model quantization to reduce model size have already existed for a while, primarily with two main methods: quantization-aware training (QAT) which uses backpropagation to update quantized weights, and post-training quantization (PTQ) that does not use extra training during quantization. Because of the scale of LLMs, typically post-training quantization is used, decreasing model quality but with no added time cost to quantize the model.

When quantizing, authors need to decide which parameters will be quantized and by how much. Some models only quantize the weights (called W4A16, where W4 is 4 bit weights and A16 is 16 bit activation, not quantized) and others quantize both weights and activation (called W8A8, as each parameter is 8 bit quantized). AWQ uses W4A16 as only the weights will be quantized. Using these quantized models requires specific system support, but have been broadly accepted in various inference systems such as FlexGen, llama.cpp, and exllama, which allow for INT4 quantization, and other systems supporting many other configurations.

Solution Overview

The paper centers its solution on preserving important weights during quantization, called salient weights. Important weights are determined by their activation magnitude, or L2-norm (noticeably not the magnitude of the actual weights values). This is done by running inference on an input dataset (the Pile dataset), observing the activation output, and choosing weights with the highest L2-norm. Once the salient weights have been determined, all other weights can be quantized with little decrease in performance. This results in the salient weights remaining as FP16 (16 bit float), and the quantized weights shrinking to INT3 or INT4 (3 or 4 bit int).

AWQ's quantization results in a mixture of weight data types. Unfortunately, most hardware is unable to perform mixed-precision arithmetic, meaning we cannot multiply quantized INT3 weights with unquantized FP16 weights natively. Storing mixed precision weights is also hardware inefficient (due to padding issues with different size parameters). AWQ's infrastructure has to be modified to permit these operations to work.

AWQ reduces quantization error by scaling parameters before rounding to their quantized version. How much to scale needs to be optimized for, which the use block search to choose from no scaling to very aggressive scaling. This results in quantized weights more similar to their original values. During inference, the quantized weights are then scaled up to FP16, allowing them to be used in conjunction with the unquantized weights.

TinyChat is AWQ's infrastructure for model inference. Noticeably, it must handle variable size data with the W4A16 configuration. The authors observe that accessing weights with memory accesses are the slowest part of inference, and most performance improvements are on reducing memory accesses. The theoretical speedup of using 4 bit integer weights compared to normal 16 bit weights is 4x, but >3x speedup is achieved with TinyChat. To use a mix of quantized parameters and non-quantized, quantized parameters must be dequantized by shifting their values into a FP16 field.

One baseline is GPTQ (Generative Pretrained Transformer Quantized), which is based on OBQ. Its goal is to greedily quantize weights, opting for ones with lower quantization error. However, this is prone to overfitting to its dataset.

AWQ is seen to result in better performing models compared to models quantized with round-to-nearest (RTN) quantization and GPTQ. AWQ's inference results achieve better perplexity (lower perplexity) trained from the same sized models, and has higher performance on captioning datasets in the COCO Captioning dataset.
AWQ also exhibits robustness with respect to the calibration dataset and does not overfit. Its drop in perplexity going from calibration to evaluation dataset is less than that in GPTQ.

Limitations

The paper never explains why the authors decided to use the top 1% of salient weights based on the activation distribution, and similarly for 0.1% of the weights. While 1% provides a much better perplexity than 3% and 0.1% performs similarly to 1%, there are many values within these ranges that should be explored. The paper also does not discuss why they choose INT3/INT4 quantization, or why they don't try less or more quantization. Lastly, this paper does not consider other approaches to reducing model sizes, such as CPU offloading, and does consider solutions that utilize these alternative approaches in their evaluation.

Future Research Directions

Further work could extend AWQ on more or less aggressive quantization. As discussed in the limitations section, not many parameter settings were tried on AWQ in terms of what portion of weights are left unquantized and how aggressive quantization should be to reduce model size. In addition, only weights are quantized-further work could also try to quantize activation parameters as well.

Since this paper does not include non-quantizing approaches to edge LLMs in its evaluation, a survey can be done to compare AWQ and other quantization approaches to other works that aim to reduce LLM size on edge devices.

Summary of Class Discussion

Q: What does 1% mean?
A: Only 1% of the weights are not quantized, the rest are quantized. This shows that most of the model can be quantized with little drop in performance.

Q: Why preserve 1% of weights compared to other percentages?
A: The authors tried portions 0.1%, 1%, 3% without an explanation on why they chose these portions.

Q: For small quantization parameters, would it hurt cache alignment if you quantize with weights that don't fit nicely in cache lines (e.g. INT3)?
A: If a weight is on a cache line boundary, you would need to access both cache lines, but most of the time the weight would lie in a single cache line.

Q: How much of a speedup results from quantization?
A: The algorithm guarantees 4x speedup going from FP16 to INT4 but only 3x speedup is observed.

Q: How does TinyChat minimize DRAM access and could the second paper be integrated?

A: Not entirely sure, we treat TinyChat as a black box for implementing the AWQ algorithm. FlashAttention's goal is offloading weights to flash memory. This could be used during inference in TinyChat, but we are still unsure of TinyChat's implementation if this is possible.

Q: Why can't we quantize the activations? Since AWQ just quantizes the weights?
A: As seen in the graph on slide 8, the actual weights leave a greater memory footprint. Quantizing reduces accuracy, and so the authors avoid quantizing activation because it does not drop memory much compared to its decrease in performance (although there was no test on how much it would drop).

Q: If the salient weight you scale up is close to the maximum, doesn't that increase Δ', which will offset actually scaling up the weight? Also, if you are scaling up 1% of all weights, are you bound to end up scaling a weight that is close to the maximum?
A: This is answered empirically; the paper mentions that as s increases, Δ'/Δ also increases, but not as much as s; for instance, with s=4, Δ'/Δ is about 1.2. The paper does not provide an explanation for why this ends up being the case.

Q: They use perplexity as the metric. Is this a good uniform, or why not accuracy?
A: It is a common metric for evaluation, it is based on negative log-likelihood. As you predict each token, you consider the probability you choose the token you choose. This is similar to the concept of accuracy for other models. In evaluation, they do also consider instruction-tuned models as well, and in other datasets they use other metrics from perplexity.

Q: If you're scaling the salient weights (1%) and you need to do some precomputation, what's the cost of doing that?
A: It's just the cost of doing that multiplication and that depends on your system, and the paper doesn't really talk about the cost of that. Quantization can only be done once and then reused on different devices, so this multiplication does not happen every time the model is used.

Q: Why does going from 0.1% to 3% decrease the perplexity?
A: Perplexity can increase with a larger % based on weight magnitude because using weight magnitude is not a good metric for determining saliency, and so this can result in quantizing the much better weights that you normally would keep salient when using activation magnitude as the metric.

Q: What is the intuition of perplexity? What does the number value mean compared to others?
A: The scale of it is kind of confusing, the only observation we have is that smaller perplexity is better. It is unsure if 2x perplexity means 2x worse model or ½ the accuracy, but larger jumps in perplexity are seen with other configurations, and using activation magnitude for saliency results in perplexity similar to no quantization (around 10 perplexity). The numbers are pretty arbitrary.

Summary of "LLM in a flash: Efficient Large Language Model Inference with Limited Memory"

Problem and Motivation

Running LLMs on personal devices has become difficult due to memory constraints. The current approach for running models on personal devices involves loading the entire model into DRAM, which upper-bounds the maximum model size as the size of the DRAM. For instance, a 7B parameter model (like Llama-7B, the smallest version of the Llama family of models) would require 14GB of memory to load the parameters in half-precision floating point format, which is currently impossible on the vast majority of mobile phones and a sizeable fraction of personal computers. To address this, Alizadeh et al. propose storing model parameters in flash memory, dynamically loading relevant parameters during inference. As flash memory tends to be an order of magnitude larger than DRAM, this enables running inference with large models locally.

Related Works

The authors classify related works addressing reducing the computational and memory requirements of LLM inference fall into two categories: compression and selective execution. Compression techniques (like quantization) aim to reduce the model size, whereas selective execution techniques (like conditional computation) reduce computation. The authors argue that the contribution in this work is orthogonal to these, as it focuses on reducing the time taken by data reads from flash memory. However, the work is more aligned with selective execution; it is similar to DejaVu, a system from Liu et al. that also capitalizes on activation sparsity to load fewer weights. Furthermore, this work is significant in that it foregoes the assumption that the model can fully reside in the DRAM, which differs from the prevailing literature.

Solution Overview

As the bandwidth of flash memory I/O is about 100 times lower than that of DRAM I/O, this paper suggests techniques for (1) reducing the amount of data transfer required, (2) increasing transfer throughput, and (3) managing loaded parameters efficiently in order to make storing parameters in flash memory feasible.

First, to reduce the amount of data transfer between flash memory and DRAM, this solution keeps all model parameters related to the attention mechanism in DRAM permanently. Instead, it focuses on dynamically selectively loading parameters of the feed-forward network (FFN) layers. Recall that an FFN layer consists of an up-projection matrix, a ReLU function, and a down-projection matrix. The ReLU function, which clips all negative activations to 0, naturally creates sparsity for all subsequent layers of the model; previous work has found that the activation sparsity in FFN layers tends to be above 90% in practice. To deal with the parameters up to the ReLU layer (namely, the up-projection matrix), the model uses a low-rank predictor to

predict which intermediate neurons will end up having positive activations (and thus need to be used) based on the outputs of the attention layer. Using this predictor, which is very accurate, leads to minimal performance degradation while saving memory I/O. The other key idea here is keeping a DRAM cache of the parameters corresponding to neurons predicted to be active for the last *k* input tokens. This allows incrementally doing memory I/O for each new token, which experimentally tends to have a lot of active neurons in common with its predecessors. Furthermore, varying the value of *k* can control the memory-compute tradeoff for specific use cases of such systems, as higher values of *k* essentially mean using more available DRAM.

Second, to increase transfer throughput, the authors rely on the simple idea of fewer, longer memory reads being faster than several, shorter reads. This idea is fairly intuitive, as fewer, larger reads lead to the one-time-costs of reading being amortized over several bytes read. The authors note that if the parameters for neuron *i* of the hidden layer in a FFN need to be loaded into DRAM, this implies that row *i* of the up-projection matrix and column *i* of the down-projection matrix need to be loaded. Storing these two arrays of data contiguously in flash memory thus leads to only a single read of size *2k* being performed for each neuron instead of two reads of size *k* each, which is much faster.

Finally, to manage data optimally within DRAM, the authors preallocate all necessary memory to avoid expensive OS operations during inference. The authors discuss a simple, efficient data structure that enables identifying parameters to evict from DRAM in linear time (in terms of the number of neurons). The data structure minimizes copying data, using the insight that the order of neurons in the intermediate output of a FFN does not matter.

The combination of these ideas enable the authors to reduce inference latency for a single token from around 3100 ms in a naive approach that requires loading the entire model into DRAM to about 1000 ms for a model with approximately 7B parameters.

Limitations

One limitation of the work is that they focus only on running a single sequence at a time; applying such ideas to more complicated scenarios such as multi-batch processing is not covered in this paper. The authors also assume that around half of the model can fit in memory, which is an arbitrary assumption. Furthermore, the authors mention that they have not looked thoroughly into power consumption and energy limitations of their method.

Future Research Directions

The authors suggest that a few promising future research directions include analyzing the power consumption of their methods, extending these ideas to multi-batch processing, and applying this method to varying memory sizes. Furthermore, the authors briefly touch on bundling neurons based on co-activation, stating that their experiments did not work as their strategy led to

multiple loadings of highly-active neurons. It seems like there is headroom for improvement in this area, as bundling neurons that are frequently activated together in memory can further improve memory throughput, bringing it closer to the theoretical maximum bandwidth.

Summary of Class Discussion

Q: On Slide 27, when talking about ReLU activation, how generalizable would this be to other activation functions and sparsity levels?
A: The paper actually mentions that the model was fine-tuned to use ReLU instead of other activation functions. This paper seems to rely on the sparsity from ReLU. Only the OPT model used in the paper originally used ReLU; the other models tested in the paper did not originally use ReLU. The authors assume that models will begin using ReLU more, as it leads to sparsity without compromising accuracy. The paper also does not go into detail about the sparsity, and how changes in sparsity can affect this model.

Q: How would this optimization extend to the attention mechanism rather than just FFN?
A: The paper doesn't go into it for attention. The authors mention that this idea of selectively loading weights can be generalized to other things like the key-value caches.

Q: A lot of modern architectures have a distinction between CPU memory and GPU memory (rather than having a shared DRAM). Would this change the work?
A: Instead of loading into one place, you would have to decide which memory you are loading into. This would also have to change the data structures used by the authors. The implementation would definitely have to change.

Q: How can you make sure that there is some stability when using this sliding window approach with incremental loading? What happens if there is a special token that differs significantly from the previous tokens, and causes a lot of memory loading to occur?
A: If a token appears multiple times, but not close enough together in the window, then the parameters would definitely have to be loaded again. If the token appears inside the window beforehand, you would not need to load parameters again.

Q: Why not have a fixed size of stored parameters rather than to just do the last k tokens?
A: Yes, this seems like a strategy that could work. You could also pin some things in memory, if you can figure out which neurons are consistently used.

Q: How much overhead does the low-rank predictor add?
A: The low-rank predictor's computation takes up less than 5% of the time, and does not seem to affect accuracy much as the predictor is very accurate.