Back to Articles

Fine-tuning LLMs to 1.58bit: extreme quantization made easy

Published September 18, 2024

Update on GitHub

medmekk
Mohamed Mekkouri

marcsun13
Marc Sun

lvwerra
Leandro von Werra

pcuenq
Pedro Cuenca

osanseviero
Omar Sanseviero

thomwolf
Thomas Wolf

As Large Language Models (LLMs) grow in size and complexity, finding ways to reduce their computational and energy costs has become a critical challenge. One popular solution is quantization, where the precision of parameters is reduced from the standard 16-bit floating-point (FP16) or 32-bit floating-point (FP32) to lower-bit formats like 8-bit or 4-bit. While this approach significantly cuts down on memory usage and speeds up computation, it often comes at the expense of accuracy. Reducing the precision too much can cause models to lose crucial information, resulting in degraded performance.

BitNet is a special transformers architecture that represents each parameter with only three values: `(-1, 0, 1)`, offering a extreme quantization of just 1.58 (

log2(3)

log

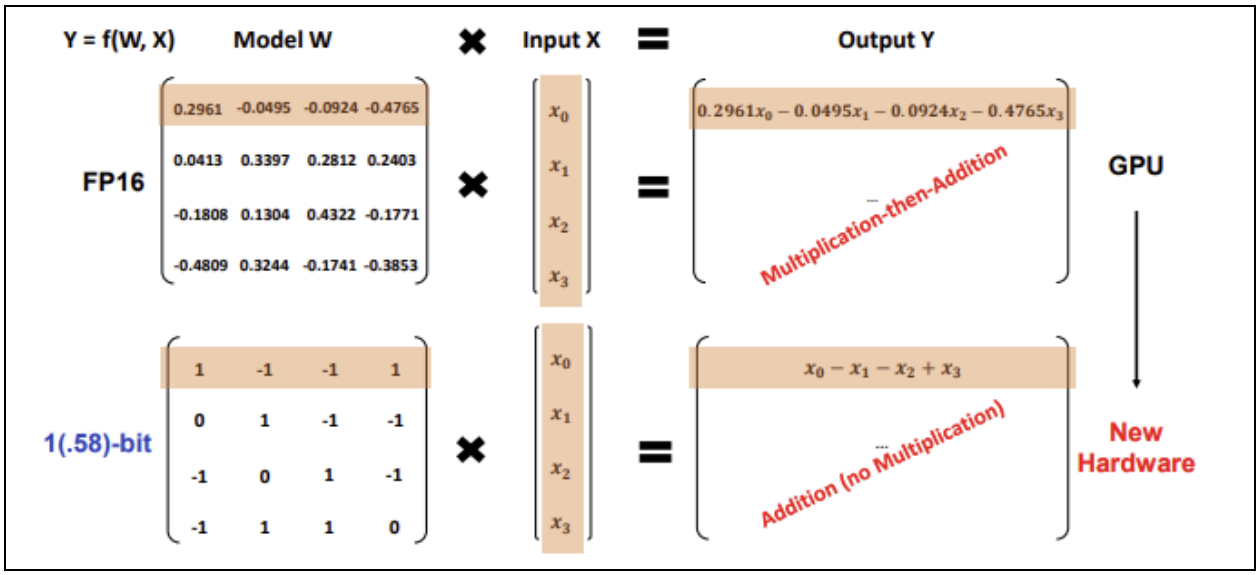
(3)) bits per parameter. However, it requires to train a model from scratch. While the results are impressive, not everybody has the budget to pre-train an LLM. To overcome this limitation, we explored a few tricks that allow fine-tuning an existing model to 1.58 bits! Keep reading to find out how !

Table of Contents

TL;DR

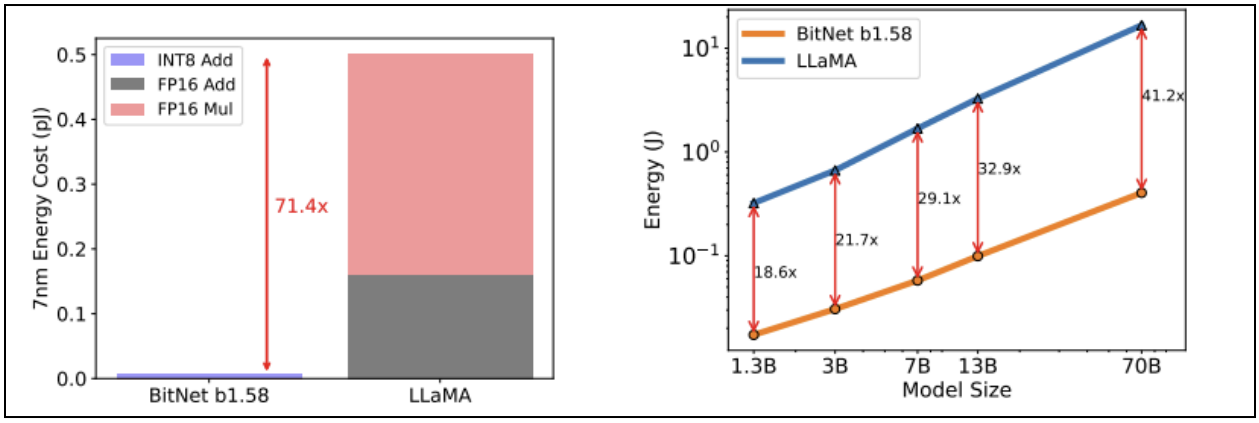
BitNet is an architecture introduced by Microsoft Research that uses extreme quantization, representing each parameter with only three values: -1, 0, and 1. This results in a model that uses just 1.58 bits per parameter, significantly reducing computational and memory requirements.

This architecture uses INT8 addition calculations when performing matrix multiplication, in contrast to LLaMA LLM's FP16 addition and multiplication operations.



The new computation paradigm of BitNet b1.58 (source: BitNet paper https://arxiv.org/abs/2402.17764)

This results in a theoretically reduced energy consumption, with BitNet b1.58 saving 71.4 times the arithmetic operations energy for matrix multiplication compared to the Llama baseline.



Energy consumption of BitNet b1.58 compared to LLama (source: BitNet paper https://arxiv.org/abs/2402.17764)

We have successfully fine-tuned a Llama3 8B model using the BitNet architecture, achieving strong performance on downstream tasks. The 8B models we developed are released under the HF1BitLLM organization. Two of these models were fine-tuned on 10B tokens with different training setup, while the third was fine-tuned on 100B tokens. Notably, our models surpass the Llama 1 7B model in MMLU benchmarks.

How to Use with Transformers

To integrate the BitNet architecture into Transformers, we introduced a new quantization method called "bitnet" (PR). This method involves replacing the standard Linear layers with specialized BitLinear layers that are compatible with the BitNet architecture, with appropriate dynamic quantization of activations, weight unpacking, and matrix multiplication.

Loading and testing the model in Transformers is incredibly straightforward, there are zero changes to the API:

```
model = AutoModelForCausalLM.from_pretrained(
    "HF1BitLLM/Llama3-8B-1.58-100B-tokens",
    device_map="cuda",
    torch_dtype=torch.bfloat16
)
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B-Instruct")

input_text = "Daniel went back to the the the garden. Mary travelled to the kitchen. Sandra journeyed to the kitchen. Sandra went to the hallway. John went to the bedroom. Mary went back to the garden. Where is Mary?\nAnswer:"

input_ids = tokenizer.encode(input_text, return_tensors="pt").cuda()
output = model.generate(input_ids, max_new_tokens=10)
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print(generated_text)
```

With this code, everything is managed seamlessly behind the scenes, so there's no need to worry about additional complexities, you just need to install the latest version of transformers.

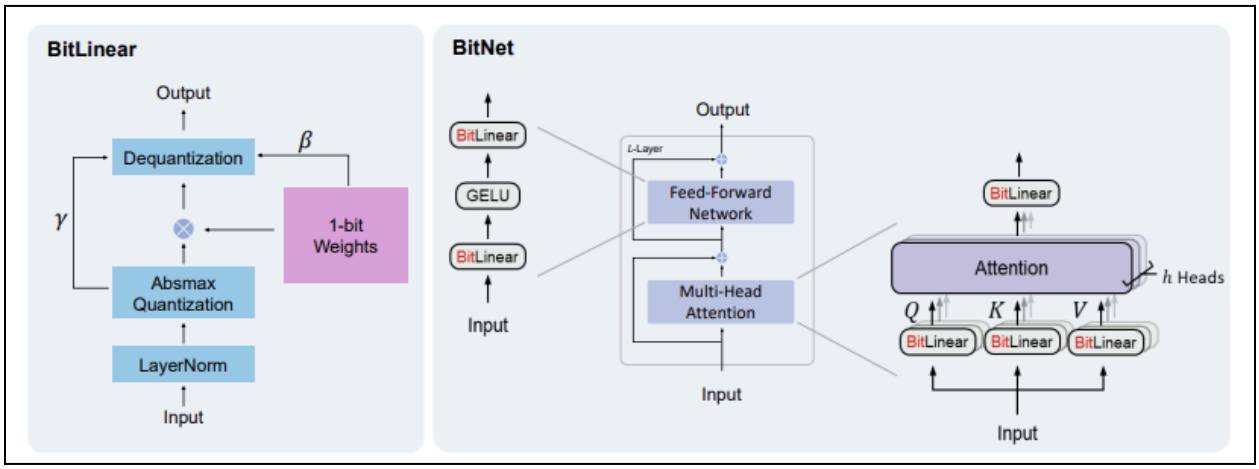
For a quick test of the model, check out this notebook

What is BitNet In More Depth?

BitNet replaces traditional Linear layers in Multi-Head Attention and Feed-Forward Networks with specialized layers called BitLinear that use ternary precision (or even binary, in the initial version). The

BitLinear layers we use in this project quantize the weights using ternary precision (with values of -1, 0, and 1), and we quantize the activations to 8-bit precision. We use a different implementation of BitLinear for training than we do for inference, as we'll see in the next section.

The main obstacle to training in ternary precision is that the weight values are discretized (via the `round()` function) and thus non-differentiable. BitLinear solves this with a nice trick: STE (Straight Through Estimator). The STE allows gradients to flow through the non-differentiable rounding operation by approximating its gradient as 1 (treating `round()` as equivalent to the identity function). Another way to view it is that, instead of stopping the gradient at the rounding step, the STE lets the gradient pass through as if the rounding never occurred, enabling weight updates using standard gradient-based optimization techniques.



The architecture of BitNet with BitLinear layers (source: BitNet paper https://arxiv.org/pdf/2310.11453)

Training

We train in full precision, but quantize the weights into ternary values as we go, using symmetric per tensor quantization. First, we compute the average of the absolute values of the weight matrix and use this as a scale. We then divide the weights by the scale, round the values, constrain them between -1 and 1, and finally rescale them to continue in full precision.

$$scale_w = \frac{1}{nm} \sum_{ij} |W_{ij}|$$

scale

w

=

nm

1

∑

ij

|*W*

ij

|

1

Wq=clamp[−1,1](round(W∗scale))

W

q

=clamp

[−1,1]

(round(*W*∗*scale*))

Wdequantized=Wq∗scalew

W

dequantized

=W

q

∗scale

w

Activations are then quantized to a specified bit-width (8-bit, in our case) using absmax per token quantization (for a comprehensive introduction to quantization methods check out this post). This involves scaling the activations into the range `[-128, 127]` for an 8-bit bit-width. The quantization formula is:

scalex=127|X|max, dim=−1

scale

x

=

|X|

max,dim=−1

Xq=clamp[−128,127](round(X∗scale))

X

q

=clamp

[−128,127]

(round(X∗$scale$))

Xdequantized=Xq∗scalex

X

$dequantized$

=X

q

∗$scale$

x

To make the formulas clearer, here are examples of weight and activation quantization using a 3x3 matrix:

Example 1: Weight Matrix Quantization

Example 2: Activation Matrix Quantization

-
-
-

We apply Layer Normalization (LN) before quantizing the activations to maintain the variance of the output:

$$LN(x) = x - E(x) \sqrt{Var(x) + \epsilon}$$

$$LN(x) = \frac{x - E(x)}{\sqrt{Var(x) + \epsilon}}$$

$$x - E(x)$$

where ϵ is a small number to prevent overflow.

The `round()` function is not differentiable, as mentioned before. We use `detach()` as a trick to implement a differentiable straight-through estimator in the backward pass:

```
# Adapted from https://github.com/microsoft/unilm/blob/master/bitnet/The-Era-of-1-bit-LLMs__Training_Tips_Code_FAQ.pdf
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F

def activation_quant(x):
    scale = 127.0 / x.abs().max(dim=-1, keepdim=True).values.clamp_(min=1e-5)
    y = (x * scale).round().clamp_(-128, 127) / scale
    return y

def weight_quant(w):
    scale = 1.0 / w.abs().mean().clamp_(min=1e-5)
    u = (w * scale).round().clamp_(-1, 1) / scale
    return u

class BitLinear(nn.Linear):
    """
    Only for training
    """
    def forward(self, x):
        w = self.weight
        x_norm = LN(x)

        # A trick for implementing Straight-Through-Estimator (STE) using detach()
        x_quant = x_norm + (activation_quant(x_norm) - x_norm).detach()
        w_quant = w + (weight_quant(w) - w).detach()

        # Perform quantized linear transformation
        y = F.linear(x_quant, w_quant)
        return y
```

Inference

During inference, we simply quantize the weights to ternary values without rescaling. We apply the same approach to activations using 8-bit precision, then perform a matrix multiplication with an efficient kernel, followed by dividing by both the weight and activation scales. This should significantly improve inference speed, particularly with optimized hardware. You can see that the rescaling process differs during training, as matrix multiplications are kept in fp16/bf16/fp32 for proper training.

```
# Adapted from
https://github.com/microsoft/unilm/blob/master/bitnet/The-Era-of-1-bit-LLMs__Training
_Tips_Code_FAQ.pdf
import torch
```

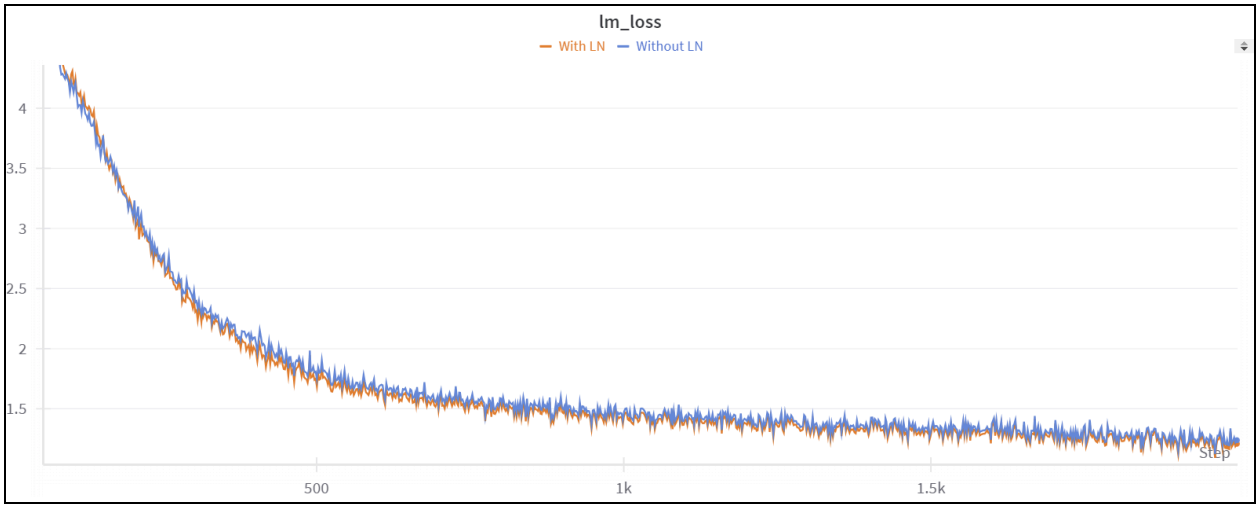
```python
import torch.nn as nn
import torch.nn.functional as F

def activation_quant_inference(x):
    x = LN(x)
    scale = 127.0 / x.abs().max(dim=-1, keepdim=True).values.clamp_(min=1e-5)
    y = (x * scale).round().clamp_(-128, 127)
    return y, scale

class BitLinear(nn.Linear):
    """
    Only for training
    """
    def forward(self, x):
        w = self.weight # weights here are already quantized to (-1, 0, 1)
        w_scale = self.w_scale
        x_quant, x_scale = activation_quant_inference(x)
        y = efficient_kernel(x_quant, w) / w_scale / x_scale
        return y
```

Pre-training Results in 1.58b

Before attempting fine-tuning, we first tried to reproduce the results of the BitNet paper with pre-training. We started with a small dataset, tinystories, and a Llama3 8B model. We confirmed that adding a normalization function, like the paper does, improves performance. For example, after 2000 steps of training, we had a perplexity on the validation set equal to 6.3 without normalization, and 5.9 with normalization. Training was stable in both cases.



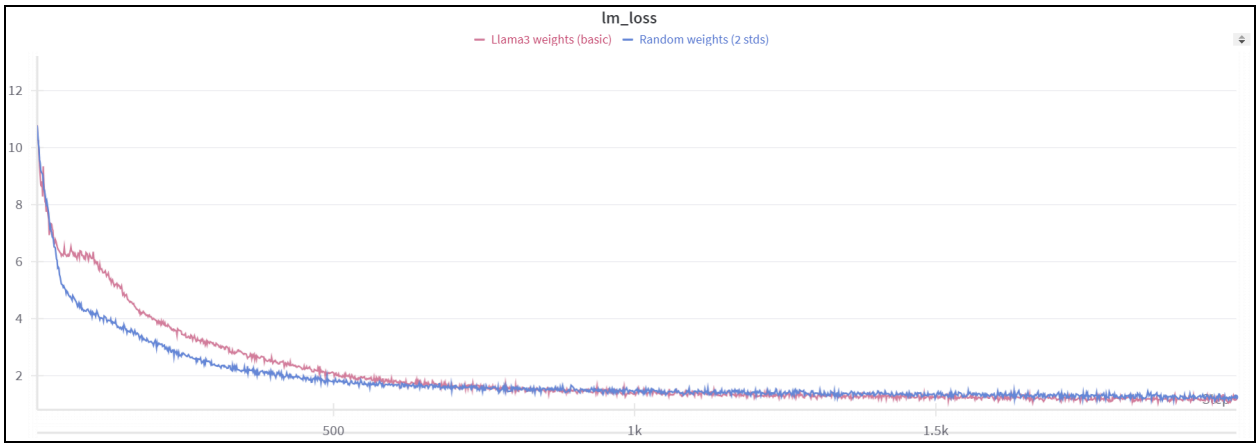
Pre-training plots without (blue) & with (orange) layer normalisation

While this approach looks very interesting for pre-training, only a few institutions can afford doing it at the necessary scale. However, there is already a wide range of strong pretrained models, and it would be extremely useful if they could be converted to 1.58bit after pre-training. Other groups had reported that fine-tuning results were not as strong as those achieved with pre-training, so we set out on an investigation to see if we could make 1.58 fine-tuning work.

Fine-tuning in 1.58bit

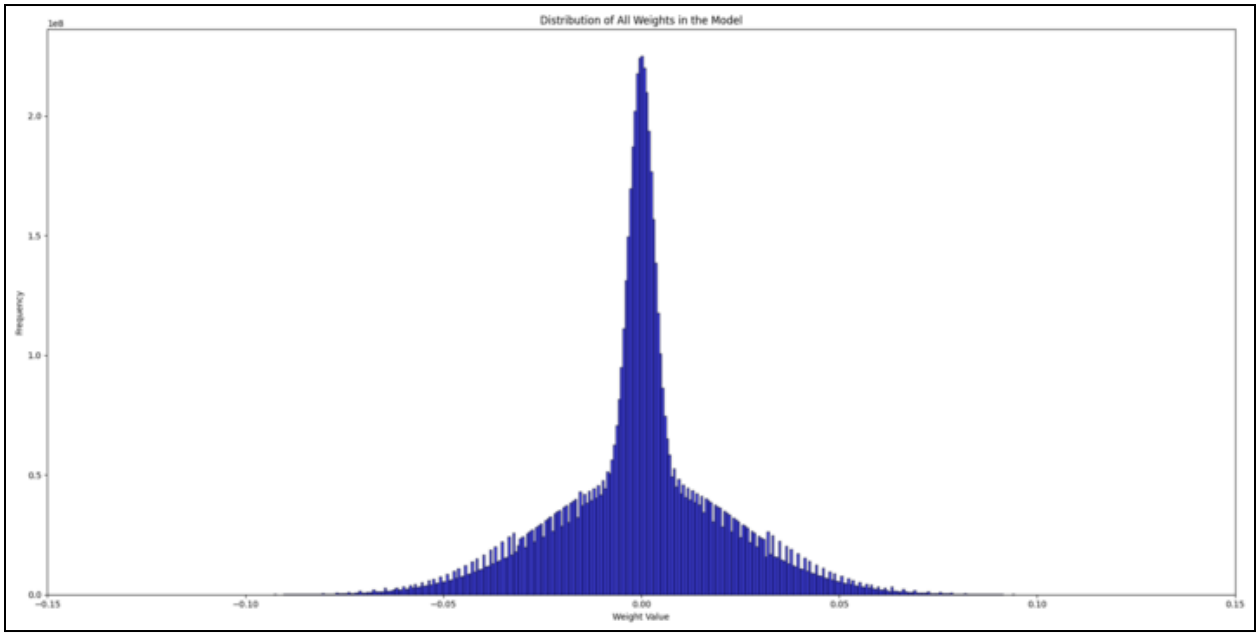
When we began fine-tuning from the pre-trained Llama3 8B weights, the model performed slightly better but not as well as we expected.

Note: All our experiments were conducted using Nanotron. If you're interested in trying 1.58bit pre-training or fine-tuning, you can check out this PR.

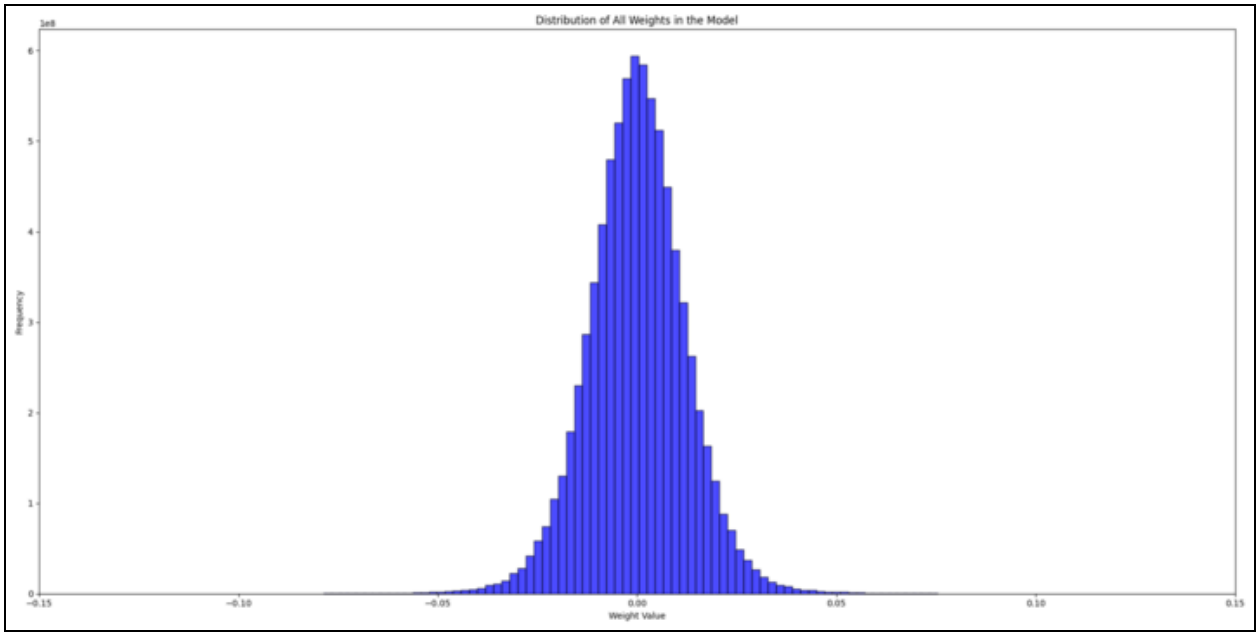


Fine-tuning plot compared to pre-training plot

To understand why, we tried to inspect both the weight distributions of the randomly initialized model and the pre-trained model to identify potential issues.

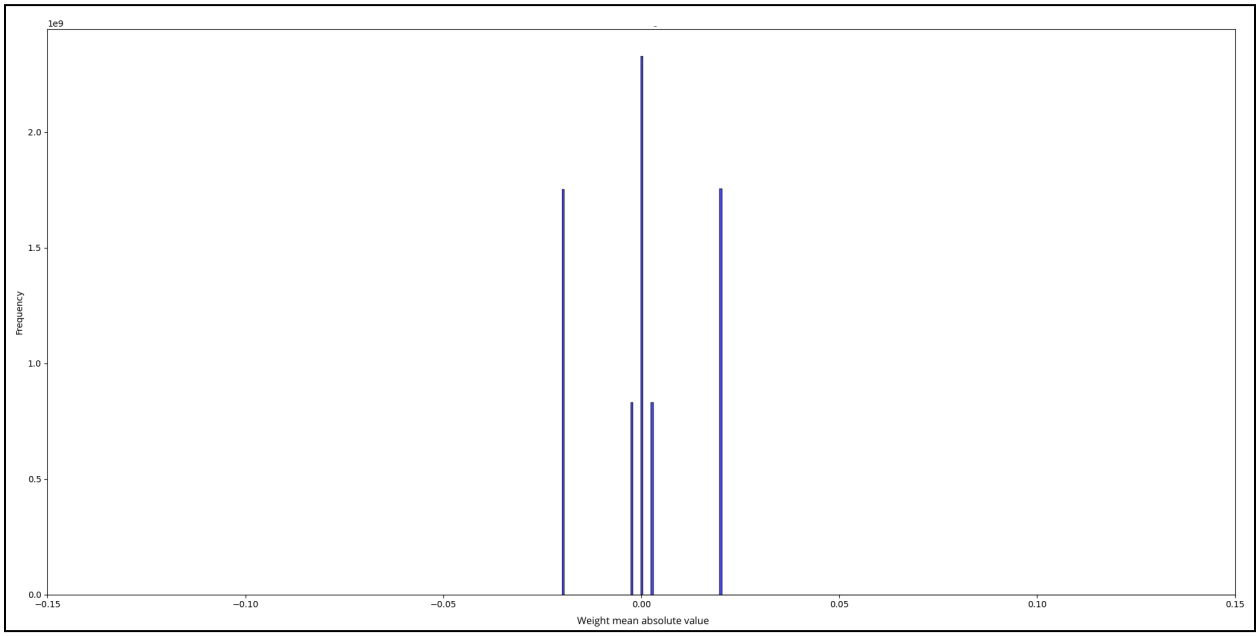


Random weights distribution (2 merged stds)

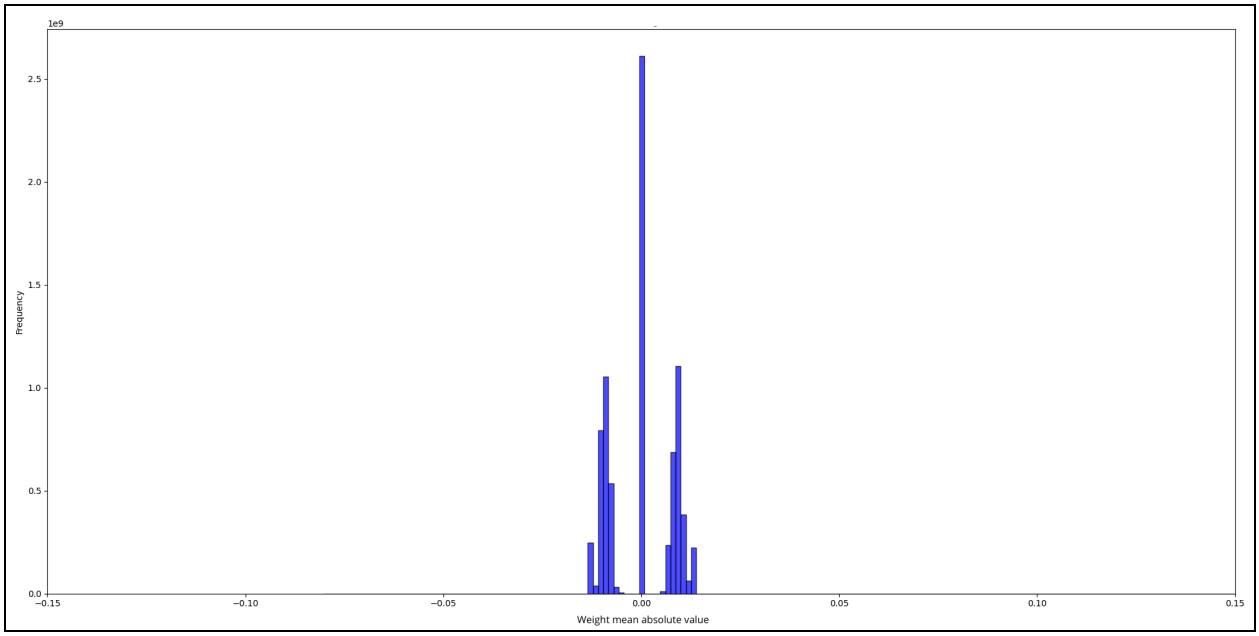


Pre-trained Llama3 weights distribution

And the scale values for the two distributions are, respectively :



Random weights scales distribution



Pre-trained Llama3 weights distribution

The initial random weight distribution is a mix of two normal distributions:

- One with a standard deviation (std) of
- 0.025
- 0.025
- Another with a std of
- $0.0252 \cdot \text{num\_hidden\_layers} = 0.00325$
- $2 \cdot \text{num\_hidden\_layers}$
-
    - 0.025
    -
- $= 0.00325$

This results from using different stds for column linear and row linear weights in `nanotron`. In the quantized version, all matrices have only 2 weight scales (50.25 and 402), which are the inverse of the mean absolute value of the weights for each matrix: `scale = 1.0 / w.abs().mean().clamp_(min=1e-5)`

- For
- scale=50.25
- scale=50.25,
- w.abs().mean()=0.0199
- *w*.*abs*().*mean*()=0.0199, leading to
- std=0.025
- std=0.025 which matches our first standard deviation. The formula used to derive the std is based on the expectation of the half-normal distribution of
- |w|
- |*w*|:

- E(|w|)=std(w)⋅2π
- E(|*w*|)=std(*w*)⋅
  - π
  - 2
  -
-
-
- For
- scale=402
- scale=402,
- w.abs().mean()=0.0025
- *w*.*abs*().*mean*()=0.0025, leading to
- std=0.00325
- std=0.00325

On the other hand, the pretrained weight's distribution looks like a normal distribution with an

std=0.013

std=0.013

Clearly, the pretrained model starts with more information (scales), while the randomly initialized model starts with practically no information and adds to it over time. Our conclusion was that starting with random weights gives the model minimal initial information, enabling a gradual learning process, while

during fine-tuning, the introduction of BitLinear layers overwhelms the model into losing all its prior information.

To improve the fine-tuning results, we tried different techniques. For example, instead of using per-tensor quantization, we tried per-row and per-column quantization to keep more information from the Llama 3 weights. We also tried to change the way the scale is computed: instead of just taking the mean absolute value of the weights as a scale, we take the mean absolute value of the outliers as a scale (an outlier value is a value that exceeds k*mean_absolute_value, where k is a constant we tried to vary in our experiments), but we didn't notice big improvements.

```
def scale_outliers(tensor, threshold_factor=1):
    mean_absolute_value = torch.mean(torch.abs(tensor))
    threshold = threshold_factor * mean_absolute_value
    outliers = tensor[torch.abs(tensor) > threshold]
    mean_outlier_value = torch.mean(torch.abs(outliers))
    return mean_outlier_value

def weight_quant_scaling(w):
    scale = 1.0 / scale_outliers(w).clamp_(min=1e-5)
    quantized_weights = (w * scale).round().clamp_(-1, 1) / scale
    return quantized_weights
```

We observed that both the random weights and the Llama 3 weights resulted in losses starting at approximately the same value of 13. This suggests that the Llama 3 model loses all of its prior information when quantization is introduced. To further investigate how much information the model loses during this process, we experimented with per-group quantization.

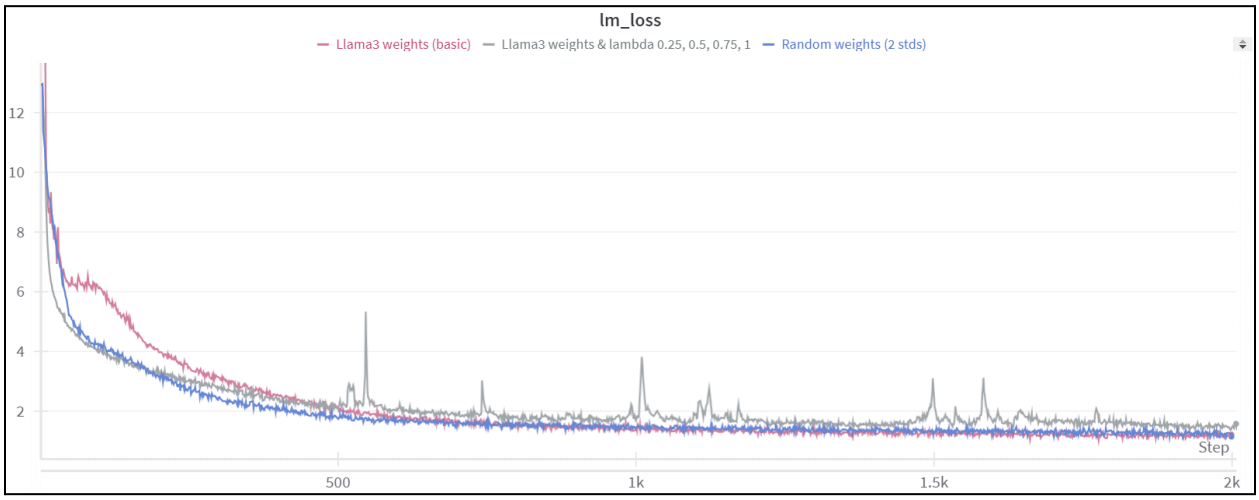
As a sanity check, we first set the group size to 1, which essentially means no quantization. In this scenario, the loss started at 1.45, same as we see during normal fine-tuning. However, when we increased the group size to 2, the loss jumped to around 11. This indicates that even with a minimal group size of 2, the model still loses nearly all of its information.

To address this issue, we considered the possibility of introducing quantization gradually rather than applying it abruptly to the weights and activations for each tensor. To achieve this, we implemented a lambda value to control the process :

```
lambda_ = ?
x_quant = x + lambda_ * (activation_quant(x) - x).detach()
w_quant = w + lambda_ * (weight_quant(w) - w).detach()
```

When `lambda` is set to 0, there is essentially no quantization occurring, while at `lambda=1`, full quantization is applied.

We initially tested some discrete `lambda` values, such as 0.25, 0.5, 0.75, and 1. However, this approach did not lead to any significant improvement in results, mainly because `lambda=0.25` is already high enough for the loss to start very high.



Fine-tuning plot with lambda = 0.25->0.5->0.75->1

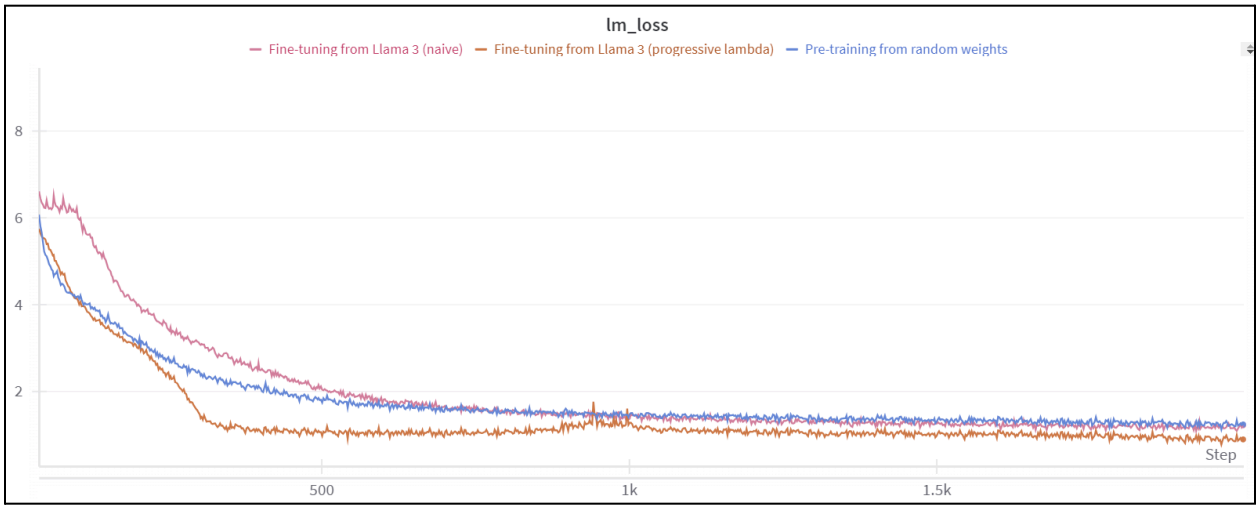
As a result, we decided to experiment with a `lambda` value that adjusts dynamically based on the training step.

```
lambda_ = training_step / total_training_steps
```

Using this dynamic `lambda` value led to better loss convergence, but the perplexity (ppl) results during inference, when `lambda` was set to 1, were still far from satisfactory. We realized this was likely because the model hadn't been trained long enough with `lambda=1`. To address this, we adjusted our `lambda` value to improve the training process.

```
lambda_ = min(2 * training_step / total_training_steps, 1)
```

With this configuration, after 2000 steps we have :



Fine-tuning plot with lambda = min(2*training_step/total_training_steps, 1)

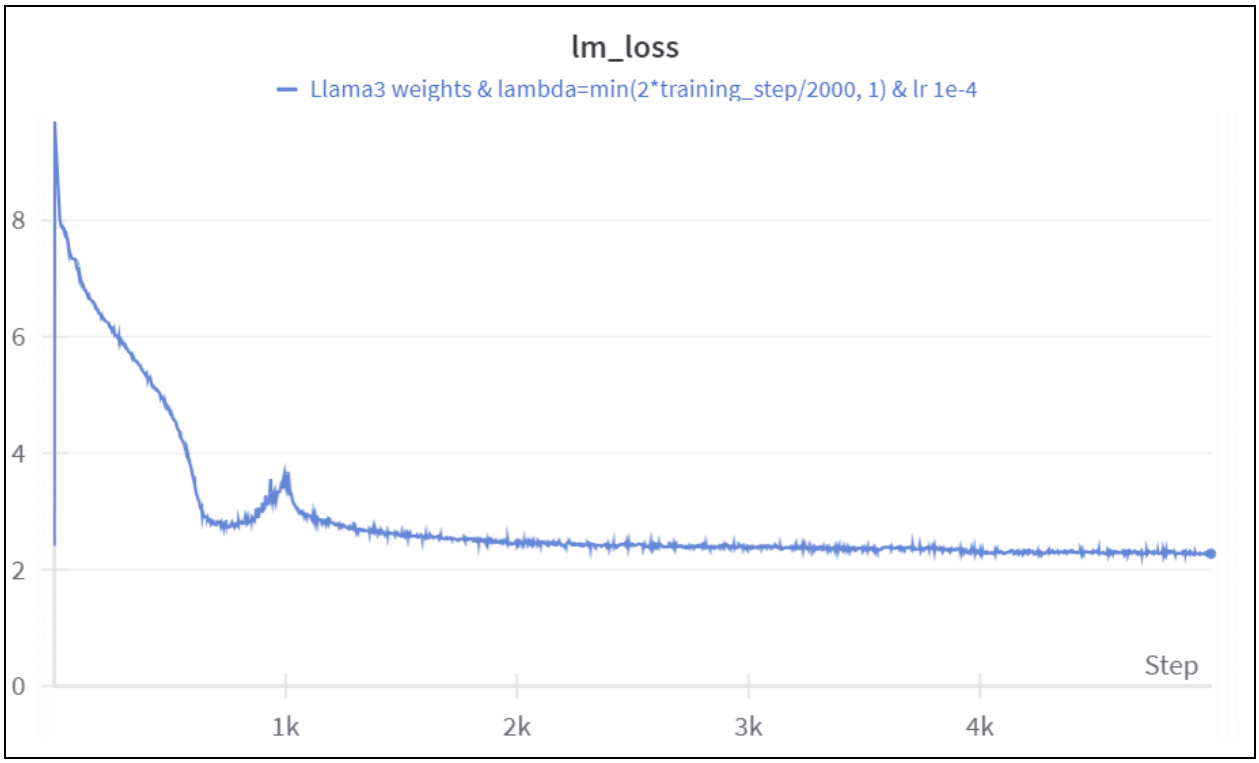
Our fine-tuning method shows better convergence overall. You can observe a slight increase in the loss curve around 1,000 steps, which corresponds to when we begin approaching `lambda=1`, or full quantization. However, immediately after this point, the loss starts to converge again, leading to an improved perplexity of approximately 4.

Despite this progress, when we tested the quantized model on the WikiText dataset (instead of the tinystories one we used for fine-tuning), it showed a very high perplexity. This suggests that fine-tuning the model in low-bit mode on a specific dataset causes it to lose much of its general knowledge. This issue might arise because the minimal representations we aim for with ternary weights can vary significantly from one dataset to another. To address this problem, we scaled our training process to include the larger FineWeb-edu dataset. We maintained a `lambda` value of:

```
lambda_ = min(training_step/1000, 1)
```

We chose this `lambda` value because it seemed to be a good starting point for warming up the model. We then trained the model using a learning rate of 1e-4 for 5,000 steps on the FineWeb-edu dataset. The training involved a batch size (BS) of 2 million, totaling 10 billion tokens.

Finding the right learning rate and the right decay was challenging; it seems to be a crucial factor in the model's performance.



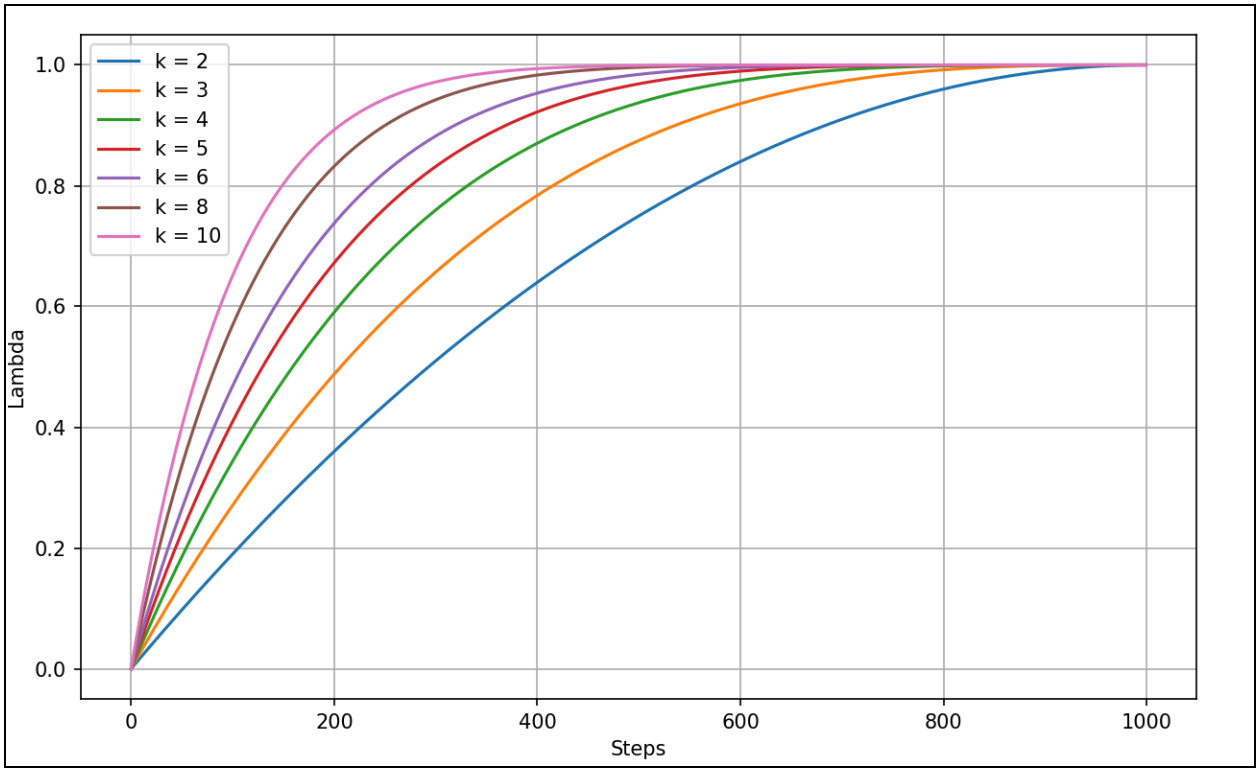
Fine-tuning plot with warmup quantization on Fineweb-edu

After the fine-tuning process on Fineweb-Edu, the perplexity on the WikiText dataset reached 12.2, which is quite impressive given that we only used 10 billion tokens. The other evaluation metrics also show strong performance considering the limited amount of data (see results).

We also tried to smooth out the sharp increase when lambda approaches 1. To do this, we considered using lambda schedulers that grow exponentially at first, then level off as they get closer to 1.

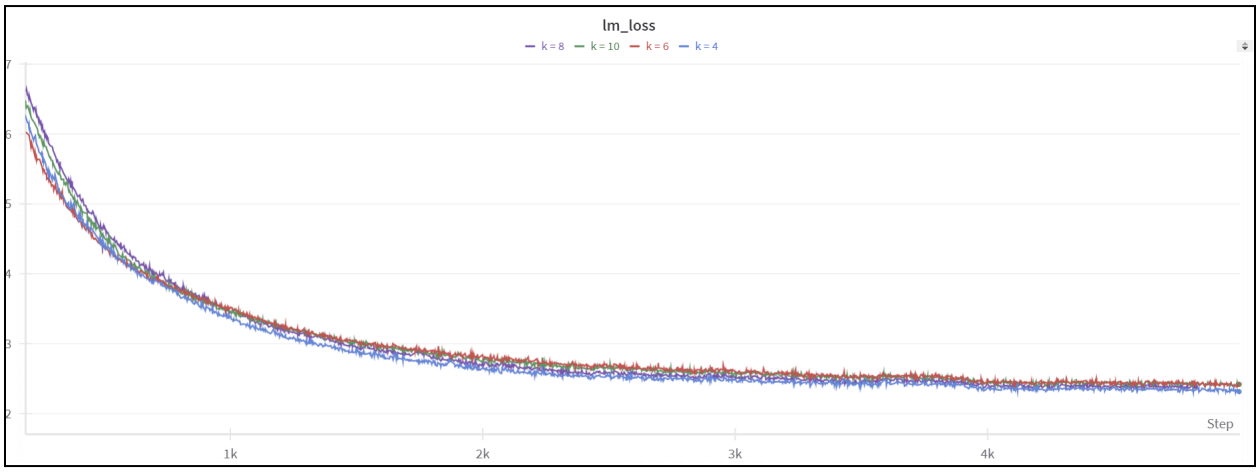
```
def scheduler(step, total_steps, k):
    normalized_step = step / total_steps
    return 1 - (1 - normalized_step)**k
```

for different k values, with a number of total warmup steps of 1, we have plots like the following :



Exponential scheduler for different k values

We ran 4 experiments using the best-performing learning rate of 1e-4, testing values of k in [4, 6, 8, 10].



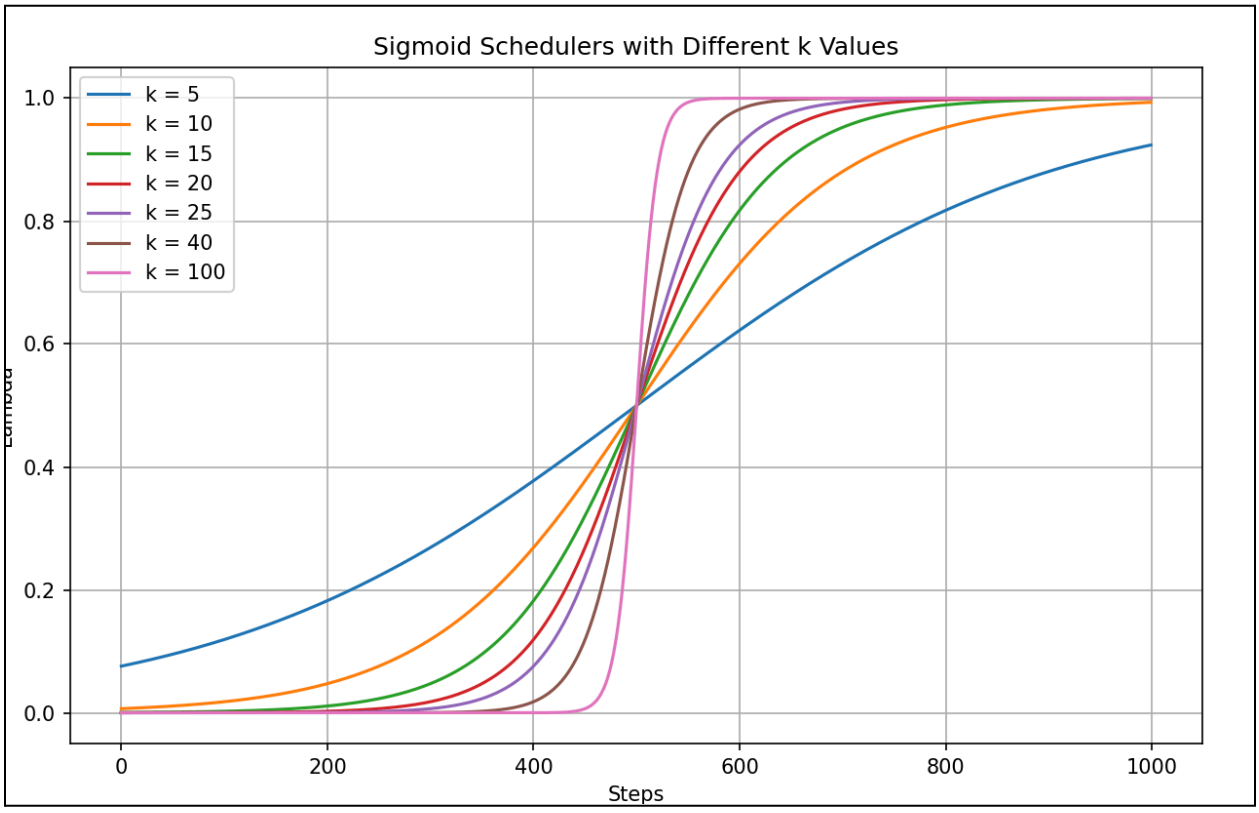
Fine-tuning plots with exponential scheduler

The smoothing worked well, as there's no spike like with the linear scheduler. However, the perplexity isn't great, staying around ~15, and the performance on downstream tasks is not better.

We also noticed the spike at the beginning, which the model struggled to recover from. With lambda = 0, there's essentially no quantization, so the loss starts low, around ~2. But right after the first step, there's a spike, similar to what happened with the linear scheduler (as seen in the blue plot above). So, we tried a different scheduler—a sigmoid one—that starts slowly, rises sharply to 1, and then levels off as it approaches 1.

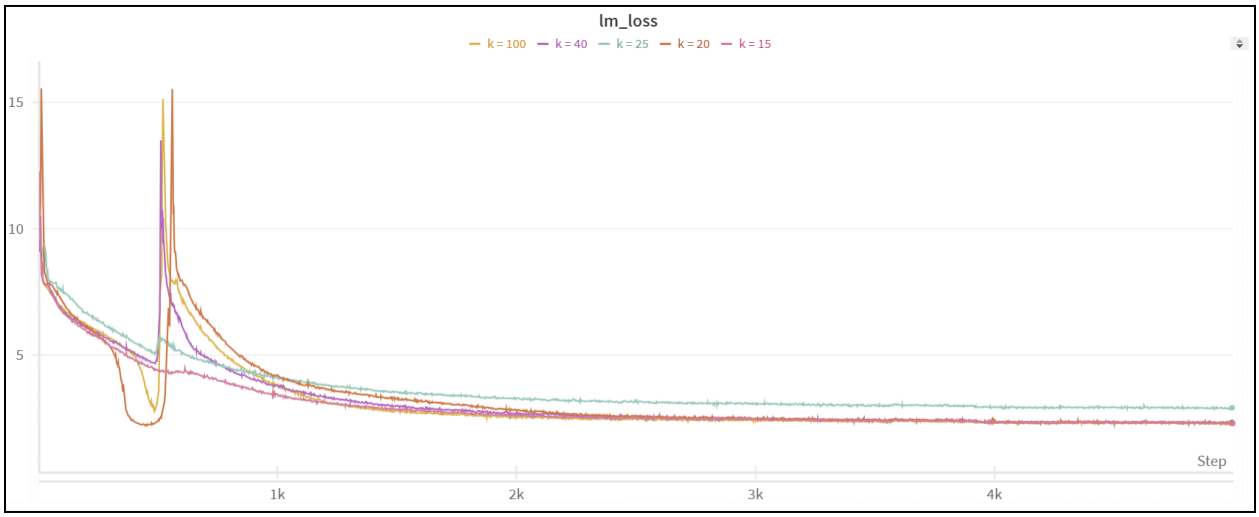
```
def sigmoid_scheduler(step, total_steps, k):
    # Sigmoid-like curve: slow start, fast middle, slow end
    normalized_step = step / total_steps
    return 1 / (1 + np.exp(-k * (normalized_step - 0.5)))
```

For different k values we have the following curves :



Sigmoid scheduler for different k values

We ran 5 experiments this time with k in [15, 20, 25, 40, 100] :



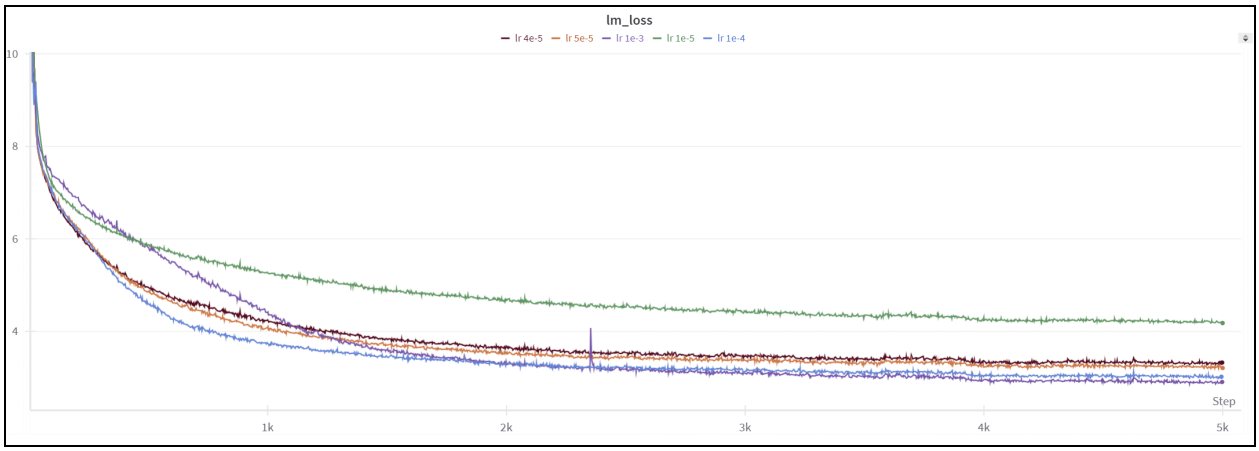
Fine-tuning plots with sigmoid scheduler

The sharp increase in lambda caused instability around the 500th step and didn't fix the first divergence issue. However, for

k=100

k=100, we observed some improvement in downstream tasks (see the results table), although perplexity remained around ~13.5. Despite this, it didn't show a clear performance boost over a linear scheduler.

Additionally, we experimented with training models from scratch using random weights and various learning rates. This allowed us to compare the effectiveness of our fine-tuning approach against traditional pre-training methods.



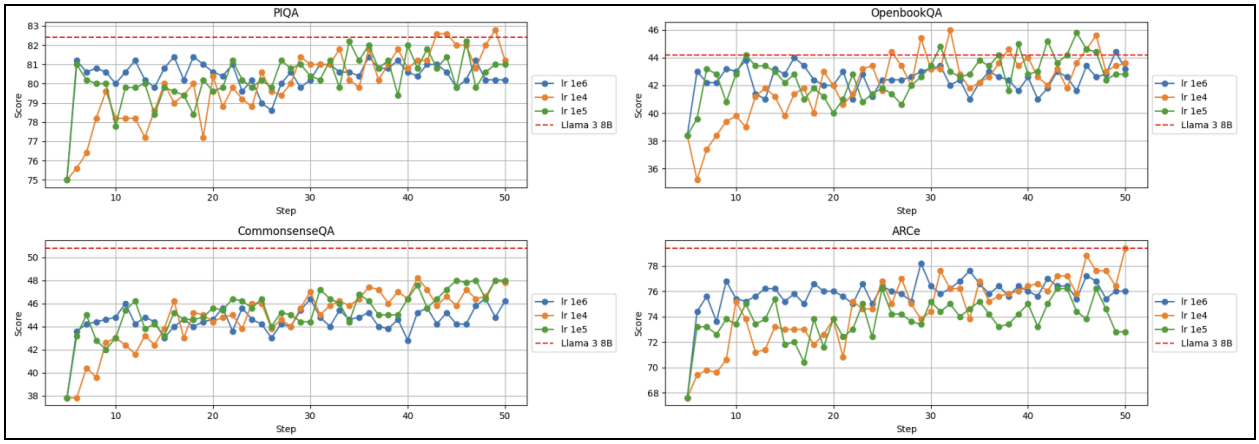
Different Pre-training plots with different learning rates

None of the models trained from random weights performed better than our fine-tuned model. The best perplexity we achieved with those models was 26, which falls short compared to the results from our fine-tuning approach.

Scaling to 100B Tokens !

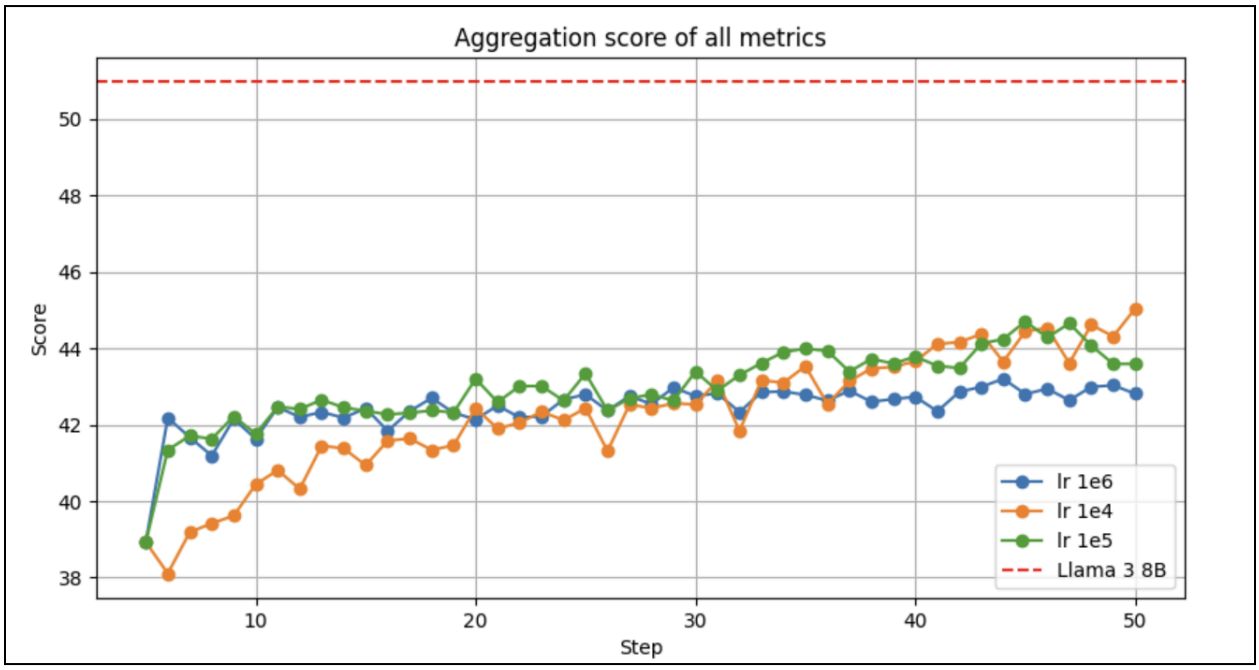
We scaled our experiments to 100 billion tokens to see if we could match the performance of Llama 3 8B. We conducted longer training runs, starting from our best-performing checkpoint from the shorter runs with the linear scheduler, and continued fine-tuning for 45,000 steps. We experimented with different learning rates, and while the model performed closely to the Llama 3 model in some metrics, on average, it still lagged behind.

Here are some examples of the metrics we evaluated at various checkpoints during the training :



Metrics evaluations during the training for different lrs

and the average score looks like :

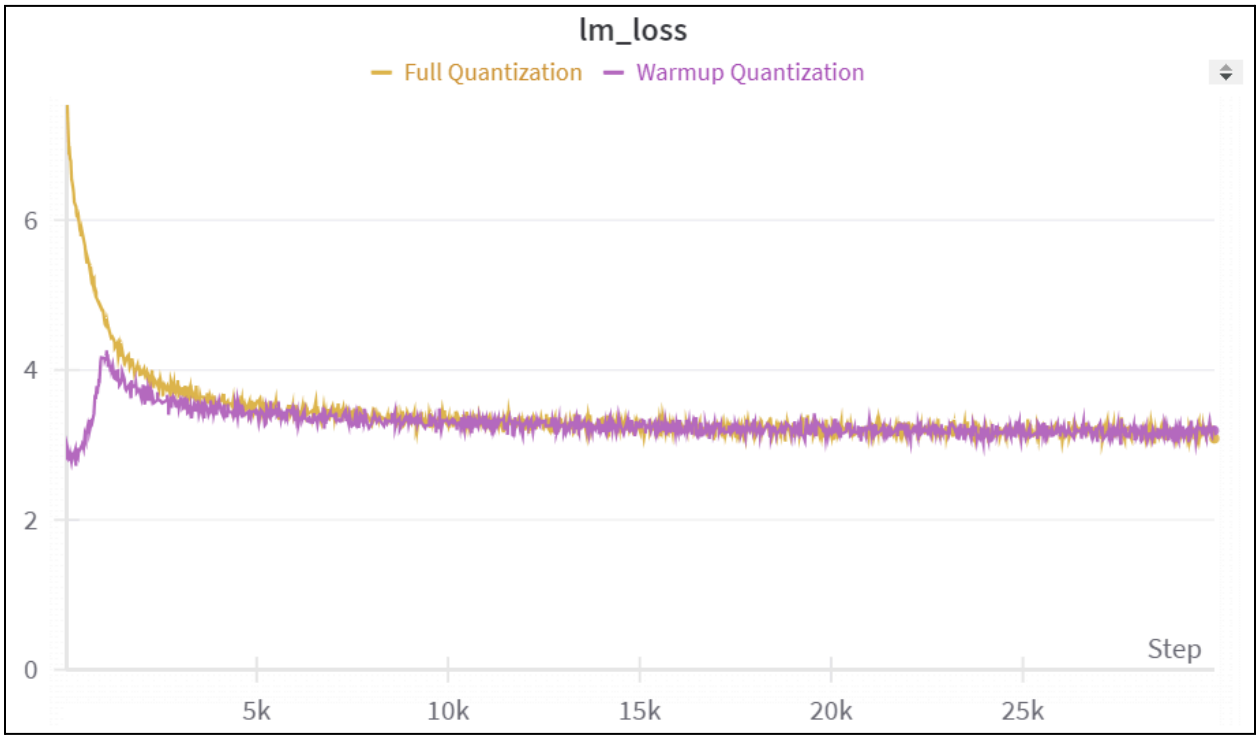


Average evaluation during the training for different lrs

Experiments on Smaller Models

In our initial experiments with smaller models like SmolLM, we observed that the warmup quantization technique didn't yield as much improvement as it did with larger models. This suggests that the effectiveness of warmup quantization could be more closely related to model size and complexity.

For example, here are the loss curves for the SmolLM 135M model, comparing warmup quantization with full quantization from the start. Interestingly, the curves closely align, and the resulting perplexities aren't significantly different.



Smoll LLm fine-tuning experiment with & without warmup quantization

Results & Comparison

BitNet is effective in delivering strong performance compared to baseline methods, especially at lower bit levels. According to the paper, BitNet achieves scores that are on par with 8-bit models but with significantly lower inference costs. In the case of 4-bit models, methods that only quantize weights outperform those that quantize both weights and activations, as activations are harder to quantify. However, BitNet, which uses 1.58-bit weights, surpasses both weight-only and weight-and-activation quantization methods.

The table below presents the results for various metrics after the 10B fine-tuning process of Llama3 8B. These results are compared against those from other model architectures to provide a comprehensive overview of performance (All evaluations were conducted using Lighteval on the Nanotron format model)

Model	PPL	ARCe	ARCc	HS	OQ	PQ	WGe	SQ	MMLU
Bitnet 7B	-	-	-	38.9	-	-	51.4	-	-
FBI 7B	9.1	53.0	29.9	57.7	36.8	72.6	58.9	-	-
Llama 7B	5.7	72.9	44.9	76.2	44.4	79.2	69.9	-	35.1
Llama2 7B	5.5	74.6	46.2	76	44.2	79.1	69.1	-	45.3
Llama3 8B	8.4	79.4	56.8	76	44.2	82.4	67.4	47.8	49.3
Llama3-8B-1.58-Linear-10B-tokens	12.2	67.6	39.4	63.4	38.4	75.0	56.6	40.4	37.05
Llama3-8B-1.58-Sigmoid-k100-10B-tokens	13.5	74.0	41.6	66.0	38.0	78.4	55.6	40.6	38.0

Metrics comparison with Llama models : Linear means Linear lambda scheduler, and Sigmoid means Sigmoid lambda scheduler (in our case k = 100)

After fine-tuning on just 10 billion tokens using ternary weights, the model demonstrates impressive performance, especially when compared to other models that underwent much more extensive training. For instance, it outperforms the Bitnet 7B model, which was trained on a significantly larger dataset of 100 billion tokens. Additionally, it performs better than the FBI LLM (Fully Binarized LLM), a model that was distilled on an even more massive 1.26 trillion tokens. This highlights the model's efficiency and effectiveness despite the relatively smaller scale of its fine-tuning process.

For the 100B tokens experiments, the best performing checkpoint we had is the following :

Model	PPL	ARCe	ARCc	HS	OQ	PQ	WGe	SQ	MMLU
Llama 7B	5.7	72.9	44.9	76.2	44.4	79.2	69.9	-	35.1
Llama2 7B	5.5	74.6	46.2	76	44.2	79.1	69.1	-	45.3
Llama3 8B	8.4	79.4	56.8	76	44.2	82.4	67.4	47.8	49.3
Llama3-8B-1.58-100B-tokens	11.7	72.8	45.4	70.6	42.8	81.0	58.0	43.4	41.6

Metrics comparison with Llama models for the model trained on 100B tokens

To replicate these results, you can check out this PR to convert models to nanotron format, unpack the weights (check the function unpack_weights), and use lighteval

Note that even though the models are fine-tuned from an Instruct-tuned model, they still need to be fine-tuned using an Instruct dataset as well. These can be considered base models.

Custom Kernels & Benchmarks

To benefit from the BitNet low-precision weights, we pack them into an `int8` tensor (this makes the number of parameters go from 8B to 2.8B!). During inference, these weights must be unpacked before performing matrix multiplication. We implemented custom kernels in Cuda and Triton to handle the on-the-fly unpacking during the matrix multiplication process. For the matrix multiplication itself, we employed the cached tiled matrix multiplication technique. To fully grasp this approach, let's first review some Cuda programming fundamentals.

Basic GPU Concepts: Threads, Blocks, and Shared Memory

Before diving into cached tiled matrix multiplication, it's important to understand some basic GPU concepts:

- Threads and Blocks: GPUs execute thousands of threads simultaneously. These threads are grouped into blocks, and each block runs independently. The grid is made up of these blocks, and it represents the entire problem space. For example, in matrix multiplication, each thread might be responsible for computing a single element of the output matrix.
- Shared Memory: Each block has access to a limited amount of shared memory, which is much faster than global memory (the main memory on the GPU). However, shared memory is limited in size and shared among all threads within a block. Using shared memory effectively is key to improving performance in GPU programs.

Challenges in Matrix Multiplication

A simple implementation of matrix multiplication on a GPU might involve each thread computing a single element of the result matrix by directly reading the necessary elements from global memory. However, this approach can be inefficient for the following reasons:

- Memory Bandwidth: Accessing global memory is relatively slow compared to the speed at which the GPU cores can perform computations. If each thread reads matrix elements directly from global memory, the memory access times can become a bottleneck.
- Redundant Data Access: In matrix multiplication, many elements of the input matrices are used multiple times. If each thread fetches the required data from global memory independently, the same data might be loaded into the GPU multiple times, leading to inefficiency. For example, if each thread is used to compute a single element in the output matrix, the thread responsible for calculating the element at position (i, j) will need to load the i-th row of matrix A and the j-th column of matrix B from global memory. However, other threads, such as the one computing the element at position (i+1, j), cannot reuse this data and will have to reload the same j-th column from global memory again.

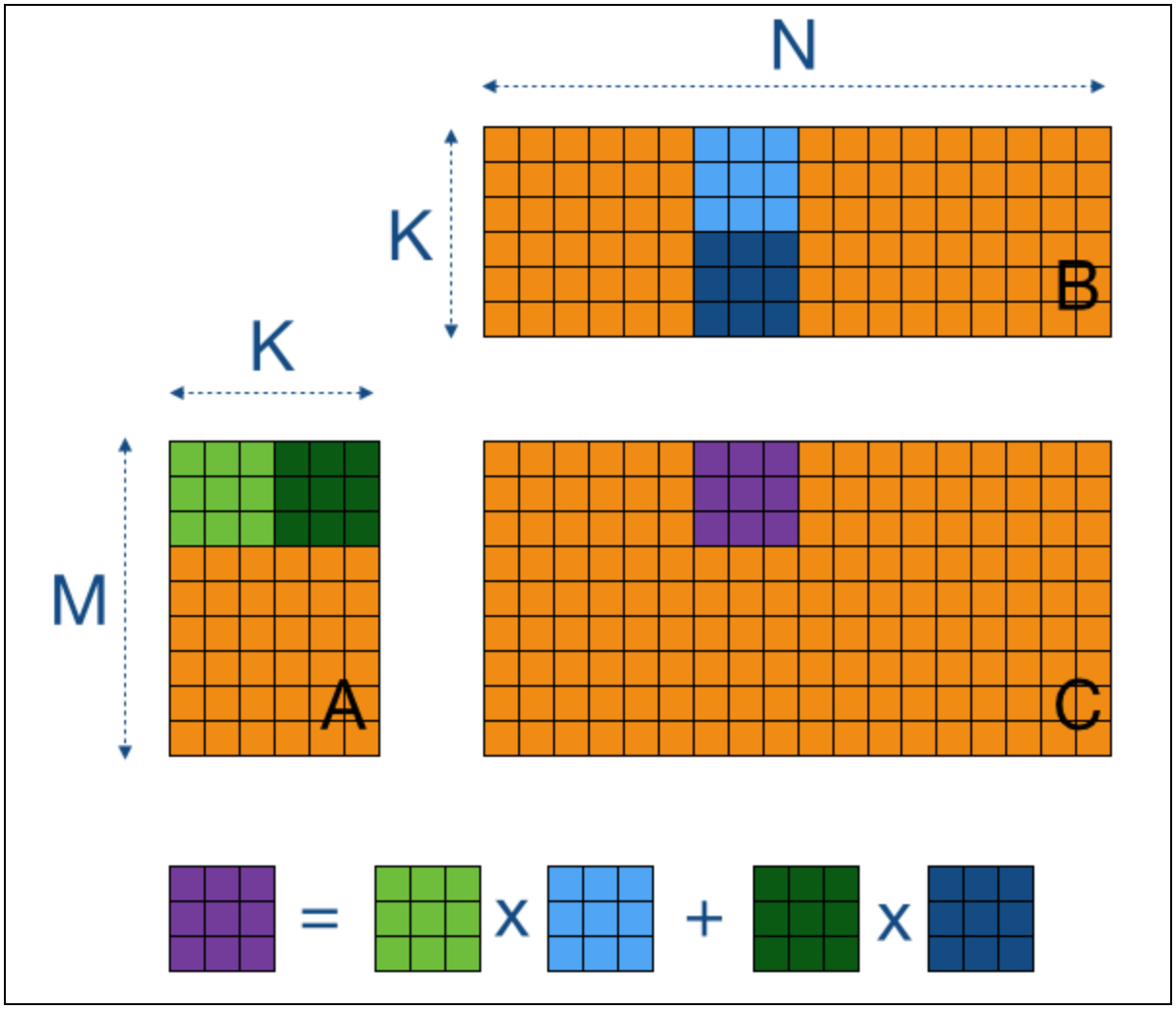
The Idea of Tiling

Tiling is a technique used to address these challenges, and it was mainly used in FlashAttention to improve the kernel's efficiency. The basic idea is to divide the matrices into smaller sub-matrices, called tiles, which can fit into the shared memory of the GPU. Instead of computing the entire output matrix in one go, the computation is broken down into smaller pieces that are processed tile by tile.

In the context of matrix multiplication, this means dividing matrices A and B into blocks (tiles), loading these tiles into shared memory, and then performing the multiplication on these smaller blocks. This approach allows the threads to reuse data stored in the fast shared memory, reducing the need to access global memory repeatedly.

Here’s how it works:

- Loading Tiles into Shared Memory: Each block of threads cooperatively loads a tile of matrix A and a corresponding tile of matrix B from global memory into shared memory. This operation is done once per tile, and then the tile is reused multiple times by the threads in the block.
- Computing Partial Products: Once the tiles are loaded into shared memory, each thread computes a partial product. Since all threads in a block are working on the same tiles in shared memory, they can efficiently reuse the data without additional global memory accesses.
- Accumulating Results: After computing the partial products for one tile, the threads load the next tiles from matrices A and B into shared memory and repeat the process. The results are accumulated in a register (or local memory), and once all tiles have been processed, the final value for the output matrix element is written back to global memory.



Tiled Matrix multiplication illustration (source https://cnugteren.github.io/tutorial/pages/page4.html)

Practical Considerations

When implementing cached tiled matrix multiplication, several factors are considered:

- Tile Size: The size of the tiles should be chosen to balance the trade-off between the amount of data that can fit into shared memory and the number of global memory accesses.
- Memory Coalescing: the global memory accesses are coalesced, which means that adjacent threads access adjacent memory locations.
- Occupancy: The number of threads per block and the number of blocks in the grid should be chosen to ensure high occupancy, which means having as many active warps (a warp is a set of 32 threads) as possible on the GPU to hide memory latency.

Triton Kernel

Here is the kernel in triton we benchmarked :

```
@triton.autotune(
    configs=get_cuda_autotune_config(),
    key=['M', 'N', 'K'],
)
@triton.jit
def matmul_kernel(
        a_ptr, b_ptr, c_ptr,
        M, N, K,
        stride_am, stride_ak,
        stride_bk, stride_bn,
        stride_cm, stride_cn,
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K:
tl.constexpr,
        GROUP_SIZE_M: tl.constexpr,
):

    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m

    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)

    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.int32)

    for i in range(4) :
        b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
        for j in range(0, tl.cdiv(K // 4, BLOCK_SIZE_K) ):
            k = i * tl.cdiv(K // 4, BLOCK_SIZE_K) + j

            # BLOCK_SIZE_K must be a divisor of K / 4
            a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0)
```

```
            b_uint8 = tl.load(b_ptrs, mask=offs_k[:, None] < K // 4 - j *
BLOCK_SIZE_K, other=0)
            mask = 3<<(2*i)
            b = ((b_uint8 & mask) >> (2*i))

            # We accumulate the tiles along the K dimension.
            tensor_full = tl.full((1,), 1, dtype=tl.int8)

            accumulator += tl.dot(a, (b.to(tl.int8) - tensor_full),
out_dtype=tl.int32)

            a_ptrs += BLOCK_SIZE_K * stride_ak
            b_ptrs += BLOCK_SIZE_K * stride_bk

    c = accumulator

    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)


def matmul(a, b):
    assert a.shape[1] == b.shape[0] * 4, "Incompatible dimensions, the weight matrix
need to be packed"
    assert a.is_contiguous(), "Matrix A must be contiguous"
    M, K = a.shape
    _, N = b.shape
    c = torch.empty((M, N), device=a.device, dtype=torch.float16)
    grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']) * triton.cdiv(N,
META['BLOCK_SIZE_N']), )
    matmul_kernel[grid](
        a, b, c,
        M, N, K,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1),
    )
    return c
```

Code Breakdown

1. Determining Tile Positions

The kernel first determines which tile (block) of the output matrix each thread block is responsible for:

- `pid` is the unique identifier for each thread block, obtained using `tl.program_id(axis=0)`.
- The grid is divided into groups of thread blocks (`GROUP_SIZE_M`). Each group processes a portion of the output matrix.
- `pid_m` and `pid_n` are the coordinates of the tile in the M and N dimensions, respectively.
- Offsets (`offs_am`, `offs_bn`, `offs_k`) are calculated to determine which elements of matrices A and B each thread in the block will work on

2. Loading and Computing Tiles

The kernel uses a loop to iterate over the K dimension in chunks of `BLOCK_SIZE_K`. For each chunk:

- Load Tiles: tiles from matrices A and B are loaded from global memory.
- Unpacking Matrix B: The kernel assumes that matrix B is packed with `int8` values, meaning each element actually represents four smaller values packed into one byte. The unpacking happens within the loop:
    - `b_uint8` is loaded from global memory as packed `int8`.
    - Each packed value is unpacked to obtain the actual weight values used for computation.
- Dot Product: The kernel computes the dot product of the loaded tiles from A and B, accumulating the results in the `accumulator`. The `accumulator` stores the partial results for the tile of the output matrix C.

3. Storing Results

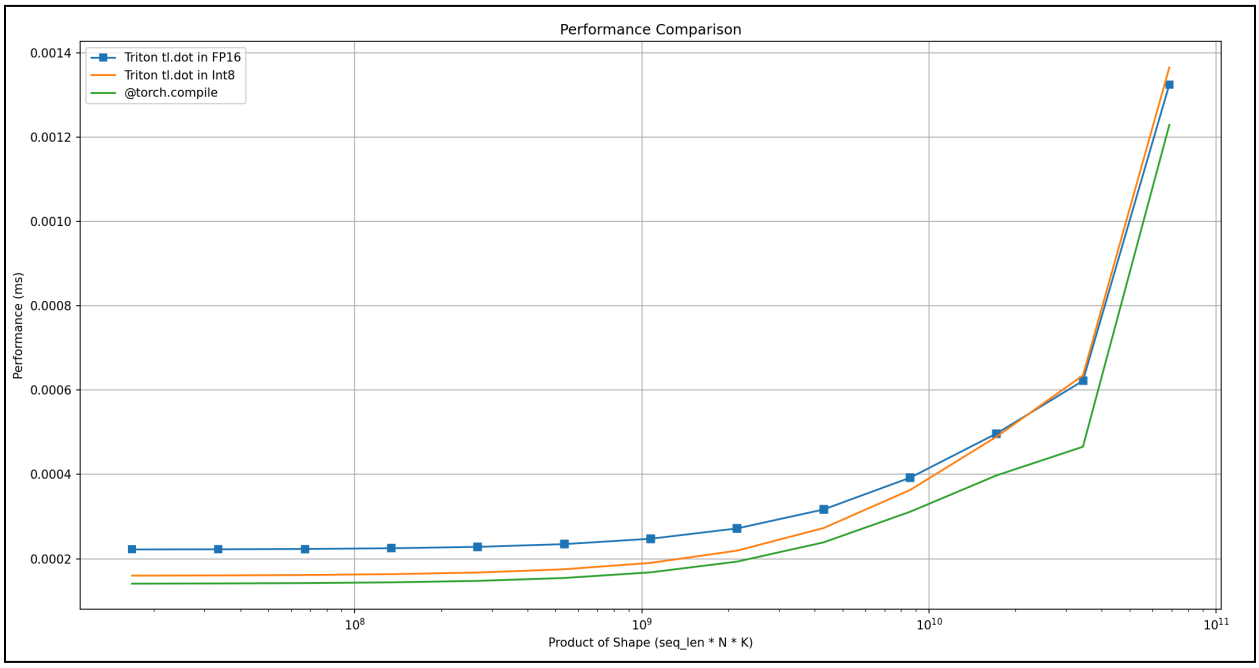
After all tiles along the K dimension have been processed, the final results stored in the `accumulator` are converted to `float16` and written back to the corresponding tile of matrix C in global memory. The writing process respects memory boundaries using a mask to ensure that only valid elements are written.

For a more detailed explanation of the code, checkout this PR

Benchmark

We benchmarked our kernel against the method of unpacking the weights using `@torch.compile` followed by performing the matmul in BF16 precision, and found that both approaches achieved approximately the same performance. To ensure accurate benchmarking, we performed the matmul operation over 2000 iterations and averaged the time taken during the last 1000 iterations, to eliminate

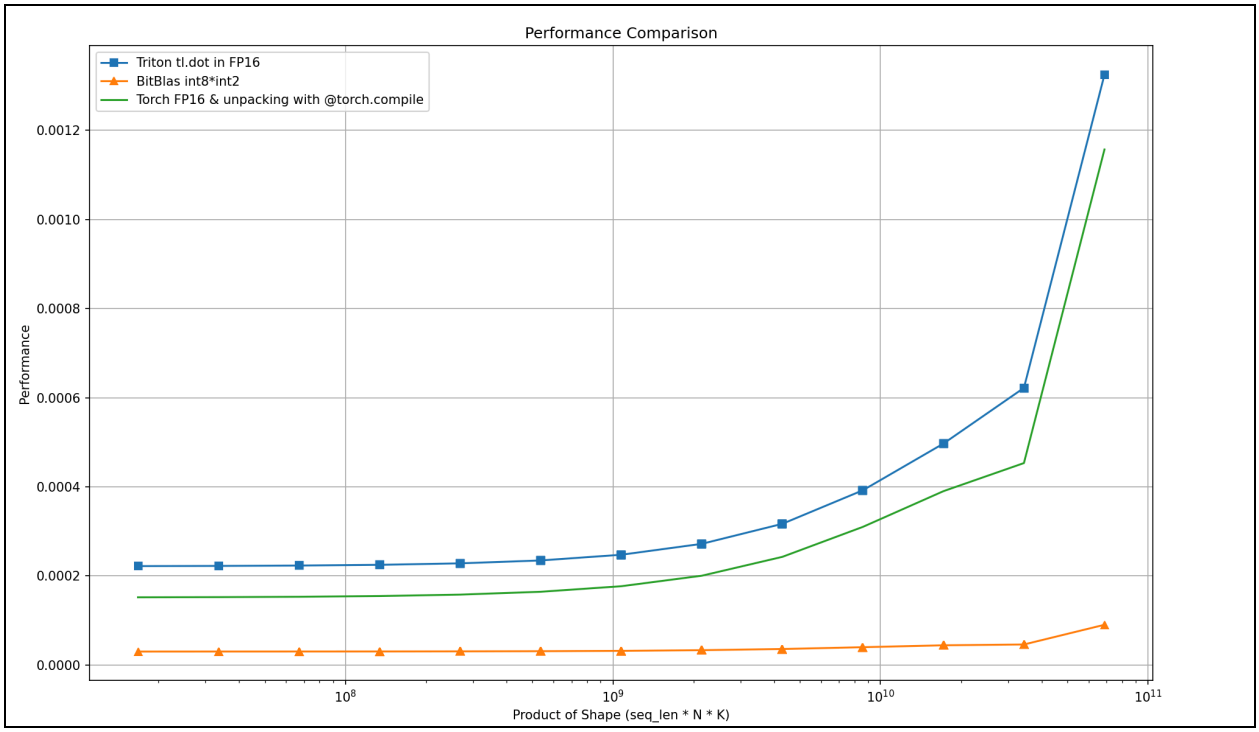
any inefficiencies related to initial loading or compilation. Below is a graph showing the benchmark results. We also tested various matrix sizes, with the x-axis representing the number of multiplications on a log scale, and the y-axis showing the average time in ms.



Triton kernel compared to torch.compile

We also tried using BitBlas, which is a software library designed to perform matrix operations with mixed precision. It helps optimize these operations by allowing calculations to be done in lower precision formats like INT8, INT4, or even INT2, instead of the traditional FP32 or FP16 formats.

The benchmark results are promising, as BitBlas outperforms both our custom kernel and Torch's `matmul` function in low precision, as shown in the graph.



Bitblas benchmark

However, during model loading, BitBlas needs to compile kernels tailored to the shape of the weight matrix and store them in a local database, which can increase the initial loading time.

Conclusion

In conclusion, as LLMs continue to expand, reducing their computational demands through quantization is essential. This blog has explored the approach of 1.58-bit quantization, which uses ternary weights. While pre-training models in 1.58 bits is resource-intensive, we've demonstrated that, with some tricks, it's possible to fine-tune existing models to this precision level, achieving efficient performance without sacrificing accuracy. By optimizing inference speed through specialized kernels, BitNet opens new possibilities for making LLMs more practical and scalable.

Acknowledgements

We would like to express our sincere gratitude to Leandro von Werra, Thomas Wolf, and Marc Sun for their invaluable assistance and insights throughout this project. We also extend our thanks to Omar Sanseviero and Pedro Cuenca for their contributions in refining this blog post, helping to communicate our findings clearly and effectively to the AI community. Furthermore, we want to acknowledge the GeneralAI team for their pioneering work on the BitNet project. Their research has been foundational to our efforts, and we are particularly grateful for the clear and precise figures provided in their paper.

Additional Resources

1. H. Wang et al., *BitNet: Scaling 1-bit Transformers for Large Language Models*. arxiv paper
2. S. Ma et al., *The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits*. arxiv paper
3. S. Ma et al., *The Era of 1-bit LLMs: Training Tips, Code and FAQ*. linkEditSign
4. RJ. Honicky, *Are All Large Language Models Really in 1.58 Bits?*. blogpost
5. L. Mao, *CUDA Matrix Multiplication Optimization*. blogpost
6. *Tutorial: OpenCL SGEMM tuning for Kepler*. link
7. *CUDAMODE*. github, youtube
8. Wen-mei W. Hwu, David B. Kirk, Izzat El Hajj, *Programming Massively Parallel Processors : A Hands-on Approach*

More Articles from our Blog

Universal Assisted Generation: Faster Decoding with Any Assistant Model

By

danielkoratOctober 29, 2024guest•10

Scaling AI-based Data Processing with Hugging Face + Dask

By

scj13October 9, 2024•22

Upvote

193









+181