ARCTIC LONG SEQUENCE TRAINING: SCALABLE AND EFFICIENT TRAINING FOR MULTI-MILLION TOKEN SEQUENCES

Stas Bekman, Samyam Rajbhandari, Michael Wyatt, Jeff Rasley,
Tunji Ruwase, Zhewei Yao, Aurick Qiao and Yuxiong He
Snowflake AI Research
{stas.bekman, samyam.rajbhandari, michael.wyatt, jeff.rasley}@snowflake.com
{tunji.ruwase, zhewei.yao, aurick.qiao, yuxiong.he}@snowflake.com

ABSTRACT

Long sequences are critical for applications like RAG, long document summarization, multi-modality, etc., and modern LLMs, like Llama 4 Scout, support max sequence length of up to 10 million tokens. However, outside of enterprise labs, long sequence training is challenging for the AI community with limited system support in the open-source space.

Out-of-box, even on a modern NVIDIA H100 80GB GPU cluster, training Llama 8B model with sequence over 32K runs out of memory on a basic Hugging Face (HF) model due to two reasons: i) LLM training workloads are not optimized to fully leverage a single GPU memory, ii) existing solutions for leveraging multiple GPU memory are not easily available to HF models, making long sequence training inaccessible.

We address this with Arctic Long Sequence Training (ALST). It offers a combination of attention-agnostic single GPU and multi-GPU memory optimizations, that enables it to support out-of-box training of multi-million sequence length for a wide variety of HF models.

ALST supports training Meta's Llama 8B model with 500K sequence length on a single H100 GPU, 3.7M on a single 8xH100 GPU node, and over 15M on a 4 node cluster, an increase of over 400x compared to the 32K baseline for the latter. ALST is fully compatible with HF models and open-sourced via DeepSpeed and Arctic Training.

Keywords Long Sequence · Sequence Parallelism · Post-Training · Fine-Tuning · Tiled Compute

1 Introduction

Long sequence capability offers the ultimate unlock for a wide range of AI applications from RAG, multi-turn conversation, long document summarization, multi-modality support, and many more. This is evident from the continuous increase in the max sequence length supported by popular Open Source LLMs, like Meta's Llama 3.x and Alibaba's Qwen 2.5 32B, which support 128K-token sequences, NVIDIA's Llama-3.1-8B-UltraLong-4M-Instruct finetuned to support a 4M sequence length, and the more recent Meta's Llama-4 Maverick and Llama 4-Scout models supporting 1M and and a whopping 10M sequence length, respectively.

While these models are capable of handling incredibly long sequence lengths, for several reasons, fine-tuning these models at these sequence lengths to enhance task-specific capabilities is out of reach for most data scientists who do not have access to sophisticated enterprise training systems and rely on open source solutions:

- First, standard LLM training workflows are not optimized for memory efficiency, which restricts the maximum sequence length per GPU and makes them suboptimal for long-sequence training.
- Second, training with multi-million sequence lengths requires more memory than available in any commercially available GPU devices, and while there are solutions that allow for leveraging aggregate GPU memory across multiple devices, they are limited. For example, Ring-based sequence parallelism [1] does not support arbitrary

attention patterns natively. While Ulysses-based sequence parallelism [2] does not have this restriction, it is not supported in popular frameworks like Hugging Face, limiting its accessibility.

- Third, PyTorch itself suffers from a multitude of memory bottlenecks that limit the available memory available to support long sequences.

In the Open Source release of Arctic Long Sequence Training (ALST) we address the above challenges with three targeted solutions:

- Ulysses Sequence Parallelism Compatible with Hugging Face Transformers: Adapted from the original Megatron-DeepSpeed Ulysses [2] and extended to support modern Attention mechanisms, this technique enables the use of aggregate GPU memory across multiple devices.
- Sequence Tiling for Memory Efficiency: A new computation tiling pattern for LLM training that reduces the memory required for memory intensive operators like logits, loss, and MLP computation from O(N) to O(1), where N is the sequence length.
- PyTorch Memory Optimizations: Through comprehensive memory profiling of long-sequence training workloads, we identified and applied a series of PyTorch-specific optimizations to eliminate the unnecessary memory overheads.

By leveraging these three components and making them compatible with Hugging Face Transformers, ALST makes long-sequence training accessible to the broader AI community:

- 500K-long sequence training on a single H100 GPU, 16 times longer than baseline[1], democratizing long-sequence training on resource constrained setups.
- 3.7M-long sequence training on a single H100 node, with a 116 times improvement relative to baseline[1].
- 15M-long sequence training on four H100 GPU node cluster, with a 469 times improvement relative to baseline[1].
- The technology is agnostic to the attention mechanism, allowing for out-of-box support for different sparsity patterns like block sparse, MoBA, etc.

Figure 1 provides a quick preview of the ALST accomplishments, which we will discuss in detail in later sections. [2]

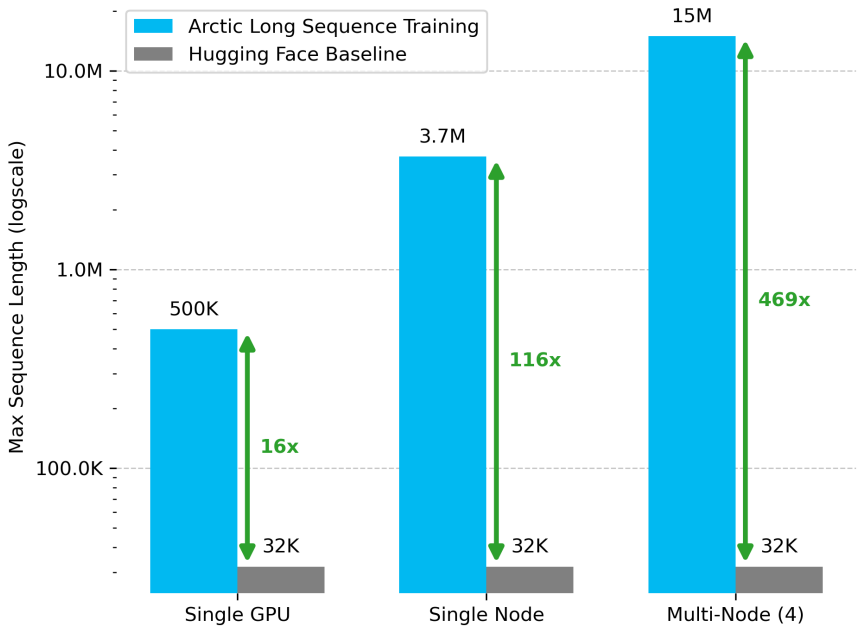

Figure 1: A dramatic improvement in sequence length with ALST enabled on 1, 8 and 32 H100 GPUs with Llama-8B. The baseline is Hugging Face with DeepSpeed ZeRO Stage 3 and optimizer states offload to CPU

[1]The best prior to this work setup we found is explained in section 5.4.

[2]A log scale had to be used in order to visualize the baseline, since otherwise for the 469x improvement it won't even show up.

Besides Ulysses SP [2] there are other approaches to sequence parallelism (also referred to as context parallelism). One approach was introduced by Megatron-LM [3] which extends Tensor Parallelism (TP) and which cannot operate without TP. And the more popular one, which can be used alone, is Ring Attention with many variants [1] [4] [5] [6], which resembles a distributed version of Flash Attention 2 [7]. Some techniques combine the Ring Attention and Ulysses SP approaches [8] [9]. The main obstacle with these approaches is that they require modeling code modifications whereas Ulysses SP is attention and model-agnostic.

In the rest of the paper, we will take a deeper dive into the memory challenges of long-sequence training, followed by a discussion of memory optimizations targeting these challenges. For our readers interested in deeper discussions, we also share our implementation details as well as the nuances of integration into Hugging Face Transformers. Finally, we present our evaluation results, as well as share how you can get started with ALST. For data scientists looking to experiment with long-sequence training, we will also share limitations and useful notes at the end of the paper.

2 Why is training with long sequences challenging?

Training models on long sequence lengths is a difficult task because models are large and the accelerator memory is typically insufficient to hold the large activations, especially when weights, optimizer states and gradients already consume a lot of the available GPU memory.

We used a combination of the PyTorch memory profiler and a helper see_memory_usage utility, that dumped memory usage stats at various places in the code, to identify where memory was allocated in a non-efficient way or not released soon enough.

2.1 Model Training Memory Map

Here is a detailed breakdown of what the GPU memory is used for:

1. **Weights + Optimizer states + Gradients**: In a typical BF16 mixed precision training about 18 bytes are needed per model parameter just to hold the model weights (2), optimizer states (8+4) and gradients (4). For example, Llama-3.1-8B-Instruct contains 8 billion parameters and thus requires 16GiB for BF16 weights, 64GiB for Adam optimizer states, 32GiB for FP32 weights used for optimizer stability, and finally 32GiB for FP32 gradients. Therefore in total each GPU already requires 144GiB of memory to train this 8B-parameter model before all the other overheads. [3]
2. **Activations**: Then there is memory required to calculate and hold activations - which is all the intermediate tensors that the model uses. This includes a variety of temporary tensors as well. The tricky part about activation memory is getting tensors that are no longer needed released as soon as possible. For example, activation checkpointing is often used to help reduce the required active memory by recalculating the intermediate forward activations during the backward call.
3. **Runtime overheads**: We observe that the remaining GPU memory is consumed by several key sources. Lower-level libraries such as CUDA and NCCL each reserve a significant amount of memory—CUDA typically uses around 1 GiB per GPU, while NCCL can consume multiple gigabytes for internal buffers and data structures, which are more difficult to track[4]. Additionally, different versions of PyTorch can exhibit varying memory usage patterns due to leaks or inefficiencies. Finally, when operating near the limits of available GPU memory, memory fragmentation becomes a concern, reducing the availability of large contiguous memory blocks.

To better understand memory requirements for different models, GPU counts, and total sequence length, we suggest the API to estimate memory usage in DeepSpeed ZeRO.[5].

2.2 Activation Memory is the Primary Bottleneck

Only aspect seq-dependent

Now that you understand what GPU memory is used for it should be easy to understand that in order to go from a short sequence length to a long one it's mainly more of the activation memory that we need to fit. All other memory allocations remain the same regardless of the sequence length. Figure 2[6] shows how activation memory for Llama-8B increases linearly with sequence length.[7].

[3] There are many mixed precision training recipes but we find the one we use to be the most practical.

[4] PyTorch memory profiler doesn't track NCCL internal memory allocations.

[5] See also: Interactive training memory calculator.

[6] Compiled with the help of Interactive training memory calculator.

[7] Memory here is activation checkpoints memory + activation and logits working memory.

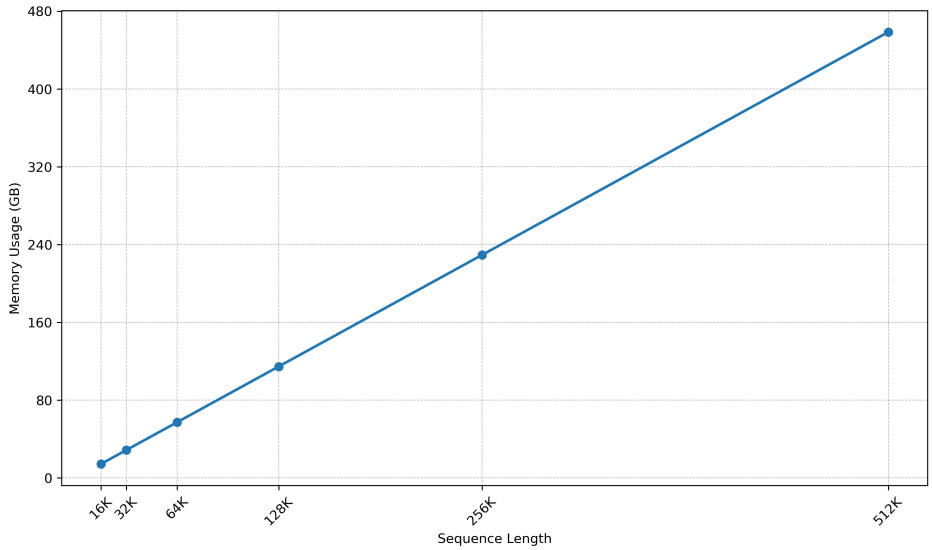

Figure 2: Estimated memory usage for Llama-8B activation memory with different sequence lengths.

3 Memory Optimizations

Next we will discuss three memory optimization groups that were instrumental at enabling training at very long sequence lengths:

1. Sequence tiling for reducing activation memory footprint
2. Ulysses Sequence Parallelism for Cross-GPU Activation Memory Sharing
3. Activation offloading and other PyTorch optimizations

3.1 Sequence Tiling for Reducing Activation Memory Footprint

GPU memory requirements for training on long sequences grow rapidly with sequence length increase. As part of our activation memory calculations 2.2, we estimated the activation and logits memory needed for various models and sequence lengths. Without our optimizations, the per-GPU memory usage quickly becomes unsupportable [8]—as shown in Figure 2—and this doesn't even include model parameters or optimizer states discussed earlier.

To address this memory explosion, we leverage *Sequence Tiling*, a technique that reduces peak memory usage by tiling forward and backward computations along the sequence dimension. Instead of processing the entire sequence in a single pass — which requires storing large intermediate tensors — Sequence Tiling breaks the computation into smaller tiles. Intermediate values are materialized only for each tile, significantly lowering the memory footprint.

This approach is applicable to operations that have no cross-sequence dependencies, such as linear layers, token embeddings, and per-token loss computations. For example, instead of computing logits or applying MLP layers across the entire sequence at once, we apply them sequentially to smaller segments, storing only the necessary intermediates at each step.

Let's start by examining how effective Sequence Tiling is at reducing memory overhead during loss calculations. Using the example of Llama-3.1-8B-Instruct with a sequence length of 16K, the model's vocabulary size of 128,256 results in a single copy of the logits in FP32 consuming approximately 8GiB of memory per GPU (calculated as $4 \times 16\_000 \times 128\_256/2^{30} = 7.65GiB$). Rather than materializing all 8GiB of logits at once for both the forward and backward passes, we shard the logits into smaller chunks and compute the forward and backward passes on each shard independently. This significantly reduces peak memory usage. For instance, using a 1GiB shard size divides the computation into about 8 chunks and can save over 14GiB of memory in practice (because the loss computation uses 2 times of 8GiB).

[8] i.e. hitting the Out-Of-Memory event.

Up to this point we have discussed memory reductions from a theoretical perspective, we can also showcase these memory reductions empirically via the help of the PyTorch memory profiler. In Figure 3 we show two plots: (left) without Sequence Tiling the loss calculation we see peak memory usage is around 50 GiB compared to (right) after updating the loss calculation to use Sequence Tiling we see the peak memory usage drops to 36 GiB which results in a 28% memory reduction. [9] [10]

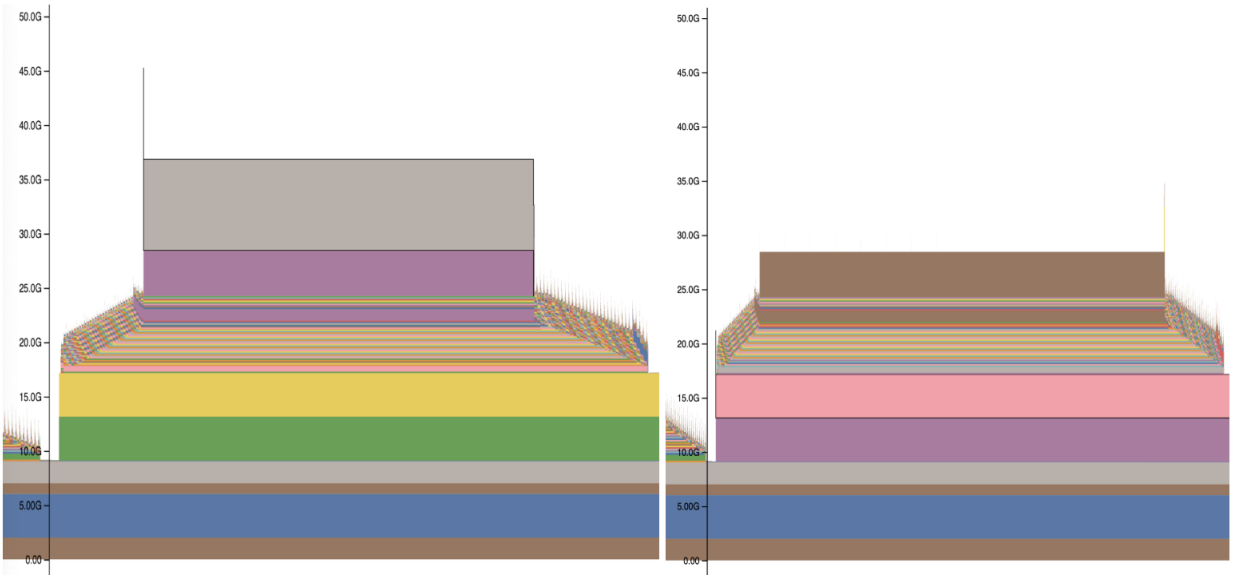

Figure 3: PyTorch memory usage plots before (left) and after (right) using Sequence Tiling to reduce loss calculation memory usage

This is not the first time a tiled compute is used. Here are some examples of recent use relevant to our discussion:

- For some years DeepSpeed's TiledLinear has been enabling much bigger compute loads that otherwise would have not fit into the GPU memory.
- Liger-Kernel [10] implemented a fused cross entropy loss, without manifesting the whole logits tensor first, thus enabling bigger batches and sequence lengths, but only for select most popular model architectures.

Now we introduce a generic TiledCompute autograd function that in theory should be able to make any function that performs large matrices multiplications use much less GPU memory. We implemented a fused cross-entropy using it, but Liger-Kernel's version of the same is a bit faster since it uses a Triton kernel. Liger-Kernel supports a limited number of popular Hugging Face Transformers architectures, but our solution in theory should work with any architecture, so we recommend using Liger-Kernel's fused cross entropy for the architectures it supports and when it doesn't, then you have the option of using our implementation.

3.1.1 TiledMLP

But why stop at tiling the logits+loss computation, we can also tile the MLP compute[11]. We created a simplified version of TiledCompute and called it TiledMLP[12] to also perform a much more memory-efficient MLP computation, which allowed a huge increase in the sequence length.

If we extract a single LlamaMLP layer from Llama-8B and run a bf16 hidden_states tensor of shape [1, 256_000, 4096] through its forward-backward, without and with sequence dimension tiling, we get about 10x memory saved as can be

[9] There is a very thin spike that goes to 49.5GiB left and 36GiB (right) but it can't be seen in the small snapshot of the memory profiler plot.

[10] The PyTorch memory profiler assigns colors at random, that's why most block colors don't match between the left and the right.

[11] The attention block can't be tiled because it needs the whole sequence to attend to.

[12] TiledMLP is almost the same as TiledCompute but its code is easier to understand as it's not generalized.

seen from Figure 4[13]. The number of shards was auto-deduced via $ceil(seqlen = 256\_000/hidden\_size = 4096) = 63$.

The evaluation section 5 shows the numerical improvements from enabling TiledMLP with full models.

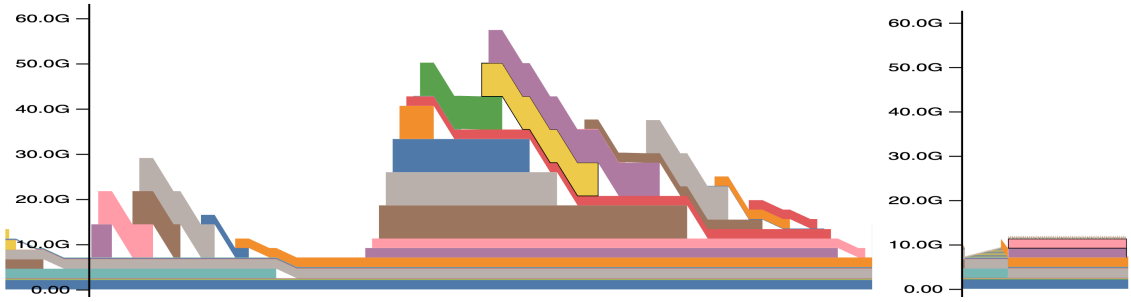

Figure 4: 1x forward-backward cycle on a single Llama-8B LlamaMLP layer (Left) without tiling (Right) with tiling.

3.2 Ulysses Sequence Parallelism for Cross-GPU Activation Memory Sharing

Now let's have a more detailed look at how UlyssesSPAttentionHF (Ulysses Sequence Parallelism Attention for Hugging Face) works.

1. Starting with sequence parallelism, the sequence is split across participating GPUs and executed in parallel till the attention block is reached.
2. Since self-attention requires an entire sequence, at this boundary we switch from sequence parallelism to attention head parallelism
3. When attention block completes we switch back to sequence parallelism

We will use the following diagram (Figure 5) from the Arctic Ulysses Inference blog post to explain how this works in details:

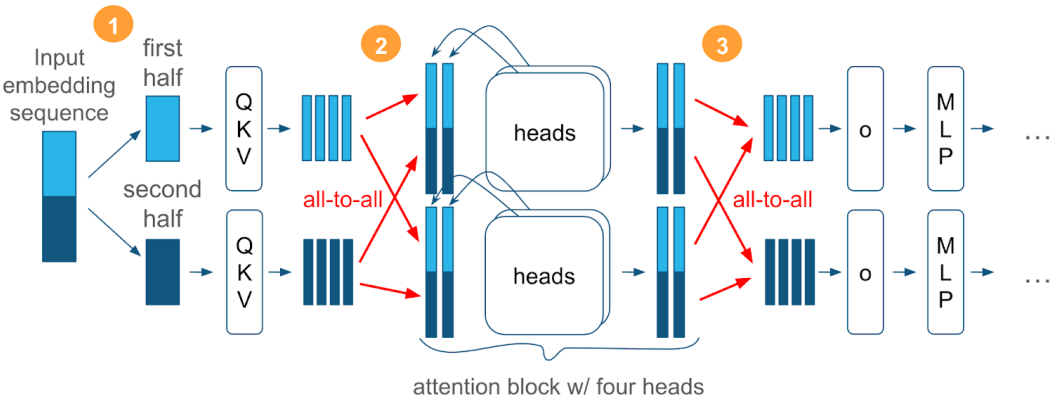

Figure 5: Ulysses Sequence Parallelism diagram with 4 attention heads per attention block model.

Since communications of the attention head projections cannot be overlapped with compute, they have to be really fast. And that's why all-to-all communication collective is used. Before the all-to-all communication, each rank has a partial sequence for all attention heads; however, after the all-to-all communication, each rank has the entire sequence, but only for a partial subset of attention heads. This allows each rank to compute the attention for the subset of the attention heads that it owns in parallel. Then after the attention, Ulysses SP performs another all-to-all communication to switch back to the original SP layout, where each rank once again has the full embedding (attention heads) but a shard of a sequence. The computation then proceeds in parallel until the next layer's attention is reached.

[13] 10-60GiB vs 7-12GiB memory used for forward-backward compute, the rest are static invariant structures.

Also in Ulysses paper

The reason Ulysses SP is attention algorithm-agnostic is because at the point of calculating the attention it recomposes the full sequence length and passes it to the desired attention mechanism (e.g., FlashAttention2 [7] or SDPA). Whereas in other SP approaches (e.g., Ring Attention [1]) the attention mechanism itself must be adapted to the specific attention algorithm being used.

3.2.1 Extending Ulysses for Modern Attention Mechanism

Figure 6 provides a visual representation of MHA, GQA and MQA types of model attention heads.

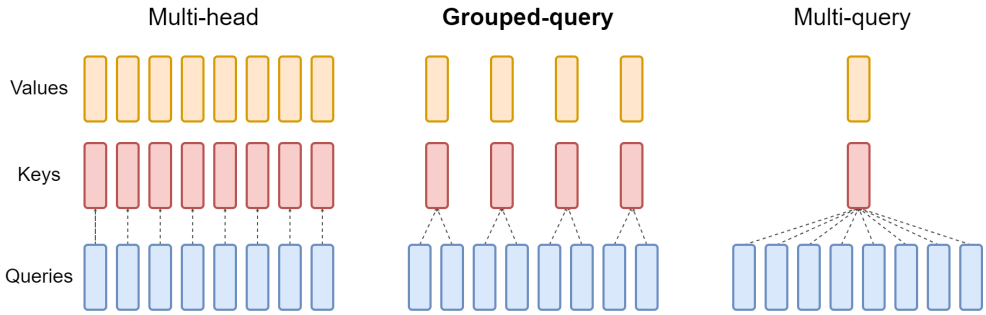

Figure 6: MHA, GQA and MQA types of model attention heads (source)

The original Ulysses SP implementation in Megatron-DeepSpeed only supports the MHA type of attention mechanism. Ulysses SP for HF was extended to support all three of the aforementioned types:

1. MHA is the simplest since it has the same number of q and kv heads. We simply split qkv_heads into SP shards. Clearly here the only limitation is that qkv_heads is divisible by SP degree.
2. GQA type has kv < q number of heads, since the kv heads get reused.
   a. If kv_heads is divisible by SP we shard q_heads into SP degree shards and kv_heads into SP degree shards.
   b. If kv_heads < SP then we replicate kv_heads to match SP degree.
3. MQA is where there is 1 kv_head and many q_heads. This is the same as 2b: we replicate kv_heads to match SP degree.

Examples:

- 32 q_heads, 8 kv_heads, sp=8 => each rank will have 4 q_heads, 1 kv_heads
- 32 q_heads, 8 kv_heads, sp=32 => each rank will have 1 q_heads, 1 kv_head (kv_heads will be replicated)
- 32 q_heads, 4 kv_heads, sp=8 => each rank will have 4 q_heads, 1 kv_heads (kv_heads will be replicated)

The kv_heads and q_heads count isn't always divisible by the desired SP degree. For example, if a model has kv_heads=3, q_heads=9 we can do SP=3 or SP=9 and not deploy the node fully to do SP=8*nodes. [14]

3.3 Activation Offload to CPU and Other PyTorch Optimizations

We employed several strategies to reduce runtime memory overhead. Here are the non-invasive ones:

- A deep memory analysis revealed that PyTorch versions have a significant impact on memory usage; due to a known issue with dist.barrier, we observed an excess memory usage of over 3 GiB in versions 2.6.0–2.7.0 and therefore standardized on version 2.8.0.dev20250507 (aka nightly) for our experiments. The just released torch==2.7.1 should be a solid alternative, where this and a few other related problems have been fixed.
- Additionally, while all_reduce_object offers convenience, it introduces an additional memory cost of over 3 GiB per GPU, so we opted to use all_reduce instead.

Need to check difference

[14] In the future we plan to come up with solutions that will be able to fit these nicely into 8x mode in a balanced compute.

- We activated PyTorch activation checkpointing to reduce memory usage, accepting a modest increase in compute overhead.
- Finally, to mitigate memory fragmentation, we enabled PyTorch's expandable segments allocator, which provided massive memory allocation improvements with minimal impact on overall performance.

Then we introduced a more invasive technique:

The activation checkpointing feature saves an insane amount of memory, but at large sequence lengths the checkpointed hidden_states tensors, which are of shape [bs, seqlen, hidden_size] still consume a lot of GPU memory. For example, at seqlen=125K/bs=1/hidden_size=4096/n_layers=32 it's 30.5GiB across all layers ($125\_000 \times 4096 \times 2 \times 32 / 2^{30} = 30.5$[15]. We monkey patched torch.utils.checkpoint.CheckpointFunction to offload the hidden_states activation checkpoint tensor to CPU - thus enabling a dramatically longer sequence length. [16].

Figure 7 shows a PyTorch memory profiler visualization of a single forward-backward iteration of Llama-8B with 32k sequence length. On the left you can see the usual pattern of memory usage growing with each layer running its forward calls (left upward slope), followed by memory usage going down during backward calls per layer (right downward slope), when the checkpointed tensors get released once they are no longer needed. On the right, this is the exact same setup, but with activation checkpoint offloading to CPU enabled. It's very clear to see that the "hill" is gone and we are now dealing with a flat structure, which leaves a lot more working GPU memory and allows for a much longer sequence lengths since the peak memory usage is no longer dependent on how many layers the model has.

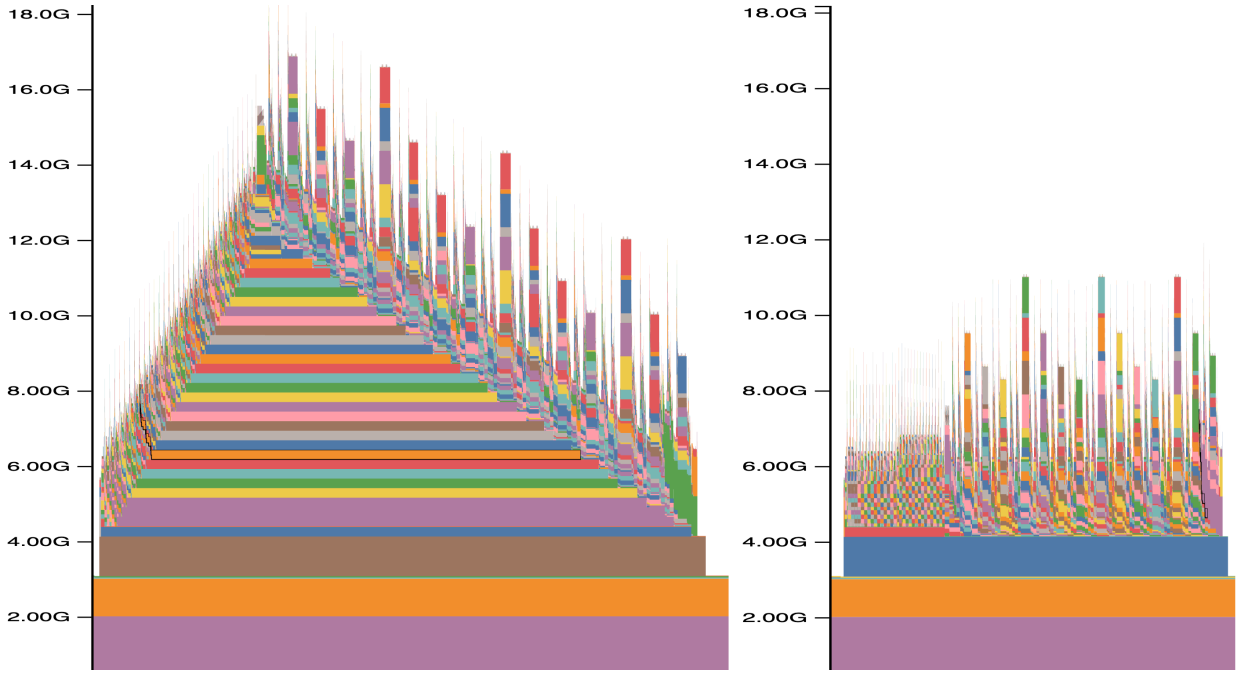

Figure 7: PyTorch memory profiler 1 iteration forward-backward CUDA memory usage: Left: normal setup. Right: with activation checkpoint offloading to CPU enabled

If you use a large model like Llama-70B you'd need to make sure you have sufficient CPU memory since hidden_states will get large and there are as many copies of this tensor as the number of layers. For example, Llama-70B at seqlen=3M, bs=1 and 32 GPUs needs 915GiB of CPU memory per node just for the activation checkpoint offloads

[15] Please note that in our experiments we always use batch size 1 when wanting a very long sequence length, because a larger batch size will require a lot more memory.

[16] This feature isn't optimized for speed yet - it needs to be switched from direct copy to asynchronous CUDA streams to allow overlap with forward compute. For backward it won't help much since it will need to wait till the data is copied back to GPU for compute to proceed - we are discussing with the PyTorch team how this feature could be implemented in the core and enabled with a simple flag in Hugging Face Transformers' from_pretrained which would make it much simpler for all users.

$(3\_000\_000/32 \times 8192 \times 80 \times 2/2^{30} \times 8 = 915GiB)$ (where hidden_size=8192, num_layers=80). So in this case the node's CPU memory is likely to be the limiting factor preventing an even bigger sequence length.

This feature can also serve smaller sequence lengths when a very large batch size is wanted.

3.4 4D Attention Mask is not possible with long sequence lengths

When using very long sequence lengths and packed samples deploying a 4D causal attention mask is not feasible because that tensor is of shape of [bs, seqlen, seqlen] so at bs=1 and seqlen=125K it requires a 29GiB tensor ($125\_000 \times 125\_000 \times 2/2^{30} = 29GiB$) and at 250K it would need 116GiB tensor on each GPU ($250\_000 \times 250\_000 \times 2/2^{30} = 116GiB$) and it grows quadratically with sequence length. Therefore, the only way to make the self-attention attend correctly to sub-samples in the batch without introducing a huge overhead is to use position_ids which are of shape [bs, seqlen], so in the case of 125K, it's just 0.2MiB ($125\_000 \times 2/2^{20} = 0.2MiB$).

Since currently we can't tell Hugging Face Transformers not to create the causal mask, other than when using Flash Attention 2, we had to monkey patch _update_causal_mask so that it won't create this mask using:

```
model_without_head = self.model_unwrapped.model
if hasattr(model_without_head, "_update_causal_mask"):
    model_without_head._update_causal_mask = lambda *args: None
```

4 Implementation and Hugging Face Transformers Integration Challenges

4.1 Challenges of Implementing Sequence Parallelism

We encountered three primary challenges when implementing sequence parallelism with Hugging Face Transformers models:

1. integrating with existing attention implementations
2. long-sequence data loading
3. loss sharding to avoid memory explosion

At its core, Ulysses Sequence Parallelism is designed to compose with existing attention implementations such as SDPA, Flash Attention 2, and others. While integration is straightforward in frameworks like Megatron-DeepSpeed, where the integration is done manually in the core, our approach focuses on extending existing attention modules within the training framework. This allows support for longer sequence lengths without requiring changes to the model's code itself.

Long-sequence data loading is particularly challenging, as each training sample is inherently large. If processed naively, this can lead to memory exhaustion—exactly what sequence parallelism aims to prevent. Our solution needed to handle these large sequences efficiently while remaining compatible with popular dataset providers such as Hugging Face Datasets [11].

Implementing loss sharding using Sequence Tiling (as introduced in section 3.1) required careful design. The goal was to avoid manual user intervention and prevent the need for modifications to model implementations which are outside of our control.

4.2 Integration with Hugging Face Transformers

We address these challenges through the following implementation spread out across Arctic Training [12], DeepSpeed [13], and in some cases small changes to Hugging Face Transformers [14] itself.

1. **Hugging Face Transformers injection** - Ulysses Sequence Parallelism Attention for Hugging Face (UlyssesSPAttentionHF) integrates into the modeling code by overriding the user-specified attn_implementation (e.g., sdpa, flash_attention_2) with ulysses, and injecting a custom wrapper into the Transformers backend via transformers.modeling_utils.ALL_ATTENTION_FUNCTIONS. This approach allows us to seamlessly wrap the user's intended attention implementation with Ulysses SP long-sequence support.
2. **DataLoader** - We introduce a specialized DataLoader adapter, UlyssesSPDataLoaderAdapter, which takes any existing DataLoader and automatically shards each batch along the sequence dimension. It then uses a

single rank's batch and processes it collaboratively across all ranks—effectively implementing a sequence-parallelism-over-data-parallelism protocol. In this setup, we iterate over ranks, processing one batch at a time using all ranks in parallel. This design preserves the traditional, iterative behavior of the original DataLoader, while enabling seamless integration with Ulysses Sequence Parallelism.

3. **Non-Attention blocks** - Each rank processes its shard of the input sequence independently up to the attention layer, as the preceding computations have no inter-token dependencies and can be executed in parallel.
4. **Attention block** - Details on how Ulysses handles attention can be found in section 3.2.

4.3 Loss Sharding in Hugging Face Transformers

In causal language models (e.g., GPT and Llama) cross-entropy loss requires labels to be shifted one position to the left because of how next-token prediction works. Cross-entropy loss compares the model's prediction at its current position to the correct token at the next position.

When computing loss in an unsharded batch we end up with (shift left):

```
input_ids   : [1 2 3 4 5 6 7    8]
labels      : [1 2 3 4 5 6 7    8]
shift_labels: [2 3 4 5 6 7 8 -100]
```

-100 is the special label value to be ignored so it gets pushed on the right.

If we naively shard on the sequence dimension (SP=2 in the following example), we end up losing the first label of each shard due to Hugging Face Transformers' loss function shifting each rank's inputs without being aware of our sharding strategy. This results in our example dropping the token id 5 entirely:

```
input_ids   : [1 2 3    4] [5 6 7    8]
labels      : [1 2 3    4] [5 6 7    8]
shift_labels: [2 3 4 -100] [6 7 8 -100]
```

To address this issue, we have modified the causal loss API in Hugging Face Transformers to support user-provided pre-shifted labels. Now we can pre-shift the labels before sharding on the sequence dimension, and thus end up with the correct labels passed to the loss function for each shard:

```
input_ids   : [1 2 3 4] [5 6 7    8]
labels      : [1 2 3 4] [5 6 7    8]
shift_labels: [2 3 4 5] [6 7 8 -100]
```

5 Evaluation

5.1 Overview

We evaluated the longest sequence length we could get for a range of configurations from 1 GPU to 64 GPUs (8 nodes)[17]. Several iterations were completed with each reported sequence length to ensure there are no memory leaks.

The hypothesis was that, once the minimal number of GPUs required to hold the model weights is met — without taking over the whole GPU memory — the maximum supported sequence length will scale linearly with the number of GPUs. And our results confirmed this hypothesis.

By sharding model weights using ZeRO Stage 3 [15], each additional GPU reduces the memory load per device, freeing up more memory for activation storage and enabling longer sequence lengths. In some cases, we even observed a slightly superlinear increase in maximum sequence length.

However, in a few scenarios where the amount of CPU offloading was very large, we run into a bottleneck of not having enough CPU memory to show the full potential of these techniques.

The evaluation results are broken down into 3 sections:

1. Maximum achieved sequence length - section 5.3
2. Feature ablations - section 5.4

[17] While ensuring the loss was valid. We have noticed that in some situations when trying to use the last few GiBs of GPU memory the loss would go to NaN.

3. Sequence length improvements over baseline - section 5.5

As the initial goal was to enable long sequence lengths for post-training, which usually takes just a few days of compute time, we weren't too concerned with the best performance, but focused primarily on the longest sequence length we could achieve in reasonable time, offloading things when it was helpful and not doing that when it was badly impacting the performance. Subsequent work will focus on improving the performance further.

5.2 Methodology

We will use 1 to 8 node configurations to run the experiments. Each node is made of 8x H100 80GiB. The nodes are interconnected with EFA v2 (AWS) that can do ~200GBps of all-reduce throughput. The intra-node connectivity is 450GBps NVLink-4.

Software versions used:

- torch==2.8.0.dev20250507 (a.k.a. torch-nightly)[18] torch>=2.7.1 should be as good[19])
- flash_attn==2.7.4 (flash_attn>=2.6.4 delivers a very similar performance)
- transformers==4.51.3 (transformers>=4.51.3 should be fine)
- deepspeed==0.17.0 (deepspeed>=0.17.0 should be fine)

The following optimizations were enabled during all, but ablation experiments:

- DeepSpeed ZeRO Stage 3
- DeepSpeed optimizer states offload to CPU
- Gradient/Activation checkpointing
- Fused tiled logits+loss computation via Liger-Kernel [20]
- PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True environment variable
- Sequence parallel communications were done in bf16 (or could reduce the communication buffer size instead)
- Tiled MLP computation
- Activation checkpoint hidden_states tensor offload to CPU

Additionally, when we used a single GPU, we also enabled model weights offload to CPU to prevent GPU OOM, which otherwise would occur even with a tiny sequence length.

5.3 Maximum Achieved Sequence Length

We measured the maximum achievable sequence length with three popular representative models by zeroing in on the maximum length that would not provide out of memory, loss=NaN and other errors.

5.3.1 meta-llama/Llama-3.1-8B-Instruct

meta-llama/Llama-3.1-8B-Instruct has 32 q_heads and 8 kv_heads and thus can be trained on 1 to 32 GPUs.

Figure 8 shows the measured outcomes.

[18] we found that torch<2.7.1 versions either had memory leaks or weren't as memory-efficient.

[19] torch==2.7.1 version had been released when this paper was about to be completed.

[20] all of Liger-Kernel's Llama features were enabled, except swiglu when TiledMLP was used, because Liger-Kernel monkey patches LlamaMLP as well when swiglu is enabled, leading to a conflict with TiledMLP.

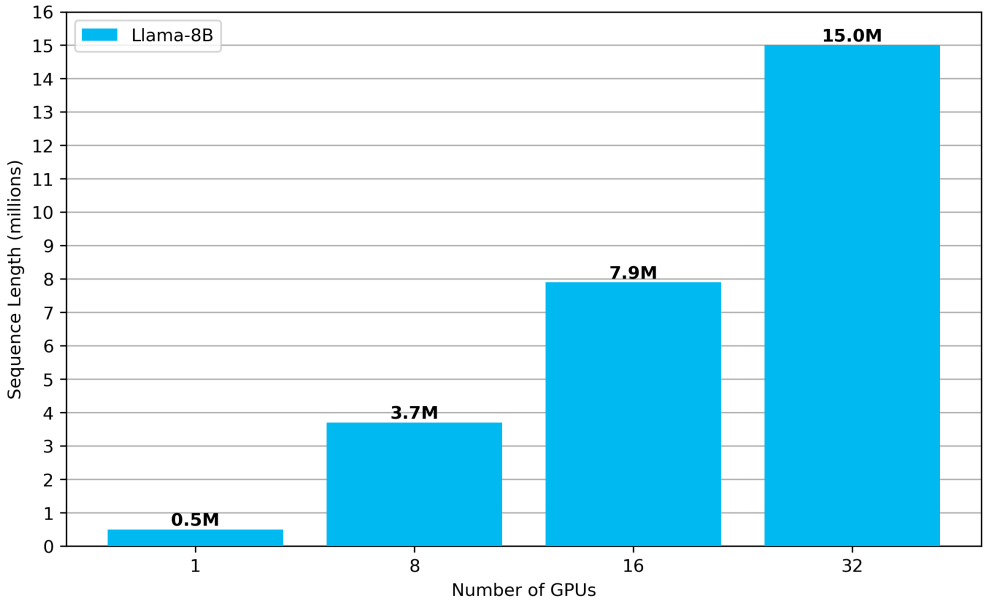

Figure 8: Maximum achieved sequence length with meta-llama/Llama-3.1-8B-Instruct

5.3.2 meta-llama/Llama-3.1-70B-Instruct

meta-llama/Llama-3.1-70B-Instruct has 64 q_heads and 8 kv_heads and thus can be trained on 16 to 64 GPUs. At least 8 GPUs are needed to just fit the sharded model and gradients, while offloading optimizer states to CPU.

Figure 9 shows the measured outcomes.

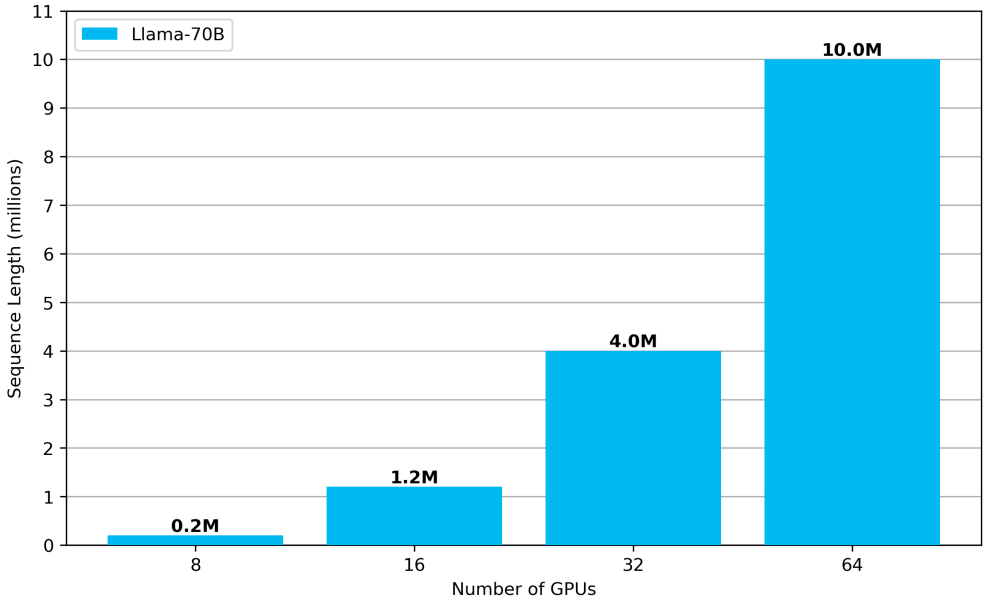

Figure 9: Maximum achieved sequence length with meta-llama/Llama-3.1-70B-Instruct

As you can see from the notes, we couldn't unleash the full sequence length potential for this model because activation checkpoint offload to CPU memory requirements for this model are so big:

- 4 nodes: $1\_000\_000/32 \times 8192 \times 80 \times 2/2^{30} \times 8 = 305GiB$ per 1M seqlen

- 8 nodes: $1\_000\_000/64 \times 8192 \times 80 \times 2/2^{30} \times 8 = 152GiB$ per 1M seqlen

And we had only 1.9TB of cpu memory available per node. We know that we "left more sequence length on the table", because the GPU memory was only about ¾ full.

5.3.3 Qwen/Qwen3-32B

Qwen/Qwen3-32B has 64 q_heads and 8 kv_heads and thus can be trained on 1 to 64 GPUs.

Figure 10 shows the measured outcomes.

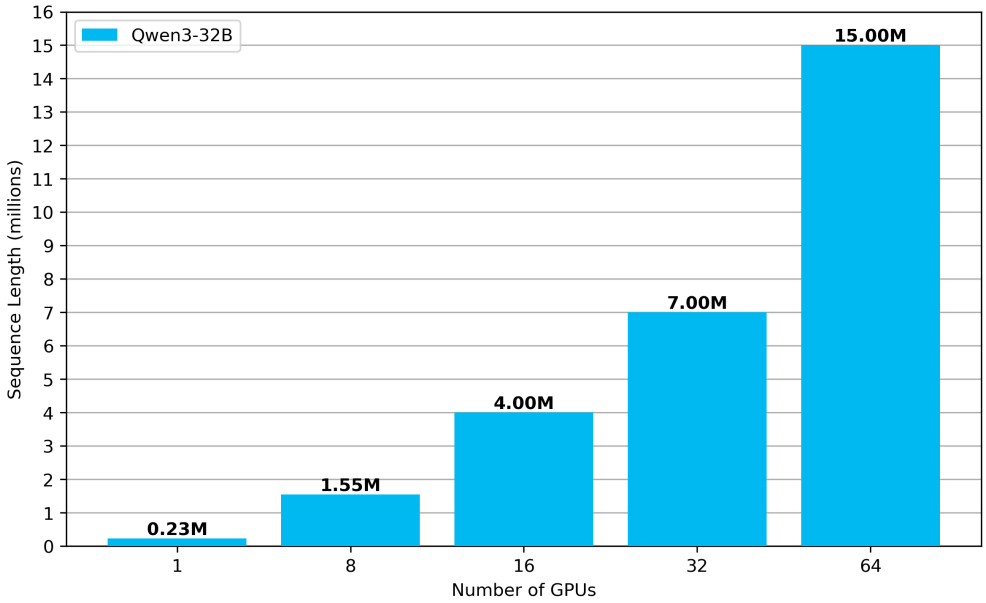

Figure 10: Maximum achieved sequence length with Qwen/Qwen3-32B

For a single GPU an additional weights offload to cpu was required.

Same as with Llama-70B for a few configurations we discovered that 1.9TiB of CPU memory weren't enough to get an even longer sequence length. For this model activation checkpoint offload to CPU memory requirements were:

- 4 nodes: $1\_000\_000/32 \times 5120 \times 64 \times 2/2^{30}times8 = 152GiB$ per 1M seqlen
- 8 nodes: $1\_000\_000/64 \times 5120 \times 64 \times 2/2^{30} \times 8 = 76GiB$ per 1M seqlen

5.3.4 Summary

As can be seen from the evaluation numbers for three different models, the possible sequence length growth is roughly linear, that is doubling the number of nodes, doubles the possible sequence length. In fact it's a bit better than linear because of DeepSpeed ZeRO Stage 3, which partitions the model constituents into shards across all GPUs, so the more nodes are used the smaller the shards owned by each GPU, and as a result the more GPU memory is available for activations.

5.4 Feature Ablations

For the feature ablation experiments we use a single 8x H100 node.

Baseline:

1. DeepSpeed ZeRO Stage 3
2. Gradient/Activation checkpointing enabled

3. DeepSpeed Optim states offload to CPU
4. PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True
5. Flash Attention 2 [7]

We next perform feature ablations on each of the following features and show the outcome in Table 1:

1. Fused tiled logits & loss compute with Liger-Kernel
2. Ulysses Sequence Parallelism for Hugging Face
3. Tiled MLP
4. Activation checkpoint offload to CPU

Table 1: Feature ablations results.

Base line	Tiled Logits & Loss	Ulysses SP for HF	Tiled MLP	Activation checkpoint offload to CPU	Sequence length (bs=1)	Iteration time h:mm:ss	TFLOPS
✓					32K	0:00:17	231.6
✓	✓				160K	0:02:03	514.4
✓	✓	✓			1.1M	0:09:24	576.1
✓	✓	✓	✓		1.2M	0:11:43	548.7
✓	✓	✓		✓	2.4M	0:43:30	585.8
✓	✓	✓	✓	✓	3.7M	1:47:35	590.6

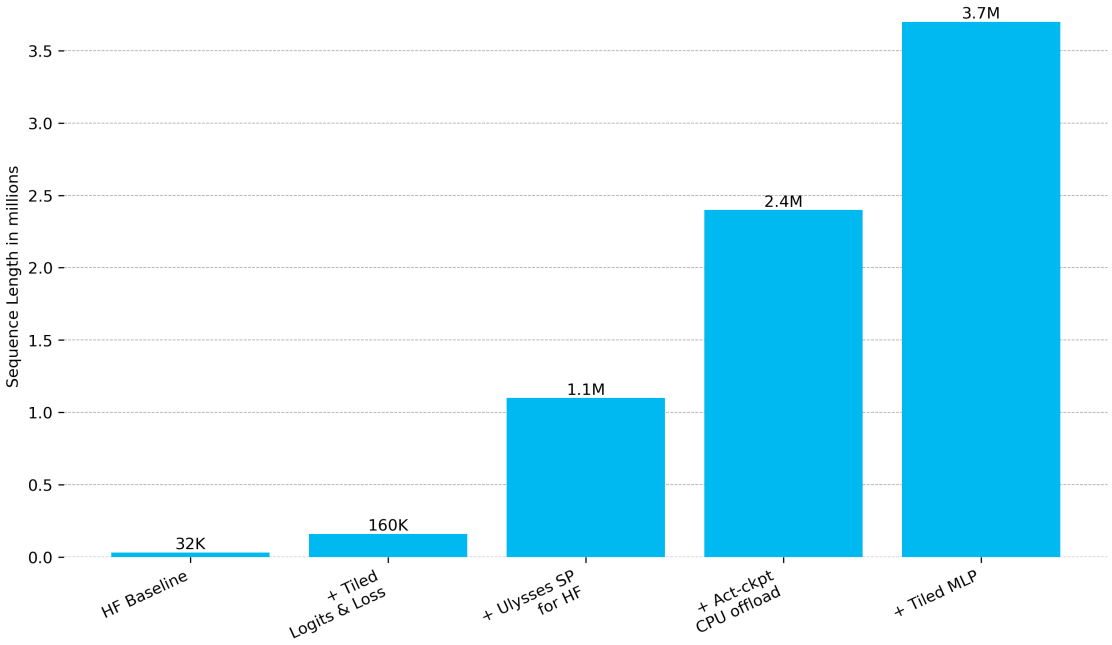

Figure 11: Feature ablation results visualized

Table 1 and the corresponding Figure 11 show that tiling computation features like tiled logits & loss and tiled MLP don't contribute much at a low sequence length, but once activation checkpoint offload to CPU enabled a much larger

sequence length, then for example tiled MLP was able to increase the sequence length by an additional 58%. Activation checkpoint offload to CPU and tiled MLP together enabled a sequence length that's ~3.5 times larger than the baseline + fused tiled logits & loss (Liger Kernel) + Ulysses SP for HF. Once we hit sequence lengths larger than 5M tiled MLP starts to massively contribute to allowing a much larger sequence length. hidden_states have a [bs, seqlen, hidden_size] shape and while hidden_size remains the same, seqlen becomes very big, leading to many GiBs-large hidden_states tensor for each layer.

Since the attention computation is quadratic with regards to sequence length, it's easy to see how the iteration time dramatically slows down as sequence length grows.

Additional notes:

- If Liger-Kernel doesn't support the architecture of your need, we have implemented Sequence Tiled Compute that can perform a tiled cross-entropy loss that should work with any causal model. It saves approximately the same amount of memory but it's slower than Liger-Kernel as it's written in pure PyTorch. It can be found here.
- When Tiled MLP is enabled Liger-Kernel's swiglu override is turned off since the 2 compete with each other over Hugging Face Transformers' modeling MLP class override[21].
- The TFLOPS and Iteration time were measured while using non-packed samples and using the standard Megatron-LM flos[22] estimation formulae taking into account repeated forwards. At such a long sequence length attention computation renders MLP compute negligible. We observed packed samples with FlashAttention2 setups report much lower TFLOPS.
- Since all 8 GPUs compute a single batch, and the micro-batch size is 1, the effective global batch size is 1 as well.

5.5 Sequence Length Improvements over Baseline

After activating ALST with LLama-8B the sequence length improvements were as following:

- 1 GPU: from 32K[23][24] to 500K, a 16 times improvement, as demonstrated by Table 2 and Figure 12.
- 8 GPUs: from 32K[23] to 3.7M, a 116 times improvement, as demonstrated by Table 3 and Figure 12.
- 32 GPUs: from 32K[23] to 15M, a 469 times improvement, as demonstrated by Table 4 and Figure 12.

Table 2: Sequence length improvement for Llama-8B on a single H100 GPU

Baseline	ALST	Sequence length (bs=1)	Iteration time h:mm:ss	TFLOPS
✓		32K	0:00:26	189.4
✓	✓	500K	0:16:50	548.1

Table 3: Sequence length improvement for Llama-8B on 8 H100 GPUs (1 node)

Baseline	ALST	Sequence length (bs=1)	Iteration time h:mm:ss	TFLOPS
✓		32K	0:00:17	231.6
✓	✓	3.7M	1:47:35	590.6

[21] but if one wants Liger-Kernel's efficient swiglu it should be possible to combine the two together with some additional work.

[22] to prevent ambiguity we use the abbreviation flos instead of flops, which was coined during BLOOM-176B work[16], as flops could mean floating point operations per second and also just floating point operations - but here we refer to the latter.

[23] the same baseline as in all evaluations.

[24] this setup's baseline additionally includes model weights offload to CPU, otherwise the setup couldn't fit on a single GPU without OOM.

Table 4: Sequence length improvement for Llama-8B on 32 H100 GPUs (4 nodes)

Baseline	ALST	Sequence length (bs=1)	Iteration time h:mm:ss	TFLOPS
✓		32K	0:00:12	393.6
✓	✓	15M	7:25:09	590.6

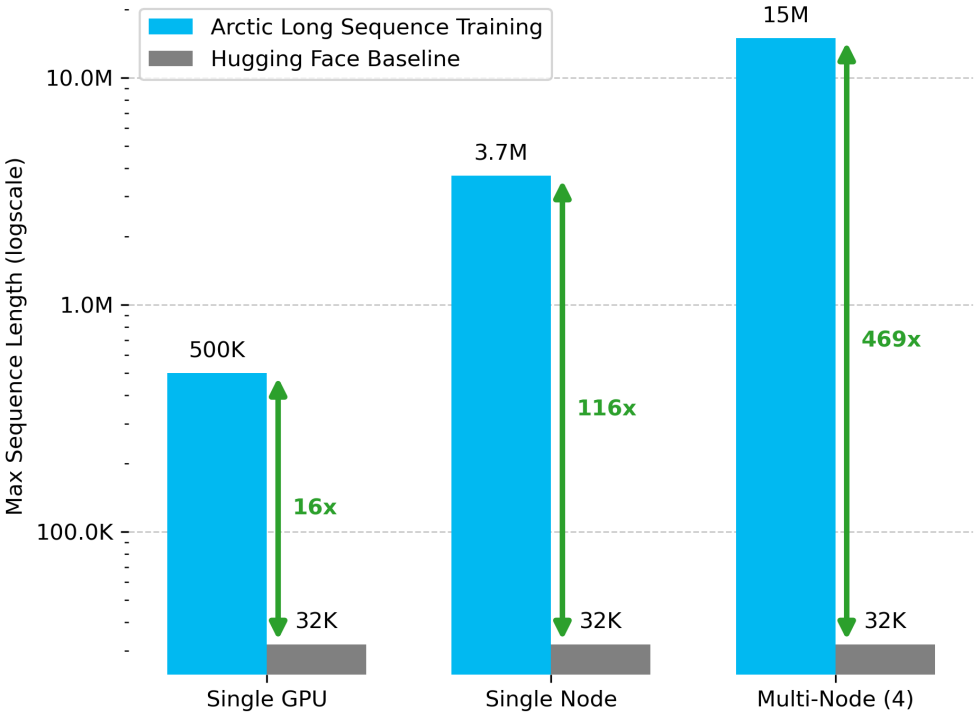

Figure 12: The impact of enabling ALST for LLama-8B on 1, 8 and 32 GPUs

Please note that sequence length is on the log scale because otherwise the baseline wasn't showing in the 32 GPU plot.

5.6 Training Correctness

We used Llama-8B to validate that ALST matches the baseline on training quality. We compared a 32k sequence length on a single 8x H100 GPUs node.

In order to ensure equal conditions, in the case of ALST, since we use 8 GPUs to process a single sample there, we enabled gradient accumulation steps of 8. That way each iteration in both setups has seen the exact same data.

As can be observed from Figure 13 we have an almost exact match for the loss with ALST [25]. Thus we know ALST provides the same training quality as the baseline.

[25]The two plots overlap almost exactly and the tiny differences can only be seen from checking the exact loss values for each iteration in the floating box with loss values per plot.

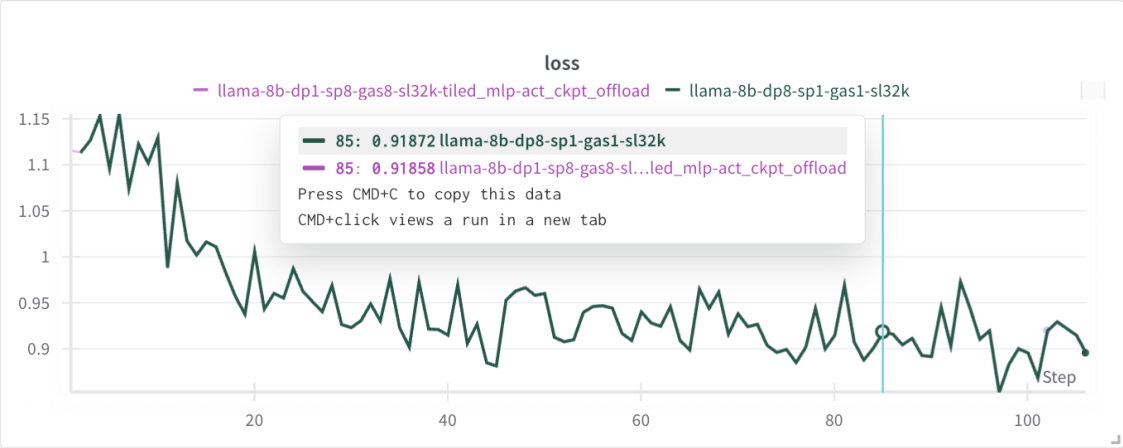

Figure 13: Training loss comparison with and without ALST

6 Trying it out

The ArcticTraining framework has fully working post-training recipes using ALST for a variety of models and quantities of GPUs. You can just drop in your dataset definition and run the post-training.

Go to https://github.com/snowflakedb/ArcticTraining/blob/main/projects/sequence-parallelism/README.md and follow the instructions there to reproduce any of the evaluations presented in this paper or adapt the existing recipes to your long sequence length finetuning needs.

7 Additional Notes for Users

7.1 Limitations

- Currently the maximum degree of sequence parallelism (SP) is limited by the number of q_heads. For example, meta-llama/Llama-3.1-70B-Instruct has 64 q_heads, so SP=64 is the maximum possible with that model. We plan to remove that limit in future work. Meanwhile you can still scale beyond the SP limit imposed by head count, while using a higher DP degree. For example, you can easily train on 1024 GPUs, there will be 16 SP replicas of SP=64 each.
- As discussed earlier q_heads need to be divisible by SP degree. For example, if the model has 9 q_heads, you'd need SP to be 1, 3 or 9. We plan to overcome this limitation in the future.

7.2 Important Training Notes

While this technology enables you to train on very long sequences, you need to be aware that if you pack many short sequences into a long one it won't learn to infer on long sequences. You need to use a dataset with samples whose sequence length matches your target goal. If you train on packed samples, it'd be akin to having a large batch size of short sequence length samples.

Because the dense attention mechanism has a quadratic O(s^2) relationship with the sequence length - the longer the individual sample, the slower the attention calculation will be. As of this paper's writing the incarnation of Ulysses SP for HF supports both SDPA and Flash Attention 2 (FA2) as they are integrated into Hugging Face Transformers. FA2 is very efficient at calculating the attention of individual samples packed into a long sequence by using position ids, whereas SDPA in Hugging Face Transformers as of this writing ignores position ids and ends up attending to the whole packed sequence which is both much slower and isn't correct. Though as explained earlier, to post-train your model for long sequence lengths you have to use actual long sequence length samples and not packed short samples, in which case both SDPA and FA2 will work correctly.

8 Future Work

While large matrix multiplications dominate training with very long sequence lengths, making other operations quite insignificant performance-wise, additional work can be done to further improve the performance of various components where they don't overlap with compute.

While the initial implementation has been integrated into Arctic Training - we next would like to integrate it into Hugging Face Accelerate and Trainer and various other frameworks to make it easy for any user to access this technology. The integration document can be found here.

Acknowledgments

Besides the paper's authors the following folks have contributed to this work and we would like to thank them. The Hugging Face team: Cyril Vallez, Yih-Dar Shieh and Arthur Zucker. The PyTorch team: Jeffrey Wan, Mark Saroufim, Will Constable, Natalia Gimelshein, Ke Wen and alband. This paper's reviewers: Ari Rean and Anupam Datta. Also, we would like to acknowledge the original Ulysses for Megatron-DeepSpeed team: Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari and Yuxiong He.

References

[1] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," 2023. [Online]. Available: https://arxiv.org/abs/2310.01889

[2] S. A. Jacobs, M. Tanaka, C. Zhang, M. Zhang, S. L. Song, S. Rajbhandari, and Y. He, "Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models," *arXiv preprint arXiv:2309.14509*, 2023. [Online]. Available: https://arxiv.org/abs/2309.14509

[3] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," 2022. [Online]. Available: https://arxiv.org/abs/2205.05198

[4] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," 2022. [Online]. Available: https://arxiv.org/abs/2105.13120

[5] D. Li, R. Shao, A. Xie, E. P. Xing, X. Ma, I. Stoica, J. E. Gonzalez, and H. Zhang, "Distflashattn: Distributed memory-efficient attention for long-context llms training," 2024. [Online]. Available: https://arxiv.org/abs/2310.03294

[6] W. Brandon, A. Nrusimha, K. Qian, Z. Ankner, T. Jin, Z. Song, and J. Ragan-Kelley, "Striped attention: Faster ring attention for causal transformers," 2023. [Online]. Available: https://arxiv.org/abs/2311.09431

[7] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," 2023. [Online]. Available: https://arxiv.org/abs/2307.08691

[8] J. Fang and S. Zhao, "Usp: A unified sequence parallelism approach for long context generative ai," 2024. [Online]. Available: https://arxiv.org/abs/2405.07719

[9] D. Gu, P. Sun, Q. Hu, T. Huang, X. Chen, Y. Xiong, G. Wang, Q. Chen, S. Zhao, J. Fang, Y. Wen, T. Zhang, X. Jin, and X. Liu, "Loongtrain: Efficient training of long-sequence llms with head-context parallelism," 2024. [Online]. Available: https://arxiv.org/abs/2406.18485

[10] P.-L. Hsu, Y. Dai, V. Kothapalli, Q. Song, S. Tang, S. Zhu, S. Shimizu, S. Sahni, H. Ning, and Y. Chen, "Liger kernel: Efficient triton kernels for llm training," 2025. [Online]. Available: https://arxiv.org/abs/2410.10989

[11] Q. Lhoest, A. Villanova del Moral, Y. Jernite, A. Thakur, P. von Platen, S. Patil, J. Chaumond, M. Drame, J. Plu, L. Tunstall, J. Davison, M. Šaško, G. Chhablani, B. Malik, S. Brandeis, T. Le Scao, V. Sanh, C. Xu, N. Patry, A. McMillan-Major, P. Schmid, S. Gugger, C. Delangue, T. Matussière, L. Debut, S. Bekman, P. Cistac, T. Goehringer, V. Mustar, F. Lagunas, A. Rush, and T. Wolf, "Datasets: A community library for natural language processing," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 175–184. [Online]. Available: https://aclanthology.org/2021.emnlp-demo.21

[12] Snowflake AI Research, "ArcticTraining: Simplifying and accelerating post-training for large language models," https://github.com/snowflakedb/ArcticTraining, 2025, version `v0.0.4` (released June 3, 2025); Apache-2.0 license.

[13] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference*

on Knowledge Discovery & Data Mining, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: https://doi.org/10.1145/3394486.3406703

[14] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[15] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," 2020. [Online]. Available: https://arxiv.org/abs/1910.02054

[16] B. Workshop, :, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, C. Raffel, A. Gokaslan, A. Simhi, A. Soroa, A. F. Aji, A. Alfassy, A. Rogers, A. K. Nitzav, C. Xu, C. Mou, C. Emezue, C. Klamm, C. Leong, D. van Strien, D. I. Adelani, D. Radev, E. G. Ponferrada, E. Levkovizh, E. Kim, E. B. Natan, F. D. Toni, G. Dupont, G. Kruszewski, G. Pistilli, H. Elsahar, H. Benyamina, H. Tran, I. Yu, I. Abdulmumin, I. Johnson, I. Gonzalez-Dios, J. de la Rosa, J. Chim, J. Dodge, J. Zhu, J. Chang, J. Frohberg, J. Tobing, J. Bhattacharjee, K. Almubarak, K. Chen, K. Lo, L. V. Werra, L. Weber, L. Phan, L. B. allal, L. Tanguy, M. Dey, M. R. Muñoz, M. Masoud, M. Grandury, M. Šaško, M. Huang, M. Coavoux, M. Singh, M. T.-J. Jiang, M. C. Vu, M. A. Jauhar, M. Ghaleb, N. Subramani, N. Kassner, N. Khamis, O. Nguyen, O. Espejel, O. de Gibert, P. Villegas, P. Henderson, P. Colombo, P. Amuok, Q. Lhoest, R. Harliman, R. Bommasani, R. L. López, R. Ribeiro, S. Osei, S. Pyysalo, S. Nagel, S. Bose, S. H. Muhammad, S. Sharma, S. Longpre, S. Nikpoor, S. Silberberg, S. Pai, S. Zink, T. T. Torrent, T. Schick, T. Thrush, V. Danchev, V. Nikoulina, V. Laippala, V. Lepercq, V. Prabhu, Z. Alyafeai, Z. Talat, A. Raja, B. Heinzerling, C. Si, D. E. Taşar, E. Salesky, S. J. Mielke, W. Y. Lee, A. Sharma, A. Santilli, A. Chaffin, A. Stiegler, D. Datta, E. Szczechla, G. Chhablani, H. Wang, H. Pandey, H. Strobelt, J. A. Fries, J. Rozen, L. Gao, L. Sutawika, M. S. Bari, M. S. Al-shaibani, M. Manica, N. Nayak, R. Teehan, S. Albanie, S. Shen, S. Ben-David, S. H. Bach, T. Kim, T. Bers, T. Fevry, T. Neeraj, U. Thakker, V. Raunak, X. Tang, Z.-X. Yong, Z. Sun, S. Brody, Y. Uri, H. Tojarieh, A. Roberts, H. W. Chung, J. Tae, J. Phang, O. Press, C. Li, D. Narayanan, H. Bourfoune, J. Casper, J. Rasley, M. Ryabinin, M. Mishra, M. Zhang, M. Shoeybi, M. Peyrounette, N. Patry, N. Tazi, O. Sanseviero, P. von Platen, P. Cornette, P. F. Lavallée, R. Lacroix, S. Rajbhandari, S. Gandhi, S. Smith, S. Requena, S. Patil, T. Dettmers, A. Baruwa, A. Singh, A. Cheveleva, A.-L. Ligozat, A. Subramonian, A. Névéol, C. Lovering, D. Garrette, D. Tunuguntla, E. Reiter, E. Taktasheva, E. Voloshina, E. Bogdanov, G. I. Winata, H. Schoelkopf, J.-C. Kalo, J. Novikova, J. Z. Forde, J. Clive, J. Kasai, K. Kawamura, L. Hazan, M. Carpuat, M. Clinciu, N. Kim, N. Cheng, O. Serikov, O. Antverg, O. van der Wal, R. Zhang, R. Zhang, S. Gehrmann, S. Mirkin, S. Pais, T. Shavrina, T. Scialom, T. Yun, T. Limisiewicz, V. Rieser, V. Protasov, V. Mikhailov, Y. Pruksachatkun, Y. Belinkov, Z. Bamberger, Z. Kasner, A. Rueda, A. Pestana, A. Feizpour, A. Khan, A. Faranak, A. Santos, A. Hevia, A. Unldreaj, A. Aghagol, A. Abdollahi, A. Tammour, A. HajiHosseini, B. Behroozi, B. Ajibade, B. Saxena, C. M. Ferrandis, D. McDuff, D. Contractor, D. Lansky, D. David, D. Kiela, D. A. Nguyen, E. Tan, E. Baylor, E. Ozoani, F. Mirza, F. Ononiwu, H. Rezanejad, H. Jones, I. Bhattacharya, I. Solaiman, I. Sedenko, I. Nejadgholi, J. Passmore, J. Seltzer, J. B. Sanz, L. Dutra, M. Samagaio, M. Elbadri, M. Mieskes, M. Gerchick, M. Akinlolu, M. McKenna, M. Qiu, M. Ghauri, M. Burynok, N. Abrar, N. Rajani, N. Elkott, N. Fahmy, O. Samuel, R. An, R. Kromann, R. Hao, S. Alizadeh, S. Shubber, S. Wang, S. Roy, S. Viguier, T. Le, T. Oyebade, T. Le, Y. Yang, Z. Nguyen, A. R. Kashyap, A. Palasciano, A. Callahan, A. Shukla, A. Miranda-Escalada, A. Singh, B. Beilharz, B. Wang, C. Brito, C. Zhou, C. Jain, C. Xu, C. Fourrier, D. L. Periñán, D. Molano, D. Yu, E. Manjavacas, F. Barth, F. Fuhrimann, G. Altay, G. Bayrak, G. Burns, H. U. Vrabec, I. Bello, I. Dash, J. Kang, J. Giorgi, J. Golde, J. D. Posada, K. R. Sivaraman, L. Bulchandani, L. Liu, L. Shinzato, M. H. de Bykhovetz, M. Takeuchi, M. Pàmies, M. A. Castillo, M. Nezhurina, M. Sänger, M. Samwald, M. Cullan, M. Weinberg, M. D. Wolf, M. Mihaljcic, M. Liu, M. Freidank, M. Kang, N. Seelam, N. Dahlberg, N. M. Broad, N. Muellner, P. Fung, P. Haller, R. Chandrasekhar, R. Eisenberg, R. Martin, R. Canalli, R. Su, R. Su, S. Cahyawijaya, S. Garda, S. S. Deshmukh, S. Mishra, S. Kiblawi, S. Ott, S. Sang-aroonsiri, S. Kumar, S. Schweter, S. Bharati, T. Laud, T. Gigant, T. Kainuma, W. Kusa, Y. Labrak, Y. S. Bajaj, Y. Venkatraman, Y. Xu, Y. Xu, Y. Xu, Z. Tan, Z. Xie, Z. Ye, M. Bras, Y. Belkada, and T. Wolf, "Bloom: A 176b-parameter open-access multilingual language model," 2023. [Online]. Available: https://arxiv.org/abs/2211.05100

[17] Q. Anthony, J. Hatef, D. Narayanan, S. Biderman, S. Bekman, J. Yin, A. Shafi, H. Subramoni, and D. Panda, "The case for co-designing model architectures with hardware," 2024. [Online]. Available: https://arxiv.org/abs/2401.14489