

Reproducible MFU optimization techniques to boost your training efficiency

Lambda's optimizations consistently drive >25% efficiency gains¹ for foundation model training on NVIDIA HGX B200 and GB300 NVL72.

Chelsea Lowman | Caia Costello | DJ Matusz | Chuan Li | Ravali Reddy | Nick Harvey | Niranjana Hira | Robin Singh | Paul Zhao | Long Fei

Executive summary

In this paper, we demonstrate that co-optimized hardware configuration and training stack can substantially increase Model FLOPS Utilization (MFU) on NVIDIA HGX B200 and NVIDIA GB300 NVL72 platforms. Across 10 representative workloads, we observe >25% MFU uplift, with a median improvement of approximately 35% over the industry baseline.

Results span 8-128 NVIDIA HGX B200 and NVIDIA GB300 NVL72 GPUs, 8B-405B parameter models, and 8k-32k sequence lengths, and are achieved without changes to model architecture or datasets. The efficiency gains were driven by Lambda's ML experts:

- Utilizing open source training stacks and extrapolating configuration best practices from previous-generation GPUs.
- Engineering high-performance infrastructure:
 - + NVIDIA HGX B200—400 Gb/s NVIDIA Quantum-2 InfiniBand connectivity per GPU for cross-node communication, and fifth-generation NVIDIA NVLink providing 1.8 TB/s of GPU-to-GPU bandwidth for intra-node communication.
 - + NVIDIA GB300 NVL72 system—800 Gb/s per GPU NVIDIA Quantum-X800 InfiniBand for cross-rack communication, and 130 TB/s total NVLink bandwidth per GPU across the rack-wide NVLink fabric.

Together, these results demonstrate that system-level optimization can unlock a substantial fraction of otherwise underutilized accelerator performance, even for mature model architectures.

The efficiency challenge: Bridging the gap between peak and actual utilization

Training foundation models on large clusters spans many layers of software and hardware, all working together, and each layer has opportunities for optimization. Even small performance improvements lead to better hardware utilization, faster training times, and faster evolution of the state-of-the-art models.

¹ A 40% MFU industry baseline was chosen to reflect the median of industry consensus, 35-45%. Workload improvements ranged from 25% to 50%, with a mathematical median of 36% above the median industry consensus baseline.

To measure the extent of optimizations, the ML community at large uses Model FLOPS Utilization (MFU) as the metric to optimize training workloads. MFU is defined as how close a training run comes to theoretical maximum performance:

$$MFU = \text{Observed FLOPS} / \text{Peak Theoretical FLOPS}$$

MFU is very easy to compute. Peak Theoretical FLOPS is a static value retrieved from open GPU datasheets, and the Observed FLOPS is computed using throughput:

$$\text{Observed FLOPS} = (\text{FLOP/token})(\text{tokens/second})$$

Notably, the *FLOP/token* is model-dependent value indicating how many floating point operations are involved in one training step for your model, or simply, “how big your model is.” Dense models typically use the following formula:

$$FLOP/token = 6(N - N_{emb}) + 12 LHQS$$

See our appendix for an explanation of these formulas and concrete examples.

In practice, MFU for production-scale LLM training typically ranges between 35–45%^{2,3,4}. As each NVIDIA GPU architecture GPUs offer more and more compute power with higher peak theoretical capability, optimizing LLM workloads to maximize GPU utilization has become a critical problem for developers to solve.

Low MFU rarely has a single root cause. For example, memory pressure forces small per-GPU batches and heavy activation checkpointing. High-degree tensor parallelism slices weights into several small shards, resulting in expensive cross-GPU communication.

A key challenge is that optimization guidance is often generic (“turn on Fully Sharded Data Parallel (FSDP)”, “use mixed precision”) and hard to translate into concrete, reproducible changes for a specific stack. Similarly, benchmark claims from providers are often presented as black boxes, without configuration guidance or methodologies, making them difficult to adapt to your workload.

To address this gap, we start from a well-known workload with a clean open-source baseline and measure how far MFU can be pushed using a set of [targeted, reproducible optimizations](#). Experiments are conducted on Lambda’s NVIDIA HGX B200 and NVIDIA GB300 NVL72 clusters. The goal is not to introduce a novel training stack, but to document practical optimizations that materially improve training performance in open-source libraries and generalize to other large-scale training efforts. That said, further optimization is still possible with additional levers. One of our partners, Ceramic, was able to push MFU even higher on Lambda infrastructure using their proprietary training stack. Their results are highlighted on our blog [“Ceramic Training Infrastructure Delivers Superior Performance on Lambda’s NVIDIA HGX B200”](#).

² [The Llama 3 Herd of Models](#) reports 38–43% MFU during Llama 405B pretraining

³ [PaLM: Scaling Language Modeling with Pathways](#) reports 46% MFU when training PaLM

⁴ [Megatron-LM](#) reports up to 47% MFU

MFU on NVIDIA HGX B200 and NVIDIA GB300 NVL72

Our goal is to measure improvements in MFU that are directly attributable to Lambda's training stack and optimizations, starting from a clean baseline. In order to fully exercise the hardware and optimization processes, we chose three standard Llama 3.1 models and trained them across Lambda's NVIDIA HGX B200 and NVIDIA GB300 NVL72 clusters:

- Llama 3.1 8B on 8x NVIDIA HGX B200
- Llama 3.1 70B on 16x NVIDIA HGX B200
- Llama 3.1 405B on 2x NVIDIA GB300 NVL72

For each model, we optimized multiple context lengths, as both short and long contexts are essential to training high-quality LLMs:

- 8k tokens (8,192)
- 16k tokens (16,384)
- 32k tokens (32,768)

This setup allows us to optimize both memory and compute-bound workloads, as well as workloads that run across nodes and racks.

All evaluated models are dense, decoder-only transformers with full attention. Despite being relatively mature and MFU-friendly architectures, the efficiency gains we achieve highlight a broader opportunity: if substantial optimization headroom remains for dense models, how much MFU remains untapped in more complex sparse architectures, including Mixture-of-Experts (MoE) models?

Training stack

All experiments use a consistent training stack to ensure comparability across runs:

- Framework: TorchTitan
- Precision: BF16 and BF16 + FP8
- Dataset: [AllenAI C4](#), about 300GB of common crawl web data, streamed from Hugging Face

Baseline (industry standard and TorchTitan)

To ensure fair, reproducible comparisons, all performance comparisons start from an identical, controlled baseline to isolate the effects of batch size, model size, and optimization strategies on performance.

Our baseline is both the industry standard 35-45% (baseline 40%, as median of the range) and the [torchtitan@e7ee95a](#) codebase with the following rubric:

- Out-of-the-box TorchTitan configs
- Global batch size increased only to saturate GPU RAM
- NVIDIA cuDNN attention
- No v-boost

This setup ensures that Lambda's results are on the same hardware, with the same models and data. The only variable is the code changes and training recipes: the set of optimizations across memory, parallelism, kernel fusion, and runtime tuning that determine how efficiently model X runs on GPU Y at scale Z.

Results

Across all evaluated configurations, our results consistently improve MFU relative to industry standard and TorchTitan baselines. Compared to industry standards, we achieve a **median MFU improvement of 1.36x** across all workloads. This improvement is consistent across sequence lengths, with MFU reaching **up to 55% at 8k** pretraining and increasing to **up to 60% at 16k–32k**.

The table below summarizes MFU throughput improvements at an 8k sequence length, specifically comparing the TorchTitan baseline (optimized for Ampere or Hopper systems) against TorchTitan with Lambda optimizations on NVIDIA Blackwell systems.

	TorchTitan baseline MFU (8k sequence length)	TorchTitan with Lambda optimizations MFU (8k seq length)	Lambda Uplift vs. TorchTitan baseline MFU
Llama-3.1-8b BF16 8x NVIDIA HGX B200	44.46%	55.34%	1.24x
Llama-3.1-70b BF16 16x NVIDIA HGX B200	23.83%	50.20%	2.11x
Llama-3.1-405b BF16 128x NVIDIA GB300 NVL72	28.62%	52.69%	1.84x

Two trends are immediately apparent:

1. Larger models benefit disproportionately from optimization.

The 70B configuration shows more than a 2x increase in MFU, reflecting how communication overhead, memory pressure, and suboptimal parallelism strategies compound at scale.

2. Baseline MFU degrades rapidly with scale.

Without careful tuning, increasing model size and cluster scale leads to underutilization driven by smaller per-GPU workloads and increased synchronization costs.

Our MFU uplift results are particularly significant given the large model parameter sizes tested, which increase workload complexity and make achieving MFU gains materially more challenging due to communication and memory overhead.

Scaling with sequence length

As sequence length increases, activation memory, attention cost, and communication volume also increase. Despite these pressures, optimized configurations maintain higher MFU across all evaluated sequence lengths:

- At 16K and 32K tokens, MFU continues to scale upward rather than collapsing under memory pressure.
- At longer sequence lengths, MFU approaches ~60%, indicating effective overlap of computation, communication, and memory access.

This behavior demonstrates that MFU improvements are not limited to short-sequence pretraining stages but extend to long-sequence regimes, which are increasingly important for modern LLM training.

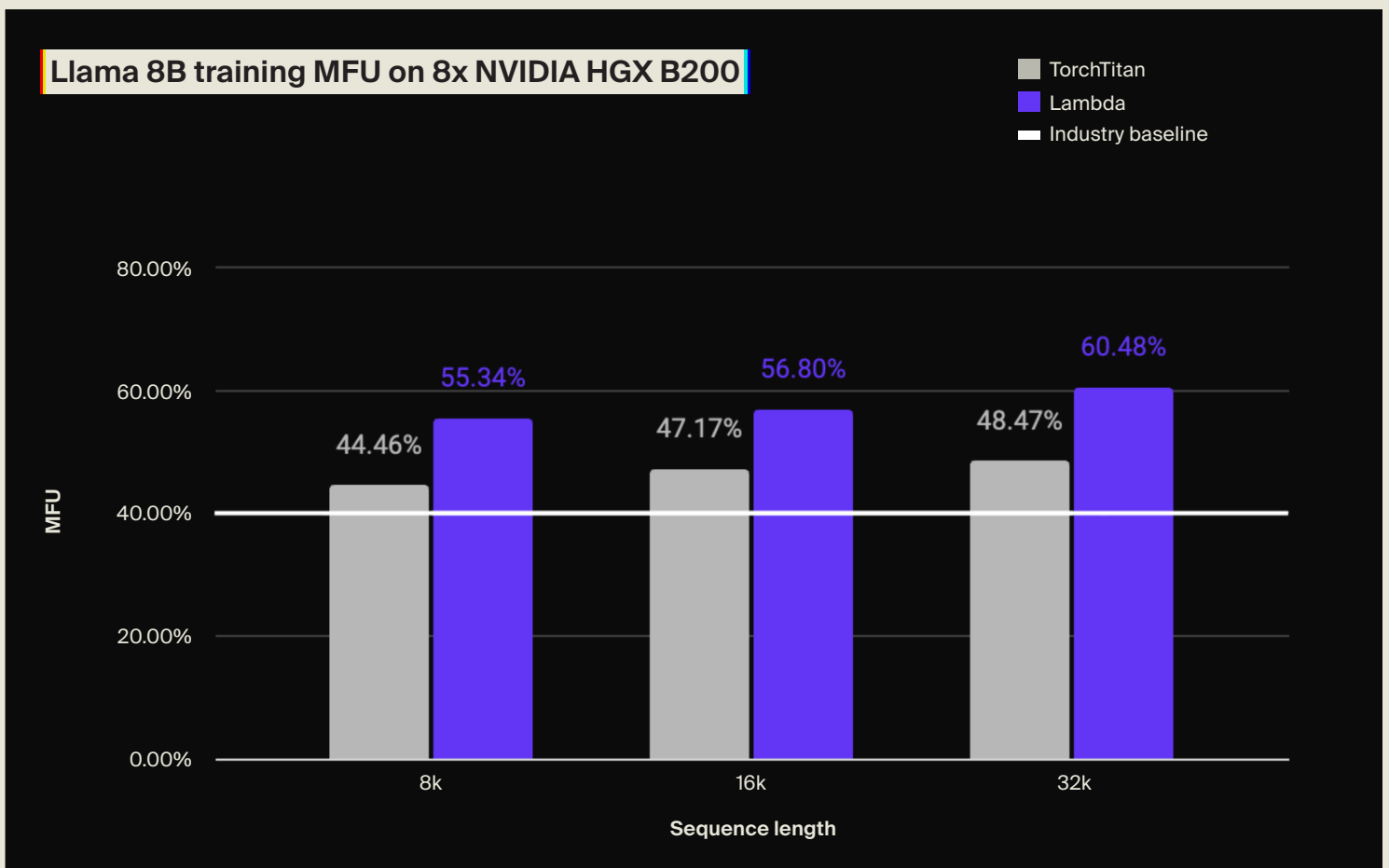


Figure 1: MFU for the smaller Llama-3.1-8B model on 8 NVIDIA HGX B200 systems, reaching 55–60% depending on sequence length.

We also observed significantly lower MFU (< 30%) when training LLaMA-70B with the default [TorchTitan configuration](#) (Figure 2). This highlights a performance-tuning opportunity: settings optimized for older GPUs (e.g., Ampere/Hopper) may not map well to newer architectures such as NVIDIA Blackwell. In this case, the default configuration targets A100. By reducing tensor parallelism from eight to one and switching activation checkpointing from *full* to *selective*, we leveraged NVIDIA Blackwell’s larger memory, higher NVLink bandwidth, and compute resources, achieving a ~2× MFU improvement.

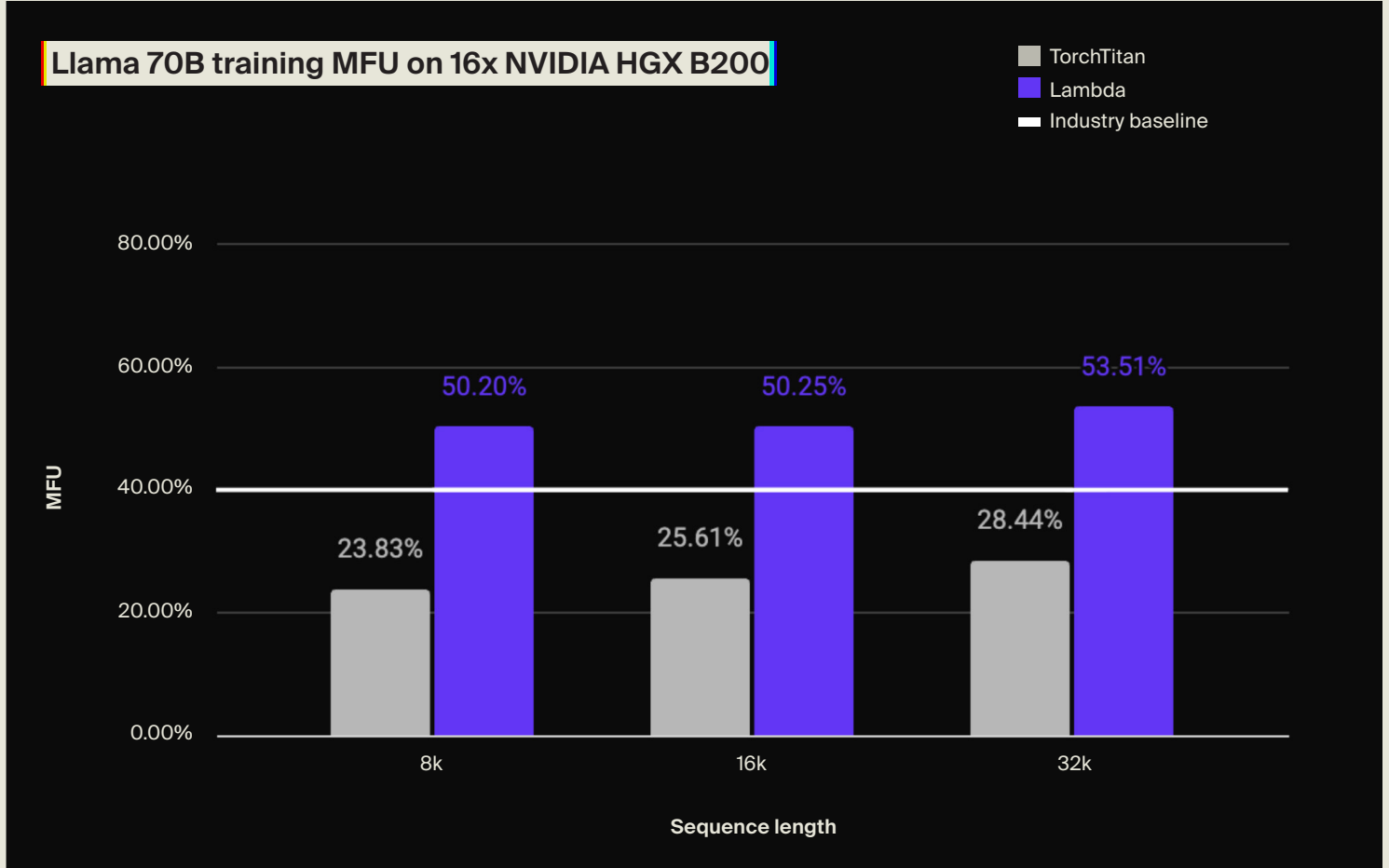


Figure 2: MFU for the mid-size Llama-3.1-70B model on 16 NVIDIA HGX B200 systems (50–53% depending on sequence lengths). The low MFU with the default TorchTitan configuration (gray bars, optimized for Ampere) highlights that settings tuned for older GPUs don't map well to newer architectures.

Training step profiling

We took a profile from a single training step (Chart 1), calculated the time each kernel took to complete, and provided a breakdown of the main kernel calls.

Takeaways:

- Matrix multiplication increased from 56% to 68% of train step time. We attribute this to kernel fusion by torch.compile and the larger batch size.
- Elementwise addition went from 14% of time to 1.5%, as torch.compile fuses these operations into multiplication kernels.
- Attention in the forward pass was optimized from 7% to 3%, while attention in the backward pass increased from 11.5% to 15.4%. Due to a larger batch size, backward is slightly slower.
- Training step time was 2.474 sec vs. 2.615 sec; the higher overall training time is due to a larger batch size (3x), which increases compute utilization by processing more samples per step.

What Llama-3.1-8B MFU improvement looks like in the profile:

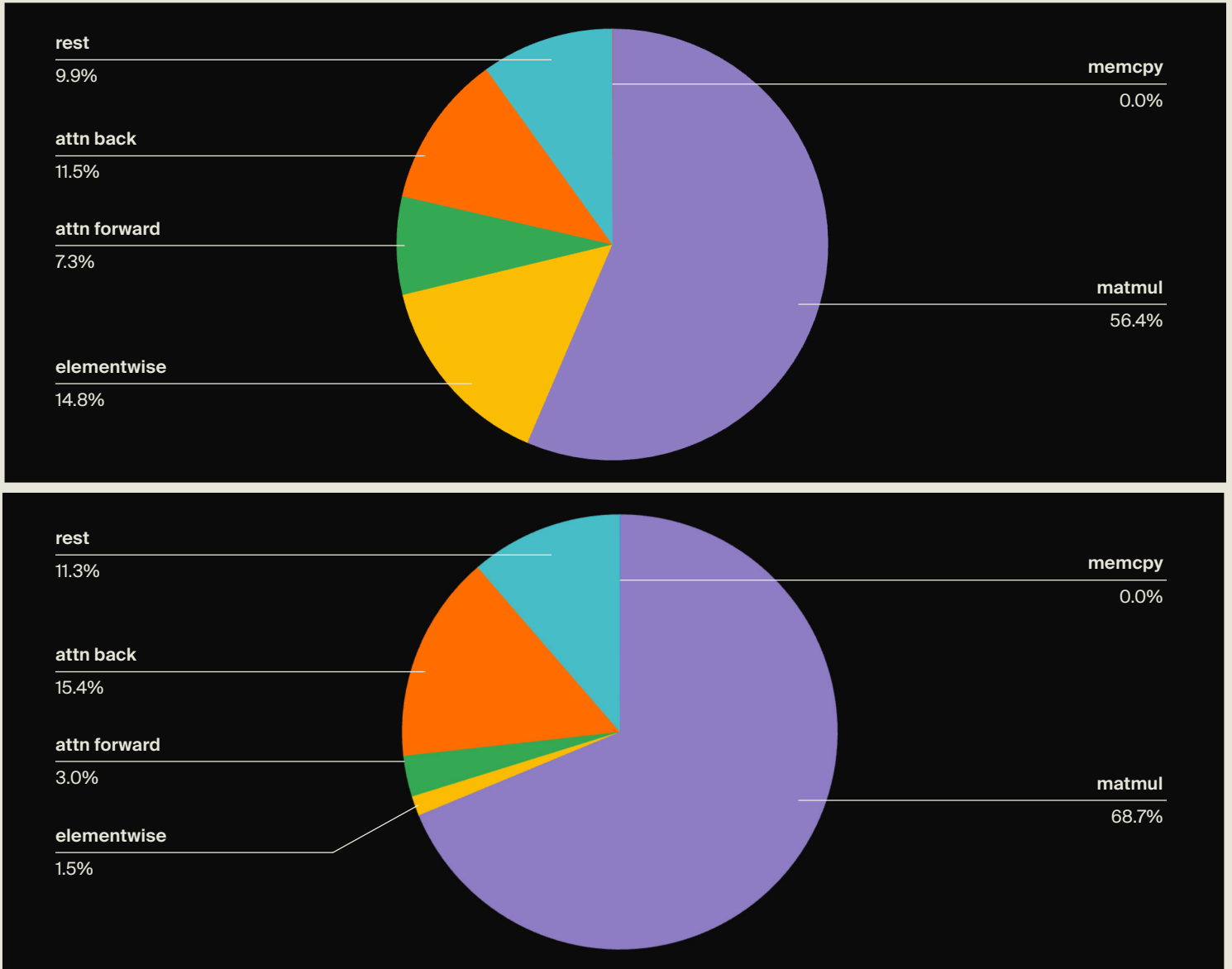


Figure 3: Profile breakdown of MFU for the smaller Llama-3.1-8B model on 8x NVIDIA HGX B200 systems, 8k sequence (baseline top, optimized bottom).

Scaling with GPUs

We also evaluate MFU as a function of cluster size, spanning single-node, multi-node, and multi-rack configurations.

Optimized runs demonstrate:

- Improved scaling efficiency as GPU count increases
- Reduced sensitivity to world size, particularly for large models
- Better utilization of multi-rack configurations, such as NVIDIA GB300 NVL72, where communication topology plays a dominant role

These results highlight the importance of aligning parallelism strategies (FSDP, TP, HSDP) with hardware interconnects rather than relying on default configurations.

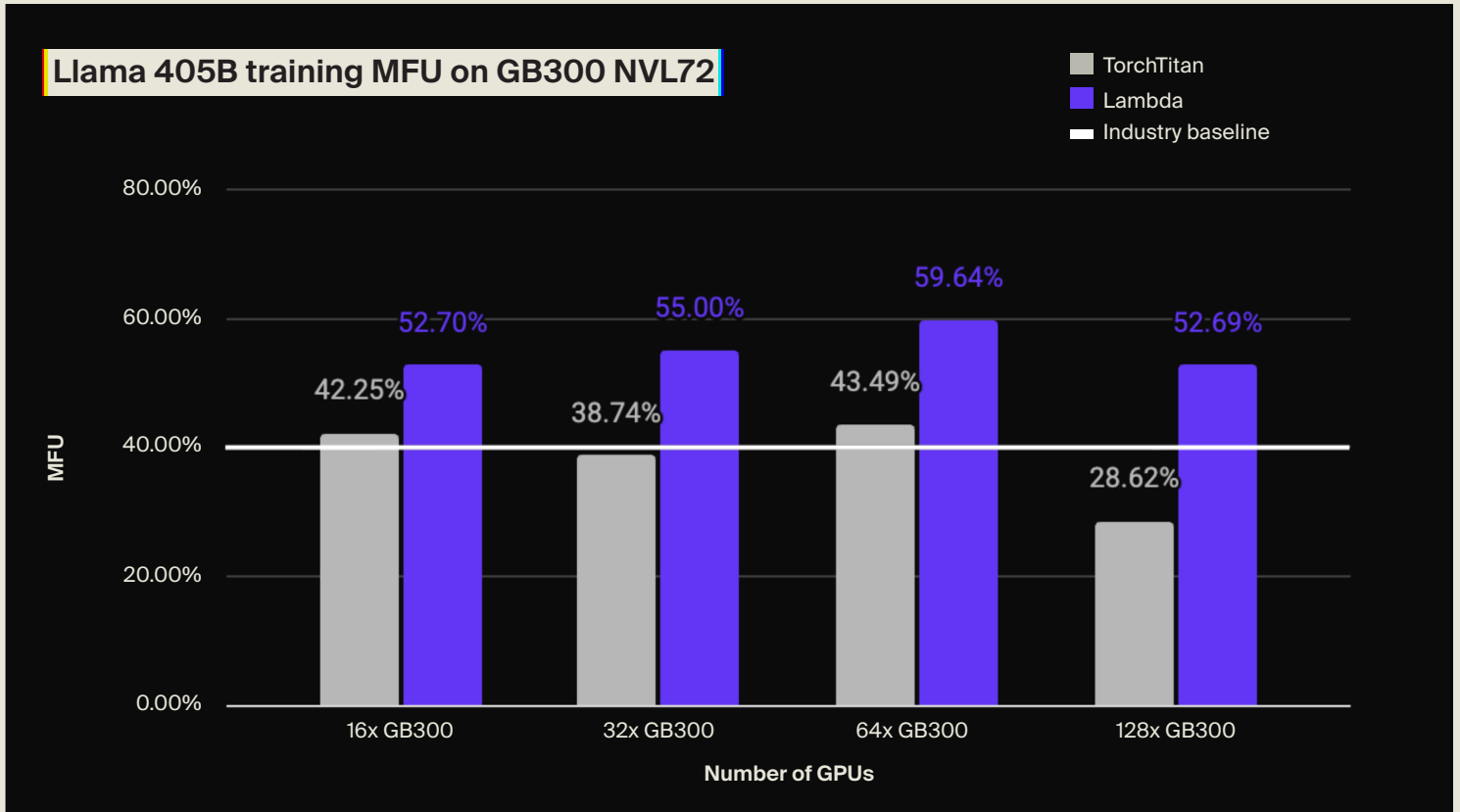


Figure 4: MFU for the large LLaMA 405B model on an NVIDIA GB300 NVL72 system with sequence length fixed at 8K, reaching 52–56% MFU depending on GPU count.

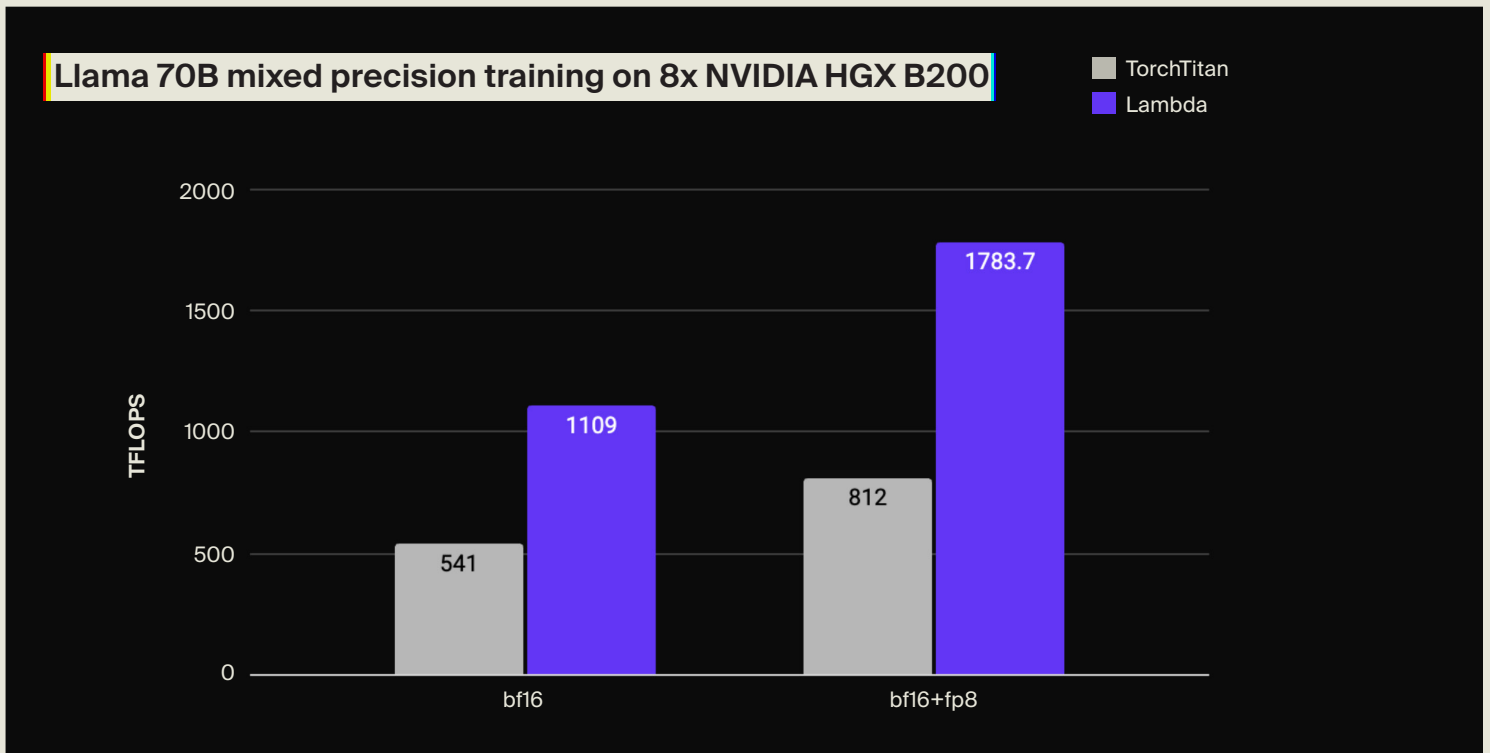


Figure 5: Mixed precision training: float8 + bfloat16

For mixed-precision performance, we use TFLOPS to measure performance, as MFU doesn't have a commonly adopted definition for workloads that span mixed precisions. Normally, MFU is TFLOPS normalized by the theoretical peak FLOPS for a specific data type (e.g., bfloat16), but with multiple data types, this normalization is less well defined. With targeted mixed-precision optimizations, including reducing gradient communication precision to lower memory pressure and overlapping FP8 scale computation with the backward pass, we achieve approximately 2x higher TFLOPS (1783 vs. 812) per node for BF16+FP8 training compared to the default TorchTitan configuration.

How to optimize your MFU

We applied a set of targeted optimizations designed to keep the GPU as busy as possible. These techniques address suboptimal software and hardware configurations that commonly limit MFU in large-scale training.

The gains in our results come from four main areas:

1. Kernel fusion
2. Memory optimizations
3. Communication/computation overlap
4. System-level configuration

1. Kernel fusion

1.1 Merging the QKV computation

In the standard transformer attention layer, there are three linear layers that run before the actual attention algorithm: the query, key, and value projections. These all use the same input data, so a straightforward optimization is to merge them into a single linear layer and then split the output. Running a single matrix multiplication kernel can be more computationally efficient than running three separate ones.

1.2 Fused optimizer kernels

Enabling fused optimizer kernels combines the optimizer's update computations into a single GPU kernel, reducing kernel-launch overhead and global-memory traffic.

1.3 torch.compile for model and loss functions

Compiling the model using `torch.compile` is now standard practice, and compiling the loss function also saves an appreciable amount of memory from operator fusion.

2. Memory optimizations

Memory constraints are one of the most common causes of low MFU. When memory is tight, batch sizes shrink, activation checkpointing increases, and parallelism strategies become more complex, often reducing effective compute utilization.

Our general approach is:

1. Prefer no AC (Activation checkpointing, prefer no TP/CP/PP (Tensor parallelism, Context parallelism, Pipeline parallelism), maximize data parallelism)
2. Reach for selective AC and then full AC first when running out of memory
3. Add TP if full AC still results in insufficient memory
4. Add CP if full AC + TP is still having memory problems

2.1 Compute and communication in bfloat16

By default, TorchTitan keeps the master (optimizer) weights in float32 to ensure stable parameter updates. When `dtype="bfloat16"` is set, forward and backward passes run in bfloat16. Additionally, setting `mixed_precision_reduce="bfloat16"` casts gradients to bfloat16 before the all-reduce collective, reducing communication bandwidth during distributed training. After the all-reduce, gradients are cast back before the optimizer step. The optimizer still retains a float32 copy of the model parameters and states to maintain numerical stability throughout.

2.2 Overlap optimizer with the backward pass

This frees activations and gradients early in the backward pass by triggering the optimizer's backward pass as soon as gradients are available. See the [PyTorch blog post](#) for greater detail.

2.3 FSDP mixed precision reduction dtype

By default, TorchTitan performs gradient reductions in float32, but setting the [reduction dtype to bfloat16](#) frees up quite a bit of memory. See [MixedPrecision::reduce_dtype](#) for more details.

2.4 Activation checkpointing (AC)

Activation checkpointing is a technique that reduces memory usage by discarding intermediate activations needed for the backwards pass and recomputing them on the fly during the backwards pass. Applying AC reduces memory usage but increases computation, and so **AC generally reduces MFU**. Note that AC is not considered in MFU calculations by design; the increase in FLOPS introduced by recomputing activations is not added to your model's total FLOPS.

There are two typical forms of AC: selective (SAC) and full (FAC). SAC applies AC to specific operations, such as softmax or attention, whereas FAC typically applies AC at the decoder layer level. SAC typically reduces memory less than FAC but requires much less recomputation.

When using AC, [ensure early stopping is enabled](#), as it allows recomputation to stop early once all necessary activations have been computed. This is disabled by default in TorchTitan, which isn't optimal for large models.

2.5 Tensor parallelism (TP)

Tensor parallelism reduces memory usage for linear layers by splitting matrices across GPUs. It reduces the size of the weights and, therefore, the size of the intermediate activations.

TP generally reduces MFU; however, there is an exception for large clusters, as detailed below. It introduces communication overhead and reduces per-device workload sizes. As a result, TP should be treated as a last resort

A key guideline is to apply TP only to GPUs that are connected to NVSwitch. This is typically done within a single node, but on rack-scale systems like the NVIDIA GB300 NVL72, all 72 GPUs are connected with NVSwitch.

2.5.1 Asynchronous TP

[Async TP](#) is a recent feature introduced by PyTorch and TorchTitan that uses symmetric device memory. It enables computation and communication overlap to achieve higher scaling efficiency. This generally improves performance and should be enabled when using TP.

2.5.2 TP on large clusters

On large clusters, FSDP's all-gather can negatively impact performance because it's an N² operation across all GPUs. In these cases, moderate TP (e.g., TP=8) can improve performance by reducing the world size by a factor of 8, thereby lowering the cost of doing all-gather operations.

2.6 Context parallelism (CP)

[Context parallelism](#) splits input sequences across GPUs to reduce per-GPU activation memory. While useful in extreme memory-constrained scenarios, CP introduces higher synchronization and communication overhead than TP and should be applied cautiously.

2.7 Hybrid sharded data parallelism (HSDP)

HSDP generalizes FSDP by partitioning the world into subsets, each maintaining a full model replica. Within each subset, FSDP splits the entire model across your entire world. Across these replicas, it uses DDP techniques to align gradients, ensuring the model replicas are maintained.

HSDP is especially useful when using 2 NVIDIA GB300 NVL72 – you can have one model replica per rack, with FSDP applied within each rack. This lets the expensive all-gathers operate within the racks to take advantage of NVSwitch, while the gradient reductions between racks minimize communication bandwidth.

3. Communication/computation overlap

Even with efficient kernels and memory layouts, MFU suffers when computation and communication serialize. The following optimizations focus on overlapping communication with useful compute wherever possible.

3.1 FSDP prefetching

FSDP works by splitting model weights across GPUs and performs all-gather operations before executing each layer's forward pass. By default, FSDP prefetches a single layer only during this all-gather.

For smaller models, [you can configure prefetching](#) to fetch more than a layer forward to achieve a better balance between communication vs. computation. This increases peak memory usage, but can be successfully applied to improve performance, especially with the smaller 8b sized models. Using prefetching was key to achieving MFU above 60% with Llama 8b.

3.2 Pinned memory

Using [pinned memory](#) for data loading is a small technique that minimizes blocking CPU<>GPU transfers during

training. This ensures the CPU always stays ahead of the GPU, and is an easy win for throughput and consistency.

3.3 First-layer prefetching

[Prefetching the first layer of the model](#) before loading the next batch further reduces startup latency at the start of each step. This technique complements pinned memory and is particularly effective when combined with aggressive input pipelining.

3.4 FP8 Scale precomputation

By default, [torchao](#) precomputes FP8 weight scales after the backward pass *finishes*, even if you have configured your optimizer to run *during* the backward pass. This serialization introduces unnecessary idle time.

We optimized this precomputation by running it [during the backward pass](#), alongside the early optimizer. This provides a huge boost in performance for mixed precision (BF16 + FP8) training.

4. System-level configuration

Finally, we tune system behavior so compute and communication overlap rather than serialize.

4.1 CPU affinity

Using [NVIDIA/gpu_affinity](#) can give you a small boost in MFU by ensuring that the CPU process affinity matches the hardware.

4.2 Overclocking the GPU (vboost)

Some NVIDIA GPUs support a frequency boost setting via `nvidia-smi boost-slider`, which overclocks the GPU. Enabling vboost typically provides an additional ~1% MFU improvement at the cost of increased power draw.

4.3 Overclocking the CPU (turboboost)

Enabling CPU turbo boost can further reduce host-side bottlenecks, particularly for input processing and coordination overhead. This setting must be configured at the hardware or BIOS level.

What this means for your training stack

If your large-scale training runs consistently operate below 40% MFU on large models, you're leaving both performance and budget on the table. Meaningful throughput improvements require keeping existing GPUs busy with useful matrix multiplications, rather than recomputation, communication, or bookkeeping.

Once you can reliably measure MFU at the per-GPU level, the path to improvement is mostly about optimizing your configuration: tuning memory and optimizer behavior, simplifying parallelism (FSDP-first instead of high TP by default), and treating checkpointing and FP8 as system-level design choices, not individual feature toggles.

A practical starting checklist for your own stack:

1. Maximize your compute by increasing data parallelism and minimizing things like AC/TP/CP/PP

2. Minimize communication across lower bandwidth links, e.g., use one replica for each rack
3. If your memory isn't all used up, raise your batch size/sequence length
4. `Torch.compile`

These are simple ways to optimize current open source libraries for your own cluster size. Our partners, like [Ceramic](#) have taken this a step further with targeted optimizations and custom code to further utilize NVIDIA Blackwell B200 GPUs! We hope this may inspire you to do the same.

[Reproduce the results](#) and improve your MFU

The optimizations described here are reproducible on your own workloads. You can:

- Test these configurations on your existing clusters
- Work with our in-house ML and systems engineers to adapt them to your codebase and constraints

Talk to our team to dive deeper into MFU tuning on Lambda's GPU clusters.

Appendix: calculating MFU

[MFU](#) is defined as “the ratio of the observed throughput (tokens-per-second) relative to the theoretical maximum throughput of a system operating at peak FLOPS”:

$$MFU = \text{Observed FLOPS} / \text{Peak Theoretical FLOPS}$$

Observed FLOPS

FLOPS (FLoating Point OPerations per Second) is the standard unit of GPU compute throughput. TorchTitan estimates the FLOP per token using the [dense](#) transformer cost formula from [PaLM: Scaling Language Modeling with Pathways](#):

$$FLOP/token = 6(N - N_{emb}) + 12 LHQS$$

Where N is the number of model parameters, N_{emb} is the number of parameters that are part of the embedding layer, L is the number of decoder layers, the number of attention heads is H, the head dimension is Q, and the sequence length is S.

To compute the observed FLOPS, we use the number of tokens per second achieved by the training stack (measured by dividing the number of tokens per training step by the time taken by the step):

$$\text{Observed FLOPS} = (FLOP/token)(tokens/second)$$

Peak Theoretical FLOPS

The [NVIDIA Blackwell datasheet](#) specifies that the NVIDIA HGX B200 system performs 4.5 PFLOPS for INT8 operations. The [NVIDIA Blackwell Ultra datasheet](#) specifies that each NVIDIA GB300 NVL72 rack performs 5 PFLOPS for bfloat16 operations. We use these numbers in the denominator of the MFU calculation.

Example

To put these metrics in context, consider training a Llama 8B model on a single NVIDIA B200 GPU using BF16 precision with sequence length 8K. The FLOP/token value for this Llama 8B model is approximately 57.93 GFLOP, computed using the formula above with $N = 8.03B$, $N_{\text{emb}} = \text{Vocab size} \times \text{Hidden dim} = 128,256 \times 4,096 = 525.3M$, $L = 32$, $H = 32$, $Q = 128$, $S = 8,192$.

If the training run achieves a throughput of 21,111 tokens/second, then the observed TFLOPS is

$57.93 \times 10^9 \text{ FLOP/token} \times 21,111 \text{ tokens/second} \approx 1,223 \text{ TFLOPS}$

The NVIDIA B200 GPU has a theoretical peak of 2,250 TFLOPS for BF16. This gives an MFU of $1,223 / 2,250 = 54.34\%$.

In other words, the optimization techniques described in this paper enable this training run to capture over half of the GPU's theoretical BF16 compute, up from the ~40% industry baseline.