

Pretraining a 21B-Parameter MoE on a Single GH200: A Measured Optimization Path

YOON DONGWOOK

Independent Researcher

powerggesa12@naver.com

2026

Abstract

We pretrained a fine-grained MoE model with 20.99B total and 3.36B active parameters on a single NVIDIA GH200. The entire FP32 training state (238.5 GiB) lives in Grace CPU memory, where the GPU’s optimizer kernels update it directly over NVLink-C2C. Over 13 benchmark measurements, MFU climbed from 21.1% to 50.7%; one of those steps changed the model itself (the expert shape). The gains came from block-level `torch.compile`, a C++ OpenMP kernel for CPU-side gradient accumulation, Triton fused cross-entropy, keeping FFN gate/up outputs in CPU memory instead of recomputing them, and per-tensor FP8. In pretraining on real data, per-GPU MFU went from 36.5% to 48.9–49.1% over 10.9B tokens, or an estimated 47.6–47.8% once synchronization is counted. A second GH200 joined through DiLoCo at 4B tokens, and from then on each GPU trained the full model. Under the 700 W power cap, BF16 dense GEMMs reach only 640–665 TFLOPS (about 66% of peak), which makes it hard to exceed roughly 60% MFU with BF16 alone. We also report what did not work, including Transformer Engine FP8, row-wise FP8, bulk activation offloading, and splitting gradient accumulation between GPU and CPU. MFU is always computed against the BF16 dense peak (989 TFLOPS), even when 47% of the FLOPs run in FP8, so BF16 and FP8 stages can be read on the same scale. We do not propose a new algorithm, and we do not claim a record MFU. Every technique here comes from prior work and is cited where it is used. What the paper offers is a measured account of how we combined them: the optimization path, the failures, and where each piece of state lives. Code and logs are released with it, and we hope the record helps teams building LLMs on a small budget.

1 Introduction

An MoE model runs only a few experts for each token, so it needs less computation per token than a dense model of the same size[1–4]. Its memory footprint, however, follows the total parameter count and does not shrink in the same way. Our model has 20.99B total and 3.36B active parameters: the BF16 weights alone take 39.1 GiB, and the full training state comes to about 278 GiB, far more than the 96 GB on a single GPU.

The original plan was to train on 64 H100s, which turned out to be beyond our budget. We kept the model as designed and moved to a single GH200 with CPU offloading. We also avoided shrinking the training state with methods that can cost accuracy, such as 8-bit optimizer states or BF16 master weights. Some linear layers have run in FP8 since 4B tokens (Section 4.7), and we have not checked how this affects quality.

The GH200 suits this situation well[5, 6]. It pairs a Hopper GPU (96 GB HBM3) with a 72-core Grace CPU (480 GB LPDDR5X) over NVLink-C2C at 450 GB/s per direction, and since GPU kernels can read and write CPU memory directly, the training state can stay on the CPU side[7]. Throughput then comes down to hiding CPU work and data movement behind GPU computation, and to keeping the GPU busy through the many small GEMMs and elementwise kernels of a fine-grained MoE. The hardware imposed its own limit as well: under the cloud

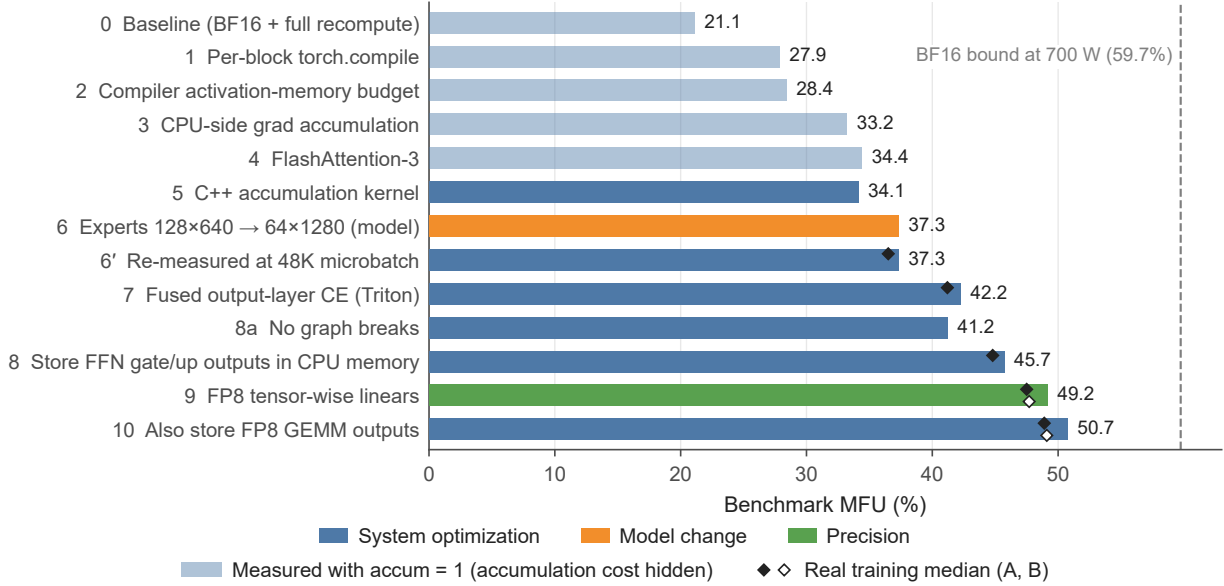


Figure 1: Benchmark MFU of the full 30-layer model after each change (bars) and median real-training MFU on the two GPUs, A and B (diamonds). Bars show `mfu_at_batch`, with the optimizer step amortized over an 8M-token batch (Section 2.2). Step 6 changes the model (expert shape), and 6' is the same model remeasured with the 48K micro-batch used in training. The dashed line marks the BF16 ceiling at 700 W (59.7%, Section 4.5). Each diamond comes from the last interval in which that configuration ran (6': 0–2B, 7: 2–3B, 8: 3–4B, 9: 8–10B, 10: after 10B tokens).

VM's 700 W cap the GPU ran at reduced clocks, and BF16 dense GEMMs reached only 640–665 TFLOPS (Section 4.5).

We report benchmark MFU, measured on synthetic data, separately from real-training MFU, measured over billions of tokens of real data. Training began on one GH200; at 4B tokens a second GH200 joined through DiLoCo[8], and from then on both GPUs trained the full model. Every MFU value in this paper is per GPU. As Figure 1 shows, benchmark MFU rose from 21.1% to 50.7%, while real-training MFU started at 36.5% and reached 48.9–49.1% after 10B tokens (about 47.6–47.8% once synchronization and checkpointing are included). These real-training figures use the conventional FLOPs formula, which ignores document boundaries; counting attention only within documents brings them down by 1.6 percentage points (Section 6.2).

The paper makes the following contributions.

- **State placement on a single superchip** (Section 3). All FP32 state (238.5 GiB) sits in pinned Grace memory, where GPU kernels run the optimizer in place. Gradient accumulation runs on the CPU in a C++ OpenMP kernel, and FFN gate/up outputs are kept in a 72 GiB CPU storage region instead of being recomputed.
- **A measured optimization path** (Section 4). We log 13 benchmark measurements, with kernel-time breakdowns for the configurations we traced.
- **Failure analysis** (Section 5). We document nine attempts that failed or were rejected, and show how we found and removed three performance problems: a CPU addition whose speed changed from run to run, allocator stalls in the offload code, and FP8 GEMM outputs that the storage pass missed.

Appendix A gives the code commit, container, environment variables, and flags for every configuration (steps 8–10 reproduce exactly; step 9 requires deleting one loop).

Table 1: Hardware, software, and model. Both instances have the same hardware and software (checked 2026-10-10).

Item	Value
Instance	Lambda 1× GH200 480GB (KVM VM, GPU pass-through), \$2.29 per hour
GPU	Hopper, HBM3 nominally 96 GB (95.6 GiB total, 94.5 GiB usable), BF16 dense peak 989 TFLOPS
Power	<code>nvidia-smi: enforced.power.limit</code> 700 W (<code>power.limit</code> 900 W), “SW Power Cap” flag on, cap set outside the VM. Module (CPU+GPU+memory) limit 1 000 W
CPU	Grace, 64 of 72 cores visible; 431 GiB of 480 GB LPDDR5X usable (node 0)
Link	NVLink-C2C, 450 GB/s per direction[6]
Software	NGC PyTorch 26.09: torch 2.14.0a0, CUDA 13.4, Triton 3.8.0, torchao 0.19.0. FlashAttention-3 3.0.0 (built from source). Details in Appendix A
Transformer	30 layers, hidden dimension 2 560. GQA[9]: 20 query heads, 4 KV heads, head dim 128. QK-Norm, RoPE. 25 sliding-window layers ($W = 1\,024$), 5 global attention layers
FFN	Layer 1: dense SwiGLU. Layers 2–30: fine-grained MoE. 64 routed experts (width 1 280, top-4) and 1 shared expert (width 5 120)[4]
Router	DeepSeek-V3 style[10, 11]: sigmoid scores. Bias used only for expert selection. Sequence-wise auxiliary loss. Dropless
Tokens	Vocabulary 204 800 (SuperBPE[12]), separate input and output embeddings. Sequence length 8 192, document masking
Parameters	20 989 542 400 total, 3 360 194 560 active

2 Setup and MFU Definition

2.1 Hardware and Model

We trained on Lambda cloud instances with one GH200 480GB each. Instance A ran from the start, and instance B was paired with it through DiLoCo at 4B tokens. Inside each VM, HBM shows up as NUMA node 1 and LPDDR5X as node 0 (Table 1).

Model. FFNs hold 92.8% of the 20.99B parameters, yet each token touches only 3.36B.

Training. Each micro-batch holds $6 \times 8\,192 = 49\,152$ tokens (48K). The global batch ramps from 2M to 8M tokens over the first 33B tokens, and each DiLoCo worker takes half of it. Muon[13, 14] updates the 19.94B matrix parameters and AdamW[15] the remaining 1.05B, with master weights in FP32. The WSD schedule[16, 17] was planned for 1.1T tokens, and Stage 1 ends at 0.51T. Every number in the paper comes from runs under the 700 W power cap (Table 1).

2.2 MFU Definition

MFU is the model FLOP rate divided by a fixed peak[18]:

$$\text{MFU} = \frac{T \times F}{P}, \quad P = 989 \times 10^{12} \text{ FLOP/s}, \quad (1)$$

where T is the number of tokens per second per GPU and F is the model FLOPs per token[19],

$$F = 6N_{\text{active}} + 6n_h d_h \sum_{l=1}^L c_l, \quad c_l = S \text{ (global)}, \quad c_l = \min(2W, S) \text{ (local)}. \quad (2)$$

With $n_h d_h = 2\,560$, sequence length $S = 8\,192$, and window $W = 1\,024$, the two terms come to 20.161 GFLOP and 1.416 GFLOP, giving $F = 21.577$ GFLOP per token and 1.0606 PFLOP per 48K micro-batch. Recomputed FLOPs are left out; counting them would give HFU[20].

Table 2: Location and size of the training state (20.99B parameters, 48K micro-batch, 1 GiB = 2^{30} bytes). The four FP32 rows sum to 238.5 GiB.

State	Location	Size (GiB)	Notes
BF16 working weights	GPU HBM	39.1	Router in FP32
Activations and workspace	GPU HBM	about 36	Peak allocation during training (75.3 GiB) minus the weights
FP32 master weights	CPU (pinned, mapped)	78.2	$20.99\text{B} \times 4$ bytes
FP32 gradient accumulation buffer	CPU (pinned, mapped)	78.2	Updated every micro-batch
Muon momentum	CPU (pinned, mapped)	74.3	19.94B parameters
AdamW m, v	CPU (pinned, mapped)	7.8	1.05B parameters
BF16 staging buffer	CPU (pinned)	39.1	One slot per parameter
CPU storage region	CPU (pinned)	72.0	56.4 used per micro-batch
Pinned CPU total		349.6	Fits in node 0 (431 GiB)

Denominator. Every run is divided by the BF16 dense peak, which keeps BF16 and FP8 stages on the same scale. The GH200’s GPU is H100 SXM-class Hopper silicon, so we use the H100 SXM figure[21]. In the final configuration 47.0% of FLOPs run in FP8. Dividing the FP8 stages by the mixed FP8/BF16 peak of 1 293 TFLOPS instead would scale their MFU by 0.76, turning 48.9% into about 37%, while BF16 stages would be unaffected (Section 7.1). Equation (2) also treats every row as one document, whereas training cuts attention at document boundaries. Section 6.2 measures the difference: counting attention only within documents lowers real-training MFU by 1.6 percentage points ($F \times 0.967$).

Measurement. We report two kinds of MFU and keep them apart. The training script computes real-training MFU every 10 steps from the growth in global tokens, wall-clock time, and the number of workers, so optimizer, data loading, validation, and synchronization time are all included. We report the median for each interval, skipping the first line after a restart, and add synchronization and checkpoint time in Section 6.3. Benchmark MFU is measured by `scripts/bench_mfu.py` on synthetic data (random tokens, one document per row), which times a micro-batches (`--accum`) and one optimizer step:

$$t_\mu = \frac{t_{\text{iter}} - t_{\text{opt}}}{a}, \quad \text{mfu_at_batch} = \frac{N_{\text{micro}} \times F}{(t_\mu + t_{\text{opt}} \times N_{\text{micro}}/B_{\text{amort}}) \times P}, \quad (3)$$

where t_{iter} and t_{opt} are the median iteration and optimizer times and N_{micro} is the number of tokens in a micro-batch. The optimizer time is amortized over $B_{\text{amort}} = 8\text{M}$ tokens. Steps 0–4 were measured with `--accum 1`, which hides the accumulation cost paid on every micro-batch; from step 5 on we used `--accum 8`, which includes it. Speedups are throughput ratios computed from micro-batch times, $t_{\text{old}}/t_{\text{new}} - 1$.

3 System Design and State Placement

3.1 State Placement

HBM holds just the BF16 working weights (39.1 GiB) and activations (Figure 2, Table 2). All FP32 state, namely the master weights, gradient accumulation buffer, Muon momentum, and AdamW moments, sits in Grace memory and totals 238.5 GiB. We allocate it as pinned memory and map it into the GPU address space, so GPU kernels read and write it directly over NVLink-C2C. In total 349.6 GiB is pinned, which fits in the 431 GiB available on node 0.

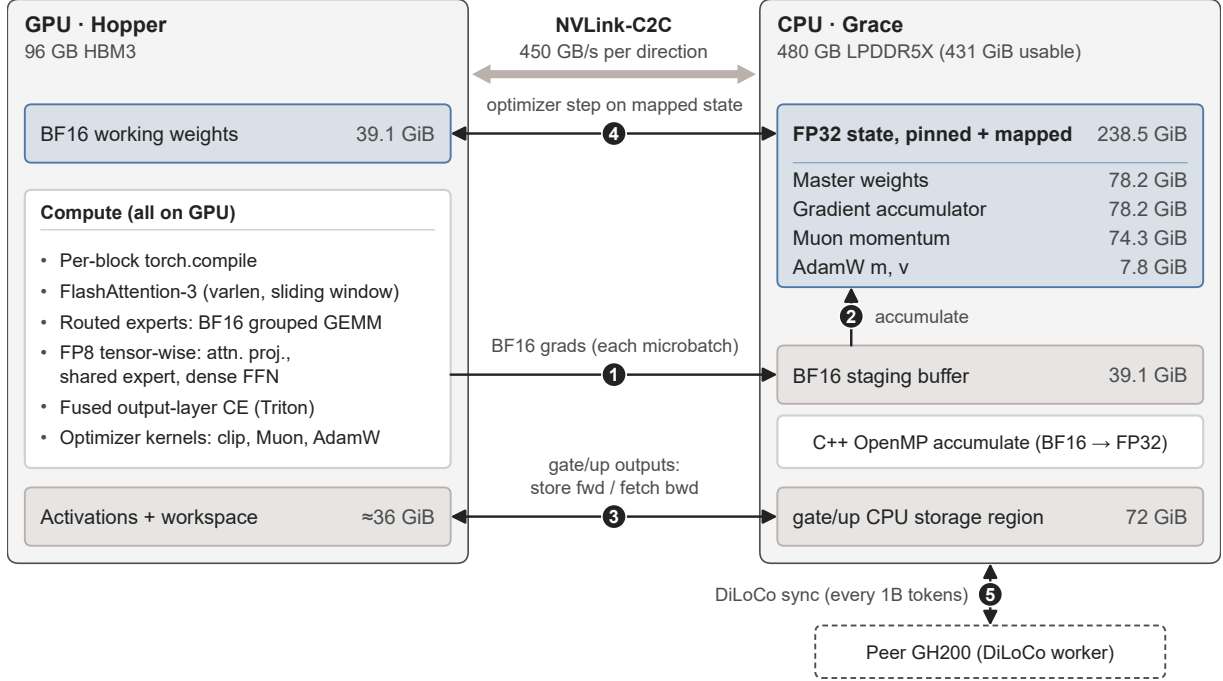


Figure 2: State placement and data flow on one GH200. (1) A post-accumulate-grad hook copies each BF16 gradient into a pinned staging buffer on a side stream, and (2) a CPU thread adds it to the FP32 accumulation buffer with a C++ OpenMP kernel. (3) FFN gate/up GEMM outputs go to the CPU storage region in the forward pass and come back one layer ahead in the backward pass. (4) At the optimizer step, GPU kernels update the mapped FP32 state over C2C and refresh the BF16 weights. (5) Every 1B global tokens, the two workers exchange pseudo-gradients (DiLoCo).

3.2 Gradient Accumulation on the CPU

Every parameter carries a post-accumulate-grad hook, 449 in all. Once its slot in the BF16 staging buffer is free, the hook copies the BF16 gradient there on a side stream; the copy engine does the work, so no SMs are spent on it. A single CPU worker thread then runs a C++ OpenMP kernel (`load_inline`, 32 threads, GIL released) that widens BF16 to FP32 with a 16-bit shift and adds the result into the accumulation buffer. On the first micro-batch of a step the kernel writes instead of adding, which removes the need for a zero-fill pass, and its output matches `add_bit` for bit. Each 48K micro-batch sends 42.0 GB to the host and causes about 210 GB of CPU memory traffic. The kernel alone takes 0.75–0.78 s (about 280 GB/s, bound by memory bandwidth); inside the full benchmark it takes 0.92–0.98 s and still hides behind roughly 2.1 s of GPU computation. Section 5.2 explains how we arrived at this design.

3.3 Optimizer on Mapped CPU Memory

The GPU runs the optimizer, operating directly on mapped CPU memory over NVLink-C2C[7]; the CPU never executes optimizer code, which is where we depart from ZeRO-Offload[22]. Gradient clipping works on the mapped accumulation buffer (`_foreach_norm`, `_foreach_mul`). Muon momentum stays in mapped memory while Newton–Schulz runs in BF16 in HBM, batched by shape. Weight decay and the update are then applied to the mapped FP32 master weights, and a single copy refreshes the BF16 working weights. AdamW runs as elementwise kernels on the mapped state, with the BF16 conversion fused in. An optimizer step takes about 5.7 s, roughly 6% of a worker step at 1.97M tokens and 3.3% at 4M tokens. Adding up the memory traffic of the individual operations, we estimate that each step moves about 1.27 TB (about 224 GB/s).

3.4 Block-Level Compilation and Static MoE

We compile each of the 30 blocks separately with `torch.compile` (Inductor, default mode)[23]. A block covers attention (FlashAttention-3[24]), the router, the routed experts, and the shared expert, while the embedding, output layer, and loss stay outside the compiled region. MoE shapes are static: the dispatch buffer always has $M = \lceil (N_{\text{micro}}K + 15E)/16 \rceil \cdot 16$ rows, where E is the number of experts and K is the K of top- K routing, so a 48K micro-batch gives $M = 197\,568$. Swapping `bincount` for a fixed-size `scatter_add_` and avoiding `.item()` removed every host synchronization. The experts run through `torch._grouped_mm` with int32 offsets, a custom SwiGLU backward saves the gate/up outputs, and the FP32 combine is deterministic. Passing the same tensor as both `cu_seqlens_q` and `cu_seqlens_k` split the block graphs 24 times per micro-batch, and cloning `cu_seqlens_k` made every one of those breaks disappear.

3.5 Selective Activation Storage

With `--recompute none` and an activation memory budget of 0.15, the partitioner limits saved activations to 15% of what saving everything would take, recomputing cheap elementwise operations first[25]. The budget on its own still left the FFN gate/up GEMM outputs to be recomputed (Section 4.5), so we add a custom pre-pass that marks nodes directly in the joint graph. The pass tags a forward node with `MUST_SAVE` when the node cannot be reached from the backward tangents and its operator is one of `aten.mm`, `_grouped_mm`, `_scaled_mm(_v2)`, or `_scaled_grouped_mm(_v2)`. The output must also be two-dimensional with an FFN width (1 280, 5 120, or 10 240) as its last dimension, and the pass raises an error if an FFN width ever equals the attention width. Saved tensors are routed to the CPU storage region through `saved_tensors_hooks`, and the same 72 GiB pinned region is reused for every micro-batch. The pack function copies each tensor on a D2H stream and holds the GPU original on a pending list until its copy event completes, without `record_stream`; once more than 4 GiB is pending, the CPU stalls and waits. The unpack function prefetches one layer ahead into a compute-stream buffer (Section 5.3). A 48K micro-batch stores 118 tensors (58 routed-expert, 58 shared-expert, and 2 dense-FFN; 56.4 GiB), moving 60.5 GB across the link in each direction. Gradients match bit for bit, FP8 included, against a run that keeps the same marked tensors on the GPU, and the number of grouped GEMM calls per MoE layer falls from 11 to 9.

3.6 Output-Layer Cross-Entropy

The Triton fused output-layer cross-entropy (Triton CE) works through tokens in chunks of 4 096. For each chunk, cuBLAS writes the logits into a reusable 1.68 GB buffer, and a Triton kernel[26] accumulates an FP32 online log-sum-exp and then overwrites the logits with their BF16 gradients, z-loss and normalization included. An FP32-output GEMM (`torch.addmm(..., out_dtype=float32)`) accumulates the weight gradient, and the count of valid tokens stays on the GPU.

3.7 FP8 Coverage

We use `torchao Float8Linear`[27] with the per-tensor recipe: one dynamic scale per tensor, with fast accumulation turned off for the forward output GEMM. FP8 covers the attention projections, the shared experts, and the dense FFN (47.0% of FLOPs), while the routed experts, the output layer, and the attention core stay in BF16. Parameters are stored in BF16 and cast to FP8 on every forward pass. With no amax history involved, a recomputed forward pass quantizes exactly as the original did. Changing only the recomputation partition still changes the FP8 gradients (relative norm difference on a small model: median 6.5%, maximum 13.9%; close to 0 for BF16), since the BF16 rounding ahead of quantization depends on the partition. FP8 tests therefore have to compare runs that share a partition.

3.8 DiLoCo Across Two Instances

Each worker runs the single-GPU system above on its own data shard, following DiLoCo[8]. Every 1B global tokens the two workers exchange 84 GB of FP32 pseudo-gradients over 8 SSH streams.

Table 3: Benchmark results for the full 30-layer model after each change. MFU is `mfu_at_batch`, measured with `--accum 1` for steps 0–4 and `--accum 8` afterward. Steps 9 and 10 were measured on instance B and the rest on A. Memory is peak reserved memory.

Step	Change	Micro-batch	Time (s)	MFU	Memory (GiB)
0	Baseline: BF16, full recompute, only elementwise functions compiled, <code>--moe-chunk 0</code>	64K	6.677	21.1%	74.7
1	Block-level <code>torch.compile</code> , static MoE	64K	5.096	27.9%	75.1
2	Activation memory budget 0.1 replaces full recompute	32K	2.499	28.4%	85.5
3	CPU accumulation (copy engine + CPU addition)	32K	2.131	33.2%	85.5
4	FlashAttention-3	32K	2.057	34.4%	83.4
5	C++ OpenMP accumulation kernel	32K	2.079	34.1%	84.9
6	Experts 128×640 top-8 → 64×1 280 top-4 (model change)	32K	1.892	37.3%	83.7
6'	Same model, training micro-batch, budget 0.05	48K	2.836	37.3%	88.2
7	Triton fused output-layer CE	48K	2.501	42.2%	88.2
8a	Graph breaks 24 → 0	48K	2.569	41.2%	71.1
8	FFN gate/up outputs stored in CPU memory, budget 0.1	48K	2.309	45.7%	83.3
9	Per-tensor FP8, budget 0.15	48K	2.147	49.2%	74.2
10	FP8 GEMM outputs also stored	48K	2.078	50.7%	79.9

Both compute the outer Nesterov update (learning rate 0.7, momentum 0.9) in rank order, which makes the results bitwise identical. Checkpoints are written only right before a synchronization, and a fingerprint of the master weights confirms that the two workers agree.

4 Optimization Path

Between October 6 and 11, 2026, we worked under the following rules: leave the training objective alone (the model change in step 6 is flagged separately), keep only gains that a measurement confirmed, and touch the main run only at checkpoint boundaries. The first rule covers changes made for throughput; the auxiliary-loss change in Table 5 and the router-bias changes at 3B and 4B tokens were made for load balancing. Table 3 lists the steps in order, and every number can be traced to result files on the servers (Appendix B). MFU in this section means the benchmark value `mfu_at_batch` (Section 2.2); real-training MFU is reported in Section 6.

4.1 Steps 1–4: Compilation, Recomputation Budget, CPU Accumulation, FA3

Step 0 (BF16, full recompute, only elementwise functions compiled) took 6.677 s per 64K micro-batch (MFU 21.1%), and a 32K profile suggested that about 45% of that time was spent outside GEMMs. Step 1 compiled each block with `torch.compile` and made the MoE shapes static (Section 3.4), cutting the 64K time to 5.10 s (+31% throughput); the static shapes by themselves were worth about 2.5%.

Step 2 swapped full recompute for an activation memory budget of 0.1. At 32K the time dropped from 2.80 s to 2.50 s (+12%), but throughput then stalled around 28.5%, about where full recompute had topped out at 96K.

Step 3 replaced GPU accumulation into mapped FP32 buffers, a scheme that moved 8 bytes per parameter over the link, used SMs, and cost 0.38 s (about 15%) at 32K. BF16 gradients now go through the copy engine (2 bytes per parameter) and a CPU thread does the FP32 addition

(Section 3.2), which brought the time from 2.50 s to 2.13 s (+17%). Step 4 built FlashAttention-3 from source and reached 2.057 s (+3.6%).

4.2 Step 5: C++ Accumulation Kernel

Step 5 replaced PyTorch’s CPU addition with a C++ OpenMP kernel (Section 3.2), which removed the throughput divergence we had seen in real training (Section 5.2). Two measurements both gave 2.079 s (MFU 34.1%), slightly behind step 4. The difference comes from the switch to `--accum 8` at this step, which exposes the CPU addition that `--accum 1` had hidden.

4.3 Step 6: Expert Shape (Model Change)

Step 6 replaced 128 experts of width 640 (top-8 routing) with 64 experts of width 1280 (top-4 routing), keeping the total and active parameter counts nearly the same. The larger grouped GEMMs lifted MFU from 34.1% to 37.3% (+9.9%). The change also lowers the granularity $G = 2d/I$ from 8 to 4, outside the 8–12 range covered by MoE scaling studies[28]; under the definition of Ludziejewski et al.[29], the original design sits at 16. We did not measure model quality, so this gain should be read apart from the system-level gains.

4.4 Step 7: Fused Output-Layer Cross-Entropy

We captured a kernel trace at one of the main run’s checkpoint stops (64×1280 model, 48K micro-batch). Elementwise and copy operations around the output layer accounted for 397 ms (13.9%). With the Triton fused CE from Section 3.6, that time fell to 17 ms, plus 21 ms for the CE kernel itself. At the same stop the micro-batch time went from 2.831 s to 2.501 s (+13.2%, MFU 37.4% to 42.2%), and reserved memory held at 88.15 GiB. The 2.831 s baseline was remeasured at that stop; the 6’ row of Table 3 comes from an earlier run (2.836 s).

4.5 Kernel Time Breakdown and the 700 W Cap

The trace of the step 7 configuration (Figure 3(a)) splits a 2518 ms micro-batch into dense GEMMs at 1185 ms (47.0%), grouped GEMMs at 686 ms (27.2%), Inductor fused kernels at 285 ms (11.3%), FlashAttention-3 at 190 ms (7.6%), CuTe copies at 60 ms, and other kernels at 61 ms. Only 51 ms (2%) was idle, which caps any gain from CUDA graphs at about 2%, so we did not try them.

cuBLAS ran the dense GEMMs at 640–665 TFLOPS, about 66% of the BF16 peak; the shared-expert gate GEMM, for example, does 1.288 TFLOP in 1.98 ms. Grouped GEMMs averaged 599 TFLOPS. We have no clock log from the trace itself, but a snapshot taken during training on October 7 showed the 700 W software power cap in effect and SM clocks averaging 1.55–1.60 GHz (maximum 1.98 GHz).

A 48K micro-batch is 1.0606 PFLOP. Counting model FLOPs without recomputation, the BF16 GEMMs and attention alone need about 1.0 s (dense), 0.56 s (grouped), and 0.19 s (FlashAttention-3), roughly 1.76 s altogether. Even if everything else were free, MFU without the optimizer would top out near 61% ($1.0606 \text{ PFLOP} \div 1.76 \text{ s} \div 989 \text{ TFLOPS}$), or 59.7% with the optimizer amortized as in the benchmark. That 59.7% is the dashed line in Figure 1 and Figure 3(b). We put the realistic ceiling at 52–55%, and getting past 60% would take FP8 or a higher power limit. Since we never ran at 900 W, we cannot say how much of the gap comes from the power cap and how much from the matrix shapes; our estimate for the gain at 900 W is +8–15%.

The trace also exposed recomputation. Each MoE layer issued 11 grouped GEMM calls (3 forward, 2 recompute, 6 backward), and the recomputed gate/up GEMMs took 250–285 ms (10–11%), which is what led to step 8.

4.6 Step 8: Storing Gate/Up Outputs

Step 8 keeps the FFN gate/up outputs in CPU memory instead of recomputing them (Section 3.5). After the allocator fix (Section 5.3), storage was 8.5% faster, but reserved memory reached 93.1 GiB, above the 91.5 GiB limit we had set for ourselves. Step 8a removed the graph

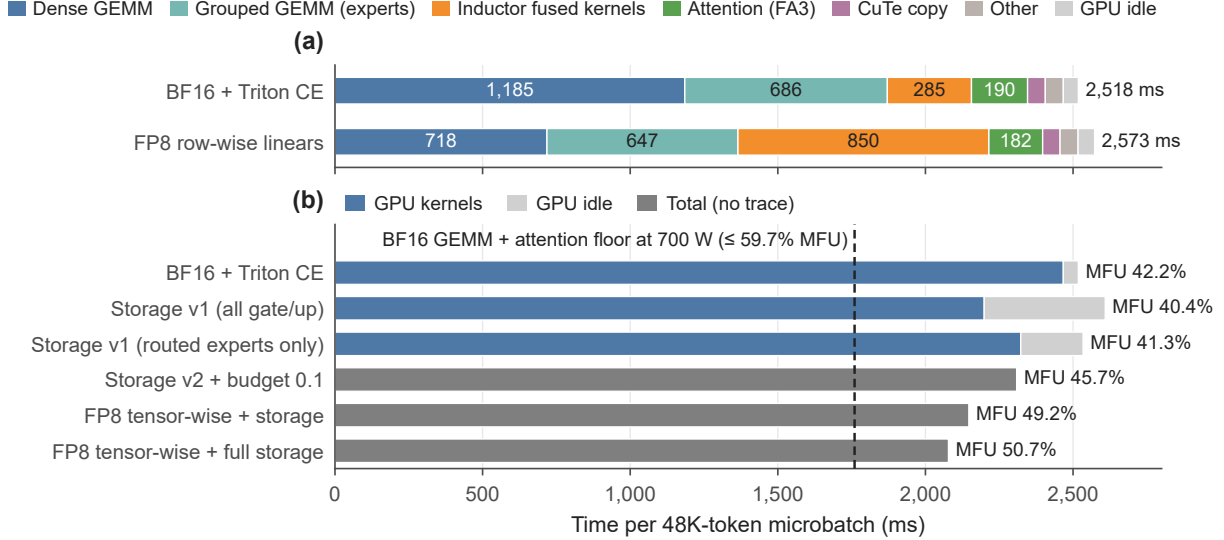


Figure 3: GPU time for one 48K micro-batch. (a) Breakdown by kernel type for the BF16 configuration with fused cross-entropy and for the torchao row-wise FP8 configuration (Section 5.1). (b) Micro-batch time across successive configurations. The top three rows show kernel and idle time from traces, and the bottom three show micro-batch times measured without traces, labeled with benchmark MFU. The dashed line is the 700 W ceiling set by the BF16 GEMMs and FlashAttention-3 alone (1.76 s, MFU 59.7%).

breaks. On its own it slowed things down (2.501 s to 2.569 s), yet it cut reserved memory from 88.2 GiB to 71.1 GiB, since the budget now applied to the whole block. That headroom let us combine full storage with a budget of 0.1, giving 2.309 s (MFU 45.7%) and 83.3 GiB with no overflowed tensors. Across the restart, the loss at the same step agreed to the fourth printed decimal place.

4.7 Step 9: Per-Tensor FP8

Row-wise FP8 failed (Section 5.1). Per-tensor FP8 uses one scale per tensor, which makes quantization cheap and removes the need for a transposed cast. At a budget of 0.05 it cut the micro-batch time from 2.569 s to 2.425 s (+5.9%, MFU 41.2% to 43.6%) and saved 3.7 GiB of memory, the first time FP8 paid off for us. We did not capture a kernel trace for it.

FP8, storage, and the budget interact instead of adding up (Figure 4(b)). On instance B, with the same code on the same day, BF16 with storage gave 45.8% at a budget of 0.10 and 45.7% at 0.15 (A: 45.7%, Table 3). FP8 with storage gave 45.8% and 49.2% at the same two budgets, and 47.0% without storage at 0.15. Our reading, which we did not test separately, is that FP8 frees activation memory and recomputation drops only once that headroom goes back into the budget.

For changes in precision, we had originally required at least a 10% speedup over BF16 (Transformer Engine FP8 fell short, Section 5.1). Per-tensor FP8 with storage delivered +7.5% throughput (2.308 s to 2.147 s, MFU 45.8% to 49.2%). We adopted it conditionally when moving to two instances at 4B tokens: if per-domain validation loss or in-context metrics drift away from the BF16 trend, training goes back to BF16.

4.8 Step 10: Storing FP8 GEMM Outputs

Under FP8 the storage pass missed the FP8 GEMM outputs (Section 5.3), so we extended the marking targets to `_scaled_mm`, `_scaled_mm_v2`, and `_scaled_grouped_mm(_v2)`. Remeasured on October 10, just before the change, the step 9 code ran at 2.143 s (49.23%); Table 3 lists the October 9 measurement of 2.147 s (49.15%). The extended marking brought the micro-batch to 2.078 s (50.74%), a +3.1% throughput gain, while reserved memory rose from 74.2 GiB to 79.9 GiB. The budget no longer mattered (50.74–50.78% at 0.10, 0.15, and 0.20), and BF16 did not regress (45.84%). We deployed the change on both instances right after the 10B-token

Table 4: Failed or rejected attempts, each with a baseline measured under the same conditions. The last row was never attempted.

Attempt	Result	Cause / Decision
TE FP8 block scaling + FP8 weights (32K, full recompute)	13.6% (BF16 20.0%)	Quantizes and multiplies each of 128 small experts separately. FP8 weights are 1.6 times slower
TE FP8, non-expert linear layers only (64K)	21.2% (BF16 21.1%)	Micro-batch throughput +2.6% (<code>mfu_at_batch</code> +0.7%), short of the 10% rule
Bulk activation CPU offload (eager, 8K)	14.1% (slower than full recompute)	77 GiB at 8K (duplicated inputs, FP32 copies). 32K exceeds CPU memory
GPU-CPU accumulation split (<code>add_path</code> , 3 expert shapes)	26.9–31.7% (fast <code>add_</code> runs: 33.8–34.5%)	GPU addition competes for LPDDR5X bandwidth and slows CPU addition
First implementation of gate/up storage	40.4% (same-session baseline 42.3%)	Suspected causes: stream-bound buffers and <code>record_stream</code> . 93.1 GiB reserved, 410 ms idle time
Compiler built-in activation offload	Failed to run	<code>setStorage</code> error in an experimental PyTorch feature
torchao row-wise FP8 linear layers	38.9% (BF16 41.2%)	Quantization kernels (+565 ms) outweigh GEMM savings (−467 ms)
Row-wise + FP8 grouped experts	37.6% (BF16 41.2%)	Same quantization cost
Experts 32×2 560 top-2	37.8–38.5% (64×1 280: 37.3%; at 48K, budget 0.07 vs. 0.05)	Slightly faster, but rejected because of the quality risk of top-2 routing with wide experts
900 W power limit	Not attempted	Limit set outside the VM. Estimated +8–15%

synchronization, at a cost of 20 minutes of downtime.

5 Failure Analysis

Table 4 collects nine attempts that failed or were rejected, each listed next to a baseline measured under the same conditions. Sections 5.2 and 5.3 look at the performance problems we ran into along the way.

5.1 FP8 Approaches

Our first plan was Transformer Engine (TE) FP8 with block scaling and FP8 weights[30], and it ran about 1.5 times slower than BF16, because TE’s grouped linear quantizes and multiplies each of the 128 small experts (width 640) separately. With FP8 weights alone it was 1.6 times slower.

torchao row-wise FP8 linear layers (graph breaks removed, budget 0.05) reached 38.9% against 41.2% for BF16, or 5.7% lower throughput. The trace of the first row-wise run, which still had its graph breaks, shows where the time went (Figure 3(a)). Dense GEMM time fell from 1 185 ms to 718 ms (−467 ms), but Inductor fused kernels grew from 285 ms to 850 ms (+565 ms), since row-wise quantization needs amax reductions along the token axis plus transposed casts for the weight gradients. The kernel count roughly tripled, to 35 408. In the end the quantization kernels decided the outcome, and the faster GEMMs could not make up for them.

5.2 Throughput Divergence in CPU Accumulation

Before the main run, real-data check runs with identical settings split into two speed regimes, one near 22% MFU and one near 32%. The benchmark had hidden this: with `--accum 1` the

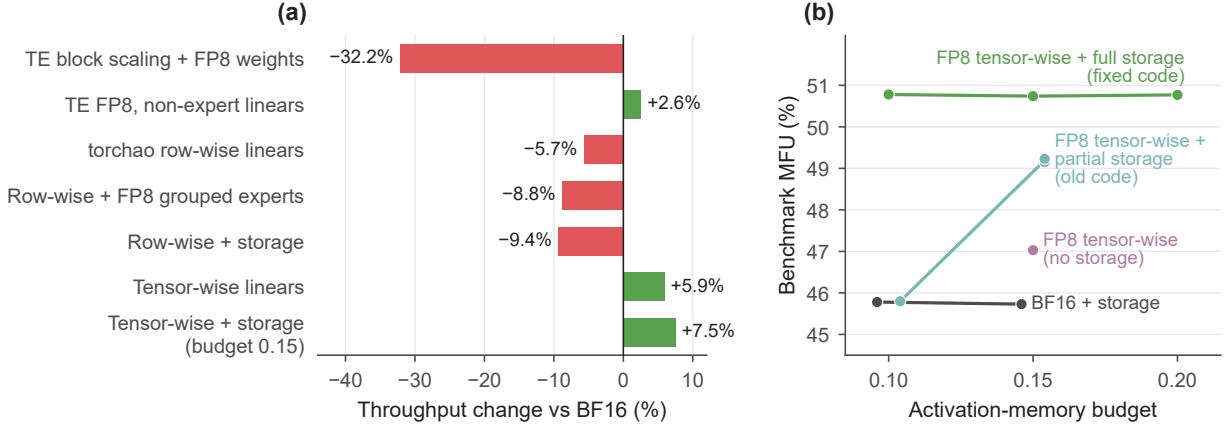


Figure 4: FP8 approaches and combinations. (a) Change in micro-batch throughput for each attempt against its own BF16 baseline. Row-wise + storage is compared with BF16 + storage at budget 0.05, and per-tensor + storage at budget 0.15 with BF16 + storage at budget 0.10 (Section 4.7). TE block scaling and torchao row-wise are slower, the TE linear-layer hybrid gains 2.6%, and per-tensor gains more. (b) Benchmark MFU against the activation memory budget on instance B, with repeated measurements shown as two points. FP8 with full storage (step 10) hardly depends on the budget.

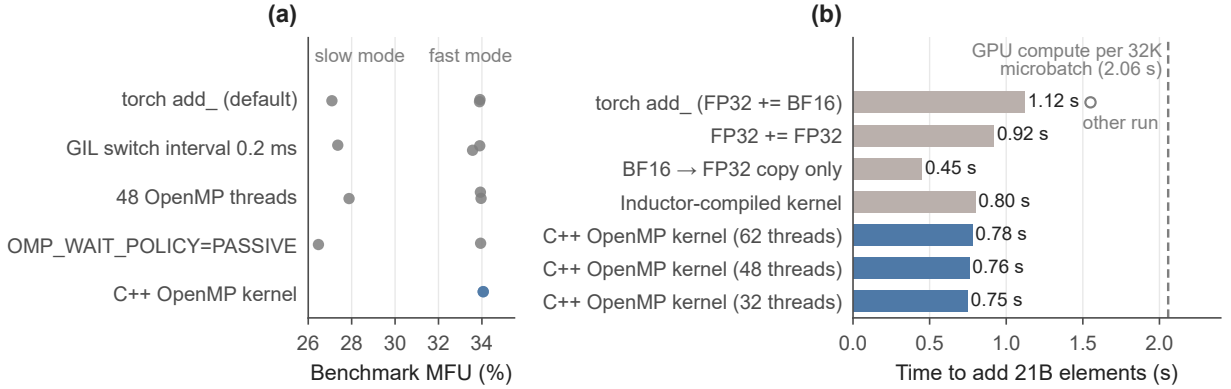


Figure 5: CPU-side gradient accumulation. (a) Benchmark MFU over repeated runs (32K, --accum 8). Across runs, the PyTorch add_ path lands in either a slow or a fast mode, and none of the settings we tried changed that; the C++ OpenMP kernel is fast every time. (b) CPU time to add 21B elements into the FP32 buffer. The dashed line marks the GPU compute time of a 32K micro-batch.

CPU addition is folded into the optimizer time, so steps 0–4 understated its cost. We switched all later benchmarks to --accum 8, and the split persisted (Figure 5(a)). Four of 11 runs were slow (2.54–2.67s, MFU 26.5–27.9%), and each process stayed in whichever regime it started in. Shrinking the GIL switch interval 25-fold, cutting the OpenMP threads from 62 to 48, and changing OMP_WAIT_POLICY made no difference.

Before writing the C++ kernel we tried splitting the accumulation, with a GPU kernel adding a subset of the parameters into the mapped buffers. The hook wait went away, yet MFU stayed at 26.9–31.7%, below the 33.8–34.5% of fast add_ runs, because the GPU additions compete for the same LPDDR5X bandwidth and slow the CPU addition down.

In slow runs the CPU addition took 2.48s per micro-batch against 1.66s in fast ones, and the hooks waited 0.88s for free staging slots. A standalone PyTorch add_ (FP32 += BF16) over 21B elements took 1.12s in some runs and 1.55s in others (Figure 5(b)). It appears not to be vectorized on Grace, though we never pinned down the root cause. The C++ OpenMP kernel takes 0.75–0.78s with 32–62 threads (about 280 GB/s, bandwidth-bound) and matches add_ bit for bit. With it in place, two measurements both came in at 2.079s, with the CPU addition down to 0.92s and the hook wait to 0.003s.

5.3 Offloading and Storage

The first version of gate/up storage cut kernel time from 2 467 ms to 2 199 ms, the 268 ms saving we had expected, but GPU idle time rose from 51 ms to 410 ms. Throughput fell by 4.4% (2.615 s vs. 2.501 s in the same session; MFU 40.44% vs. 42.26%, with the step 7 run at 42.24%), and reserved memory reached 93.1 GiB. Neither a lower budget nor storing only half the tensors changed the reserved memory, which pointed to the allocation pattern. We had two suspects: prefetch buffers allocated on the H2D stream might be tied to that stream’s pool, and `record_stream` on the original tensors might delay their release. We fixed both at once and did not separate their effects. Destination buffers now come from the compute stream’s pool, and references to the originals sit on a pending list until their copy events complete. The fix gave +8.5% throughput (MFU 42.3% to 45.8%), and step 8a later solved the memory problem (Section 4.6).

While preparing this paper we noticed that FP8 stored only 40.9 GiB across 87 tensors, compared with 56.4 GiB across 118 tensors for BF16. The marking pass knew only `aten.mm` and `_grouped_mm`, so FP8 GEMMs (`aten._scaled_mm`) went unmarked; the shape-based selector still picked up one BF16 tensor of shared-expert width per MoE layer (13.6 GiB in total). Step 10 fixes this (Section 4.8).

6 Real-Training Results

The numbers in this section come from training on billions of tokens of real data, and we keep them apart from the benchmarks of Section 4. They are taken from `main_phase1.log`, which covers 2026-10-06 through 07:35 UTC on 2026-10-11 (cumulative 0–10.91B tokens); instance A is logged from the start and B from 4B tokens. The time series extracted from the log are in `data/train_log_{A,B}.tsv`.

6.1 MFU During Training

Table 5 gives the per-interval medians of the 10-step MFU. After the FP8 storage extension (Section 4.8) the medians reached 48.9% (A) and 49.1% (B). Over the stretches after 10B tokens that contain no synchronization, time-weighted MFU was 48.87% (A) and 49.09% (B), which corresponds to 22.4K and 22.5K tokens/s per GPU and 483–486 TFLOPS of model FLOPs.

MFU kept climbing through the FP8 intervals because the per-worker batch grew from 1.4M to 1.9M tokens, shrinking the share taken by the optimizer step, which is fixed at 5.7 s. B’s 8–10B interval includes a 22-minute pause for benchmarking, and the post-10B values rest on about 5.6 hours of observation (23 ten-step windows).

6.2 Gap Between Benchmark and Real Training, and Document Masking

Per-worker batches in real training are about 2M tokens. At that size the benchmark formula predicts 48.3% for step 10 at 1.97M tokens and 48.4% at 2.02M tokens, and the measured 48.9% and 49.1% sit 0.5–0.8 percentage points higher. For step 9 with a 1.77M-token batch the formula predicts 46.5%, against time-weighted values over 8.0–8.4B tokens of 47.4% (A) and 47.6% (B) (these are not the table medians). Real training therefore ran 0.5–1.1 percentage points above the benchmark formula.

Equation (2) counts attention as if each 8 192-token row were one document, but in training attention stops at document boundaries. To measure the effect exactly, we drew 20 100 rows the way the loader builds them (Stage 1 mixture) and took segments as FA3 varlen sees them. Document fragments have a median length of 427 tokens and a mean of 850.6. Masked attention comes to 0.707 GFLOP per token, against 1.416 from the formula and 1.367 for an exact causal count over a row that is a single document. Total F drops by 3.3% (ratio 0.967, 95% interval 0.9670–0.9674), which lowers real-training MFU by 1.6 percentage points, to 47.3% (A) and 47.5% (B) after 10B tokens. The benchmark has one document per row and barely moves: 50.74% becomes 50.62%, which reflects only the overcount for local windows. The earlier estimate of about 1 percentage point came from the 0.5–1.1 point gap above, and that gap reflects time actually

Table 5: Real-training MFU by interval (median of 10-step values, excluding the first line after a restart). Where two values appear, they are for instances A / B. “MFU, masked” counts attention only within documents (Section 6.2). ^aArithmetic evaluation probe (one forward micro-batch every 50 steps), added at 8B tokens. Throughput did not change.

Tokens (B)	Changed setting	Tokens/s/GPU	MFU (%)	MFU, masked (%)	Batch per worker (M tokens)
0.02–1.98	BF16, torch CE, memory budget 0.05	16 744	36.5	35.3	2.0 → 2.4
2.00–2.98	+ Triton CE	18 898	41.2	39.8	→ 2.6
3.01–3.98	+ gate/up storage, no graph breaks, budget 0.1	20 553	44.8	43.3	→ 2.7
4.01–5.99	+ per-tensor FP8, budget 0.15, DiLoCo (2 workers)	21 345 / 21 454	46.6 / 46.8	45.1 / 45.3	1.4 → 1.5
6.02–7.98	Sequence-wise auxiliary loss α $10^{-4} \rightarrow 10^{-3}$	21 610 / 21 686	47.1 / 47.3	45.6 / 45.7	→ 1.7
8.01–9.97	+ evaluation probe ^a	21 774 / 21 869	47.5 / 47.7	45.9 / 46.1	→ 1.9
10.01–10.91	+ FP8 GEMM output storage (step 10)	22 404 / 22 508	48.9 / 49.1	47.3 / 47.5	2.0

saved; it says nothing about overcounted FLOPs. For comparability, the headline numbers keep the conventional formula, with the masked values listed alongside.

6.3 DiLoCo with Two Instances

All six synchronizations between 5B and 10B tokens produced matching master-weight fingerprints on the two workers. Each one spent 91–124s writing pseudo-gradients, 35–53s moving 84 GB, and 195–227s on the outer update, plus waiting time. With checkpoint saving, this added up to about 10 minutes per 6.2 hours of training, or 2.6%. Applying that 2.6% to the post-10B MFU gives an estimated effective MFU of 47.6% (A) and 47.8% (B), or 46.0% and 46.2% with attention counted only within documents.

6.4 Power and Clocks

At 07:46 UTC on 2026-10-11 we sampled both training instances every 500 ms for 2 minutes. Median power was 659.5 W (A) and 661.8 W (B), and the “SW Power Cap” flag was on in 98.3% and 99.6% of samples. Median SM clocks were 1 500 MHz and 1 485 MHz, 24–25% below the 1 980 MHz maximum, with means of about 1 550 MHz and 1 530 MHz. With a 1 000 W module limit and about 150 W going to the CPU side, lifting the GPU cap would leave room for up to about 850 W.

6.5 Training Stability

Training loss fell from about 12.75 to 2.5–2.7 by 10.9B tokens, with no spikes where settings changed. Validation loss went down in every domain between 5B and 10B tokens (instance A: code from 1.171 to 0.969, web from 4.219 to 3.785).

7 Discussion

7.1 How to Read the MFU Numbers

Denominator and FP8. We divide every MFU by the BF16 dense peak of 989 TFLOPS, which puts BF16 and FP8 stages on one axis; since the GPU is H100 SXM-class, that peak is the H100

SXM specification. With 47.0% of FLOPs in FP8, the mixed peak is 1 293 TFLOPS (taking the FP8 dense peak of 1 979 TFLOPS for that share). Dividing by the mixed peak instead scales MFU by 0.76, so the same 483 TFLOPS would read as about 37%.

Power cap. At 700 W, BF16 dense GEMMs ran at 640–665 TFLOPS (about 66% of peak), which holds BF16 MFU to about 60% (Section 4.5). Because 700 W is also the TDP of the H100 SXM, the power conditions resemble those of H100 clusters.

Document masking. Counting attention only within documents lowers real-training MFU by 1.6 percentage points ($F \times 0.967$, Section 6.2).

Benchmarks and aggregation. The benchmark (random tokens, 8M-token amortization) is not a clean ablation: steps 0–4 used `--accum 1` and later steps `--accum 8`, and no code snapshots survive for steps 0–6. In intervals without synchronization (0–4B and after 10B), the time-weighted mean stays within 0.04 percentage points of the median. Between 4B and 10B, each 2B-token interval contains two synchronizations, of which the log records at most one (the other coincides with a restart). A single logged synchronization lowers the time-weighted mean by 0.5–0.7 percentage points (Section 6.3).

7.2 What Generalizes

Every 48K micro-batch moves 102.5 GB from GPU to CPU over NVLink-C2C (42.0 GB of gradients and 60.5 GB of stored outputs) and 60.5 GB back, all of it hidden behind about 2 s of computation. On PCIe Gen5 x16 (about 64 GB/s per direction), the D2H traffic alone would take 1.6 s of that 2 s window and leave almost no slack, so this part of the design depends on the superchip. The remaining lessons do not depend on the hardware.

1. When the CPU does part of the work, benchmark with an accumulation count close to that of real training (`--accum  $\geq 8$`) (Section 5.2).
2. Decide on recomputation from the GEMM call counts in a trace (11 vs. 9 per layer). Removing graph breaks changes how the memory budget behaves, so the effect of a change depends on what it is combined with.
3. In offload code, allocator behavior can decide performance (Section 5.3).
4. FP8 gains can be capped by the quantization kernels, whatever the GEMMs gain; for us only per-tensor scaling paid off (Section 5.1).
5. Report `enforced.power.limit` alongside MFU.
6. Count attention FLOPs with real document lengths (3.3% in this study).

7.3 Limitations

1. The ablation mixes code versions and measurement methods.
2. FP8 quality is unverified, since there is no BF16 control run, and under FP8 even a change in recomputation partition shifts the gradients by several percent (Section 3.7).
3. Power data consist of one 2-minute sample and a few snapshots, with nothing recorded during benchmarks.
4. Step 6 changes the model, and DiLoCo, FP8, and the router settings all changed together at 4B tokens.
5. A single model was trained on a single instance type.
6. We cover only about the first 11B tokens of Stage 1 (0.51T) and did not measure the later 8M-token batches or the long-sequence stages (32K, 128K).

7.4 Related Work

Reported MFUs include 38–43% for Llama 3[31], 46.2% for PaLM[18], and 55.2% for MegaScale[32], and MoE Parallel Folding[33] reports 49.3% on Mixtral 8×22B and 39.0% on Qwen2-57B. HFU additionally counts recomputation[20]. Offloading goes back to vDNN[34] and continues through ZeRO[35, 36] and ZeRO-Offload[22]; on the superchip itself, Schieffer et al.[7] measured 375 GB/s and 297 GB/s over NVLink-C2C. Selective recomputation[20] and PyTorch’s activation memory budget[23, 25] decide what to recompute, while MEMO[37] and Transformer Engine[38] offload activations to the CPU. MoE systems such as MegaBlocks[39] mostly center on expert parallelism, whereas on a single GPU the hard parts are small grouped GEMMs, dispatch and combine, and memory. For FP8 we use torchao[27, 40], and DeepSeek-V3 uses block scaling[10]. Fused cross-entropy kernels avoid materializing the logits in memory[41, 42].

SuperOffload[43] reported up to about 600 TFLOPS in a blog post on fine-tuning GPT-OSS-20B and Qwen3-14B at sequence length 4K on a single GH200[44], though it is unclear whether MoE FLOPs were counted over active parameters. Our run sustained 483–486 TFLOPS of model FLOPs, counted over active parameters, during pretraining at 700 W with 47% of FLOPs in FP8. The two numbers cannot be placed on the same axis.

8 Conclusion

We pretrained a 20.99B-parameter MoE (3.36B active) on a single 700 W GH200, with all FP32 state kept in Grace memory. Benchmark MFU rose from 21.1% to 50.7%, and real training reached 48.9–49.1% per GPU, or an estimated 47.6–47.8% if synchronization and checkpointing are counted instead of the masking correction. What held us back was usually the way we measured, the CPU addition, the allocator, the quantization kernels, or the power cap; the arithmetic itself was rarely the limit.

References

- [1] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations (ICLR)*, 2017.
- [2] Dmitry Lepikhin, HyukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations (ICLR)*, 2021.
- [3] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.
- [4] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. DeepSeekMoE: Towards ultimate expert specialization in mixture-of-experts language models. In *Proc. 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1280–1297, 2024. doi: 10.18653/v1/2024.acl-long.70.
- [5] NVIDIA. NVIDIA Grace Hopper Superchip data sheet. Product datasheet, 2023. URL <https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip>.
- [6] NVIDIA. NVIDIA Grace Hopper Superchip architecture in-depth. NVIDIA Technical Blog, 2022. URL <https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/>. Accessed 2026-10-11.
- [7] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. Harnessing integrated CPU-GPU system memory for HPC: a first look into Grace Hopper. In *Proc. 53rd Int. Conf. on Parallel Processing (ICPP ’24)*, pages 199–209. ACM, 2024. doi: 10.1145/3673038.3673110.
- [8] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc’Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. DiLoCo: Distributed low-communication training of language models. arXiv:2311.08105, 2023.
- [9] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In

- Proc. 2023 Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4895–4901, 2023. doi: 10.18653/v1/2023.emnlp-main.298.
- [10] DeepSeek-AI. DeepSeek-V3 technical report. arXiv:2412.19437, 2024.
 - [11] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. Auxiliary-loss-free load balancing strategy for mixture-of-experts. arXiv:2408.15664, 2024.
 - [12] Alisa Liu, Jonathan Hayase, Valentin Hofmann, Sewoong Oh, Noah A. Smith, and Yejin Choi. SuperBPE: Space travel for language models. In *Conference on Language Modeling (COLM)*, 2025.
 - [13] Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. Blog post, 2024. URL <https://kellerjordan.github.io/posts/muon/>.
 - [14] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for LLM training. arXiv:2502.16982, 2025.
 - [15] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.
 - [16] Shengding Hu et al. MiniCPM: Unveiling the potential of small language models with scalable training strategies. In *First Conference on Language Modeling (COLM)*, 2024.
 - [17] Alexander Hägele, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro Von Werra, and Martin Jaggi. Scaling laws and compute-optimal training beyond fixed training durations. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. doi: 10.52202/079017-2427.
 - [18] Aakanksha Chowdhery et al. PaLM: Scaling language modeling with Pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.
 - [19] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. arXiv:2001.08361, 2020.
 - [20] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *Proceedings of Machine Learning and Systems (MLSys) 5*, 2023.
 - [21] NVIDIA. NVIDIA H100 Tensor Core GPU datasheet. Product datasheet, 2023. URL <https://resources.nvidia.com/en-us-gpu-resources/h100-datasheet-24306>. BF16 dense 989 TFLOPS (1,979 with sparsity); SXM TDP up to 700 W.
 - [22] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564. USENIX Association, 2021.
 - [23] Jason Ansel et al. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proc. 29th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Vol. 2 (ASPLOS ’24)*, pages 929–947, 2024. doi: 10.1145/3620665.3640366.
 - [24] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. doi: 10.52202/079017-2193.
 - [25] PyTorch Team. Current and new activation checkpointing techniques in PyTorch. PyTorch Blog, 2025. URL <https://pytorch.org/blog/activation-checkpointing-techniques/>.
 - [26] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proc. 3rd ACM SIGPLAN Int. Workshop on Machine Learning and Programming Languages (MAPL ’19)*, pages 10–19, 2019. doi: 10.1145/3315508.3329973.
 - [27] Andrew Or, Apurva Jain, Daniel Vega-Myhre, Jesse Cai, Charles David Hernandez, Zhenrui Zhang, Driss Guessous, Vasily Kuznetsov, Christian Puhersch, Mark Saroufim, Supriya Rao, Thien Tran, and Aleksandar Samardžić. TorchAO: PyTorch-native training-to-serving model optimization. In *ICML 2025 Workshop on Championing Open-source DEvelopment in ML (CODEML)*, 2025.
 - [28] Changxin Tian, Kunlong Chen, Jia Liu, Ziqi Liu, Zhiqiang Zhang, and Jun Zhou. Towards greater leverage: Scaling laws for efficient mixture-of-experts language models. In *International Conference on Learning Representations (ICLR)*, 2026.
 - [29] Jan Ludziejewski, Jakub Krajewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebia, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, Marek Cygan, and Sebastian Jaszczur. Scaling laws for fine-grained mixture of experts. In *Proc. 41st International*

- Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 33270–33288, 2024.
- [30] NVIDIA. Transformer Engine. GitHub repository, 2026. URL <https://github.com/NVIDIA/TransformerEngine>. Version used: 2.19.0+5e52befd (NGC PyTorch 26.09); accessed 2026-10-11.
 - [31] Llama Team, AI @ Meta. The Llama 3 herd of models. arXiv:2407.21783, 2024.
 - [32] Ziheng Jiang, Haibin Lin, Yinmin Zhong, et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.
 - [33] Dennis Liu et al. MoE parallel folding: Heterogeneous parallelism mappings for efficient large-scale MoE model training with Megatron Core. arXiv:2504.14960, 2025.
 - [34] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
 - [35] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '20)*, 2020. doi: 10.1109/SC41405.2020.00024.
 - [36] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021. doi: 10.1145/3458817.3476205.
 - [37] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, Fang Yang, Yuanbo Peng, Dian Jiao, Shuaipeng Li, Jinbao Xue, Yangyu Tao, and Bin Cui. MEMO: Fine-grained tensor management for ultra-long context LLM training. *Proceedings of the ACM on Management of Data*, 3(1), 2025. doi: 10.1145/3709703. SIGMOD 2025.
 - [38] NVIDIA. CPU offloading (Transformer Engine user guide). Online documentation, 2026. URL https://nvidia.github.io/TransformerEngine/features/other_optimizations/cpu_offloading/cpu_offloading.html. Accessed 2026-10-11.
 - [39] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. MegaBlocks: Efficient sparse training with mixture-of-experts. In *Proceedings of Machine Learning and Systems (MLSys) 5*, 2023.
 - [40] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 formats for deep learning. arXiv:2209.05433, 2022.
 - [41] Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger Kernel: Efficient Triton kernels for LLM training. arXiv:2410.10989, 2024.
 - [42] Erik Wijmans, Brody Huval, Alexander Hertzberg, Vladlen Koltun, and Philipp Krähenbühl. Cut your losses in large-vocabulary language models. In *International Conference on Learning Representations (ICLR)*, 2025.
 - [43] Xinyu Lian, Masahiro Tanaka, Olatunji Ruwase, and Minjia Zhang. SuperOffload: Unleashing the power of large-scale LLM training on superchips. In *Proc. 31st ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '26)*, Vol. 1, pages 249–264, 2026. doi: 10.1145/3760250.3762217.
 - [44] Xinyu Lian, Minjia Zhang, Masahiro Tanaka, and Olatunji Ruwase. SuperOffload: Unleashing the power of large-scale LLM training on superchips. PyTorch Blog, 2025. URL <https://pytorch.org/blog/superoffload-unleashing-the-power-of-large-scale-llm-training-on-superchips/>. Posted 9 Oct 2025.

A Reproducibility Information

Code. The released code lives in the `apex3/` directory of <https://github.com/DW-dev-UE/LLM-from-scratch> at commit `1d491e2`.

The 62 training files (`apex3/**/*.py` and `scripts/*.py`) have identical md5 hashes on instances A and B (checked read-only on 2026-10-11 and 2026-10-12), and B also holds `scripts/router_diag.py`, which training never imports. These server files produced the step 10 benchmark and the real-training numbers after 10B tokens. The released code is the same code with comments and docstrings stripped, so its md5 hashes differ, but with docstrings excluded every Python file parses to the same syntax tree. No

Table 6: Environment of the reported runs.

Item	Value
Container	<code>nvcr.io/nvidia/pytorch:26.09-py3</code> , digest <code>sha256:6e8ccc607fc2a51e3741667b86316a0889418ba8b78ed0ba96e8d282b648a7a6</code> . The run command is <code>docker run --gpus all --ipc=host --cap-add SYS_NICE --ulimit memlock=-1 --ulimit stack=67108864</code> , with code, data, and checkpoints attached via bind mounts. Commands are executed with <code>docker exec</code>
Host	GH200 480GB, 64 of 72 Grace cores visible, enforced power limit 700 W; Ubuntu 22.04.5, kernel 6.8.0-1046-nvidia-64k; driver 580.105.08 and the image’s CUDA 13.4 forward-compatibility <code>libcuda</code> 615.71.09
Software	Container Ubuntu 24.04.4, Python 3.12.3, GCC and g++ 13.3.0, ninja 1.13.2; PyTorch 2.14.0a0+b2c75dd062.nv26.09 (CUDA 13.4, cuDNN 9.26, NCCL 2.31.2), Triton 3.8.0, torchao 0.19.0+git4ee58b57, NumPy 2.1.0; Transformer Engine 2.19.0 (used only in the failed attempts of Table 4)
FlashAttention-3	<code>flash_attn_3</code> 3.0.0 from Dao-AILab/flash-attention, commit 47e91f1f7dd8a22649cee0dd9a182b69ffe782ef, directory <code>hopper/.server/build_fa3.sh</code> sets <code>FLASH_ATTENTION_DISABLE_X=TRUE</code> for X in FP16, FP8, HDIM64, HDIM96, HDIM192, HDIM256, SPLIT, PAGEDKV, APPENDKV, SOFTCAP, and SM80, sets <code>MAX_JOBS=24</code> , and then runs <code>pip wheel --no-build-isolation --no-deps</code> . The resulting wheel retains only BF16, head dimension 128, and causal or sliding-window varlen attention. Wheel sha256 549de4ff3e88f673191008496d1b51c0d9ae43dc625a1a02288ec019d425e2c5
Variables set	<code>PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True</code> (all runs); <code>TORCH_LOGS=recompiles,graph_breaks</code> and <code>PYTHONPATH</code> (benchmarks); <code>APEX_SET</code> , <code>APEX_MICRO_BS</code> , <code>APEX_ACT_BUDGET</code> , <code>APEX_MAIN_EXTRA</code> , <code>APEX_SKIP_SMOKE</code> (training launcher; values are expanded in the commands below). <code>TORCHINDUCTOR_*</code> and <code>TRITON_*</code> variables are not set, and neither is <code>OMP_NUM_THREADS</code>
Image defaults that matter	<code>TORCH_ALLOW_TF32_CUBLAS_OVERRIDE=1</code> , so FP32 matmuls such as the router run in TF32 (the code does not change this value); <code>TORCHINDUCTOR_LOOP_ORDERING_AFTER_FUSION=0</code>

code snapshots survive for steps 0–6, and since the released code always includes the static MoE and the step 8a graph-break fix, flags can only approximate steps 0–7 (Table 7). Table 6 lists the environment as read inside the container, and the installation steps are in `server/b_setup.sh`.

C++ OpenMP kernel. With `--offload-accum cpu`, `apex3/offload.py` builds a single inline C++ source with `torch.utils.cpp_extension.load_inline` on first use (extension name `apex3_bf16_accumulate`). It compiles with `-O3 -fopenmp -mcpu=native` (Neoverse-V2 on the Grace host) and without `-ffast-math`, links with `-fopenmp`, and needs `ninja` and `g++`; PyTorch adds `-std=c++20 -fPIC`. The kernel converts BF16 gradients to FP32 and either stores them (first micro-batch) or adds them to the FP32 accumulation buffer, running under `#pragma omp parallel for schedule(static) num_threads(threads)`. With `threads = os.cpu_count()//2`, 64 exposed cores give 32 threads. The code also calls `torch.set_num_threads(os.cpu_count()-2)`, while `OMP_NUM_THREADS` is left unset and has no effect on the kernel. If the build fails, the code only prints a warning (“CPU accumulation kernel not built ...; using torch add_ (slower)”) and falls back to `torch add_`, the slow path behind the throughput divergence (Section 5.2), so check the logs for that warning.

Compilation and activation memory budget. `--compile-blocks` wraps each block’s forward in `torch.compile` with default settings: Inductor and the default mode, with `fullgraph` and `dynamic` untouched (Block in `apex3/model/transformer.py`). `--act-budget F` sets `torch._functorch.config.activation_memory_budget` and requires `--compile-blocks --recompute none`, and `--gemm-offload hooks` registers the marking pass as `torch._inductor.config.joint_custom_pre_pass`. From step 8a on, benchmark logs written with `TORCH_LOGS=recompiles,graph_breaks` should show no graph breaks. Also look for “Triton CE unavailable” in the logs; if it appears, the torch cross-entropy path ran instead of `--ce-impl triton`.

Selective activation storage. `--gemm-offload hooks --act-offload 72` keeps the FFN gate/up GEMM outputs in a 72 GiB pinned CPU storage region instead of recomputing them, and it also requires `--offload`, `--compile-blocks`, and `--recompute none`. `ActivationOffload` (`apex3/act_offload.py`) wraps each micro-batch’s forward in saved-tensor hooks and moves a tensor to the CPU region only if it is a BF16 matrix whose last dimension is an FFN width (`selector` in `apex3/gemm_offload.py`), is at least

1 MiB, and still fits in the region. Offsets are 512-byte aligned. The forward pass waits when more than 4 GiB of originals are awaiting copy, and the backward pass prefetches one layer ahead; these values are constructor defaults or call arguments, not flags. The marking pass also covers `aten._scaled_mm`, `aten._scaled_grouped_mm`, and their `_v2` variants, so FP8 gate/up outputs are stored too. This extension has no flag and is always on in the released code, and it is what separates step 10 from step 9: the step 9 code marked only `aten.mm` and `aten._grouped_mm`. To reproduce step 9, delete the loop in `_gemm_ops()` of `apex3/gemm_offload.py` that adds the four `_scaled_*` operators.

FP8 settings. `--precision fp8_rowwise` turns on the torchao `Float8Linear` path; the name is left over from an earlier version. `--fp8-recipe tensorwise` selects per-tensor scaling through `Float8LinearConfig.from_recipe_name`. Unless `--fp8-fast-accum` is passed (we never passed it), `apex3/model/fp8.py` sets `Float8GemmConfig(use_fast_accum=False)` for the forward output GEMM.

Commands. The launcher `server/train_gh200.sh` appends flags, so `--act-budget`, `--precision`, and `--keep-ckpts` appear more than once and `argparse` takes the last value. The block below is the command in effect after 10B tokens. It was launched after dropping the host page cache, and the peer host, key, and directories have been generalized. `PYTHONPATH=.` in both commands is for the public repository layout; on the servers, training ran from an installed package and the benchmark used `PYTHONPATH=$PWD`. `--seq-len 8192` and `--seed 0` (initialization and data order) are defaults. The Stage 1 mixture is code 0.35, web 0.40, math 0.06, papers 0.06, PDF 0.05, medical 0.03, textbooks 0.02, QA 0.02, and books 0.01. The arithmetic probe runs every 50 steps inside the training loop, so its time counts toward the logged wall-clock MFU.

```
PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True PYTHONPATH=. \
numactl --membind=0 torchrun --nproc_per_node 1 scripts/train.py \
  --config C --eos-id 203776 --data NAME=WEIGHT:GLOB ... --val-data NAME=GLOB ... \
  --set n_experts=64 expert_ffn=1280 top_k=4 router_bias_rate=1e-3 \
    router_bias_update=proportional seq_aux_alpha=1e-3 \
  --offload --offload-accum cpu --compile-blocks --recompute none --act-budget 0.15 \
  --moe-chunk 0 --micro-bs 6 --attn fa3 --ce-impl triton --ce-chunk 4096 \
  --gemm-offload hooks --act-offload 72 --precision fp8_rowwise --fp8-recipe tensorwise \
  --total-tokens 1.1e12 --stop-at-tokens 0.51e12 --ckpt-dir CKPT --ckpt-every 1e9 \
  --keep-ckpts 2 --arith-probe PROBE --arith-every 50 \
  --diloco-workers 2 --diloco-rank R --diloco-dir DIR --diloco-peers PEER --diloco-key KEY
```

The step 10 benchmark command is the base flag line of `server/bench_plan.sh` plus the step 10 flags, with that line's `--act-budget 0.05` changed to 0.15. The benchmark uses `router_bias_rate=0.01`, while training used `1e-3`.

```
PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True PYTHONPATH=. \
TORCH_LOGS=recompiles,graph_breaks \
numactl --membind=0 torchrun --nproc_per_node 1 scripts/bench_mfu.py --config C \
  --set n_experts=64 expert_ffn=1280 top_k=4 router_bias_rate=0.01 \
    router_bias_update=proportional \
  --offload --offload-accum cpu --compile-blocks --recompute none --act-budget 0.15 \
  --moe-chunk 0 --micro-bs 6 --attn fa3 --accum 8 --ce-impl triton --ce-chunk 4096 \
  --steps 4 --warmup 2 --gemm-offload hooks --act-offload 72 \
  --precision fp8_rowwise --fp8-recipe tensorwise --json OUT.jsonl
```

Because the benchmark draws random tokens from a fixed seed and makes each row one document (`--doc-len N` splits the rows), it needs no corpus; the 2.2 TB training corpus is not released. The step 10 record shows `micro_s_median 2.078s`, `mfu_at_batch 0.5074`, `offloaded_tensors 118`, `offloaded_gib 56.4`, and `overflow_gib 0`.

Per-step flags. Table 7 gives the flags and benchmark record for each step. Applying the step 8a fix by itself drops the MFU of the step 7 flags from 42.2% to 41.2%. Steps 3 and 4 used `torch.add_`, a path the released code takes only when the kernel build fails. Rerunning step 8 in BF16 with the released code gave 45.84% on instance B, against 45.7% in Table 3. The records do not store `--steps` and `--warmup`; `bench_plan.sh` uses 4 and 2.

Caveats for reproduction. Linux exposes the GH200's HBM as NUMA node 1. While we were decompressing data, the page cache on that node grew to 85.7 GB, and `nvidia-smi` reported the HBM as in use with no process running. Drop the page cache before each run and launch every process under

Table 7: Benchmark flags and records by step. All rows use `--config C --offload`. Row 0 lists the baseline flags, and later rows list only the changed flags. The router settings, `--steps`, and `--warmup` appear only in the step 10 command in the text. Sources are files and line numbers in `/ckpt` on instance A (B for steps 9 and 10).

Step	Flags	Exactness	Source
0	<code>--set n_experts=128 expert_ffn=640 top_k=8</code> <code>--micro-bs 8 --recompute full --compile</code> <code>--moe-chunk 0 --attn fa2 --offload-accum</code> <code>gpu --accum 1</code>	Approximate	<code>bench.jsonl:5</code>
1	<code>--compile-blocks</code> instead of <code>--compile</code>	Approximate	<code>bench.jsonl:16</code>
2	<code>--recompute none --act-budget 0.1</code> <code>--micro-bs 4</code>	Approximate	<code>bench.jsonl:19</code>
3	<code>--offload-accum cpu</code>	Approximate, <code>torch add_</code>	<code>bench.jsonl:21</code>
4	<code>--attn fa3</code>	Approximate, <code>torch add_</code>	<code>bench.jsonl:26</code>
5	<code>--accum 8</code> (C++ kernel built automatically)	Approximate	<code>bench_kern.jsonl:1</code>
6	<code>--set n_experts=64 expert_ffn=1280 top_k=4</code>	Approximate	<code>bench_kern.jsonl:4</code>
6'	<code>--micro-bs 6 --act-budget 0.05</code>	Approximate	<code>bench_kern.jsonl:6</code>
7	<code>--ce-impl triton --ce-chunk 4096</code>	Approximate	<code>bench_mfu2_</code> <code>10071230.jsonl:3</code>
8a	None (graph-break fix in <code>apex3/model/attention.py</code>)	Code change, always on	<code>bench_fp8_</code> <code>10081759.jsonl:5</code>
8	<code>--gemm-offload hooks --act-offload 72</code> <code>--act-budget 0.1</code>	Exact	<code>bench_decide_</code> <code>10081829.jsonl:3</code>
9	<code>--precision fp8_rowwise --fp8-recipe</code> <code>tensorwise --act-budget 0.15</code>	Requires the step 9 marking list (see text)	<code>bench_plan_</code> <code>10090423.jsonl:2</code>
10	Same flags as step 9	Exact (released code)	<code>bench_plan_</code> <code>10101907.txt, summary</code>

`numactl --membind=0`; pinning more memory than node 0 has free spills into node 1 and ends in GPU out-of-memory errors. The container needs `--cap-add SYS_NICE` because the default seccomp profile blocks `set_mempolicy`, and with `--cpuset-mems 0` the GPU is no longer visible. We pin host memory by calling `cudaHostRegister` on NumPy-allocated buffers (`pinned_empty` in `apex3/offload.py`) instead of using `tensor.pin_memory()`, which rounds sizes up to a power of two and would turn a 78.2 GiB buffer into 128 GiB. Unregister memory before freeing it. Otherwise re-registering the same address fails with `cudaErrorHostMemoryAlreadyRegistered`, and the error surfaces only at a later kernel launch. The default distributed-checkpoint writer (`single_file_per_rank=True`) duplicated views of the flat buffers and pushed RSS to 410 GiB, so `apex3/checkpoint.py` writes one file per tensor instead (161 GiB, 1 081 files, about 2.5 minutes).

B Evidence Files

Table 7 gives the file and line of each benchmark record (JSON Lines, one run per line). Two bundles dated 2026-10-10 hold the records along with per-file sha256 hashes: A is `apex3_evidence_A_20261010.tar.gz` (sha256 `1906f5b1218f3806de6faf09b1043870ded71992a34598f8cde2f91f45ab0f6d`), and B is `apex3_evidence_B_20261010.tar.gz` (sha256 `3ae89807320d3067d109b505ffc7586523837483014d77e9320260963204c3d0`). Both are in `paper/evidence/` of the repository. The step 10 results, training logs, extracted files under `data/`, figure scripts, and the document-length FLOPs script (`paper/repro/doclen_flops.py` and its result JSON) will be released together with the code at <https://github.com/DW-dev-UE/LLM-from-scratch/tree/main/apex3>.