Large Scale AI Engineering Project: CPU Offloading and CUDA Graphs*

Youssef Boughizane † Ines Bouchama Elvan Konukseven Ozan Solmaz

1. Project Overview

This report covers two investigations on GH200 hardware. In **Part I** we explore activation offloading to CPU memory, leveraging the high-bandwidth connection between the Grace CPU and Hopper GPU available on GH200 systems, as an alternative to activation recomputation (gradient checkpointing) for a memory-constrained 23B model. We compare activation offloading, recomputation, and combinations of both strategies. Our key finding is that combining pipeline parallelism (PP=4) with a hybrid memory strategy—offloading attention-side activations (`core_attn`, `qkv_linear`, `attn_proj`, `attn_norm`, `mlp_norm`) to CPU while selectively recomputing `layernorm` and `mlp`—achieves the highest MFU of $\sim$44.2%, outperforming all tensor-parallel and pure recomputation baselines.

In **Part II** the goal is to determine whether CUDA Graphs are a useful optimization for our Megatron-LM training setup on GH200. CUDA Graphs reduce CPU-side launch overhead by recording a static sequence of CUDA work and replaying it with fewer CPU dispatches. We find that CUDA Graphs are approximately neutral for the 8B target (iteration time $\approx$24 s, $\approx$500 TFLOP/s/GPU), while smaller 350M and 760M models at MBS=1 show modest gains of 2–5%. Megatron-local full-iteration capture causes a 25–32% regression and should be avoided.

Our scripts, logs and plots are shared at `https://github.com/youssef62/lsaie-ss26-gipfelsturm`.

2. Implementation

2.1. Setup & Methods: Part I

We focus on the memory-constrained scenario with a 23B GPT2 model (40 layers, hidden dimension 7168, FFN dimension 20480, 56 attention heads, 8 KV heads) on a single GH200 node. Training this model traditionally requires about 46 GB for the model weights (FP16), 46 GB for the gradients (FP16), 92 GB for the master weights (FP32), and 184 GB for the optimizer states. In addition, activations with tensor parallelism (TP) = 4 require roughly 18 GB per GPU. Altogether, this amounts to approximately 104 GB per GPU, which exceeds the 96 GB memory capacity of our GH200 GPUs. Making this task unfeasable with a single GH200 node using only GPU memory.

Megatron-LM [1–3] handles CPU activation offloading and activation recomputation through the following built-in flags (see Table A in the appendix for the offloaded/recomputed modules, backing implementation, and compatibility constraints):

- `--cpu-offloading-num-layers N`: coarse, layer-level CPU offload via TransformerEngine. PP-incompatible.
- `--fine-grained-activation-offloading`: per-submodule CPU offload via Megatron's `PipelineOffloadManager`. PP-compatible.
- `--recompute-granularity selective`: per-submodule recompute via `CheckpointWithoutOutput`.
- `--recompute-granularity full`: recomputes everything from layer inputs.

The two offload flags are mutually exclusive, and neither recompute flag composes with `--cpu-offloading-num-layers`.

For interpretability, we report results in terms of MFU (model flops utilization):

$$\text{MFU} = \frac{T \times F_{\text{model}}}{P_{\text{hw}}} \tag{1}$$

where T is the actual throughput (toks/s/GPU), F_{model} is the model FLOPs per token, and P_{hw} is the hardware peak FLOPs. where Hardware Peak FLOPs/s = 989 TFLOPS (GH200 dense BF16 peak throughput).

2.2. Setup & Methods: Part II

We use the same 8B model as Experiment 1 (32 layers, hidden dimension 4096, FFN dimension 14336, 32 attention heads, 8 KV heads) trained with Megatron-LM and TransformerEngine on one GH200 node using TP=PP=1 and DP=4, Global batch size = 256, Micro-batch size = 2 and BF16 weights.

We evaluated both TransformerEngine-level and Megatron-local graph capture. The TransformerEngine

*This work was done as part of the Large-Scale AI Engineering course of ETH Zurich.

†The ordering is kept same as in the project proposal.

path captures selected layer regions, while the Megatron-local path attempts to capture a larger full-iteration region:

- `--cuda-graph-impl transformer_engine`: TransformerEngine CUDA Graph capture using the default TE scope (full-layer capture per TE transformer layer).
- `--cuda-graph-impl transformer_engine --cuda-graph-scope attn`: capture only attention sublayers.
- `--cuda-graph-impl transformer_engine --cuda-graph-scope mlp`: capture only MLP sublayers.
- `--cuda-graph-impl local --cuda-graph-scope full_iteration`: Megatron-local full-iteration graph capture.

The primary metric is average tokens/sec/GPU over the final logged iterations. The single best iteration is useful as an upper-envelope sanity check, but we base decisions on the steady-state average, not the best observed point.

3. Experiments

3.1. Experiment 1: Parallelism Strategy Sweep (8B)

Before addressing the memory constraints of the 23B model, we first profile different parallelism strategies on an 8B transformer model. This model fits entirely in GPU memory without requiring any memory reduction technique, and therefore provides a clean throughput baseline. We sweep:

$$\{(\mathrm{TP}, \mathrm{PP}, \mathrm{DP})\} \in \{(4,1,1),(1,4,1),(1,1,4),\\ (2,2,1),(2,1,2),(1,2,2)\}$$

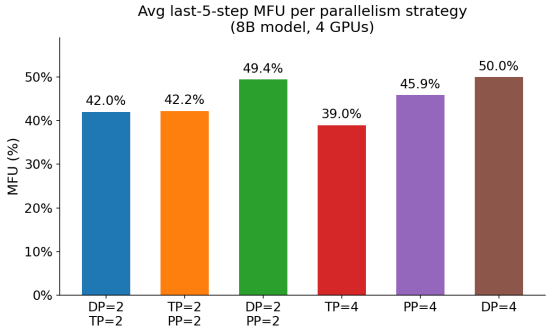


Figure 1. MFU across parallelism configurations on the 8B model trained for 50 steps.

DP achieves the highest MFU, as it requires no inter-GPU communication during the forward and backward passes—only a single gradient all-reduce between steps.

More interestingly, among the two model parallelism strategies, PP significantly outperforms TP.

On the effectiveness of PP. A natural first hypothesis is communication volume: TP requires 4 all-reduce operations per layer (2 in the forward pass and 2 symmetrically in the backward pass), totalling $4L = 160$ synchronization points per step. PP=4 by contrast requires only point-to-point transfers at stage boundaries, with no collective operations.

However, on GH200 where GPUs are connected via NVLink, one might expect this communication to be cheap enough to close the gap. To investigate, we profile both runs with the PyTorch profiler.

The kernel breakdowns (Figure 2) reveal the true cost. For TP (Figure 2a), `ncclDevKernel_AllReduce` is the single largest kernel at **22.6% of total kernel time**—meaning more than one in five GPU cycles is spent on all-reduce collectives. Crucially, these appear under kernel time in the summary profiler, not under communication, because NVLink collectives execute as CUDA kernels on the GPU. For PP (Figure 2b), no NCCL kernel appears in the top 10—all kernel time is useful forward and backward compute, dominated by matrix multiplication kernels (`nvjet_sm90_tst_256x128`).

This shows that TP's underperformance on GH200 is not a bandwidth problem but a *kernel occupancy problem*: all-reduce collectives consume a substantial fraction of GPU compute time regardless of interconnect speed, leaving less time for actual model computation. These results motivate our use of PP in Experiment 3.

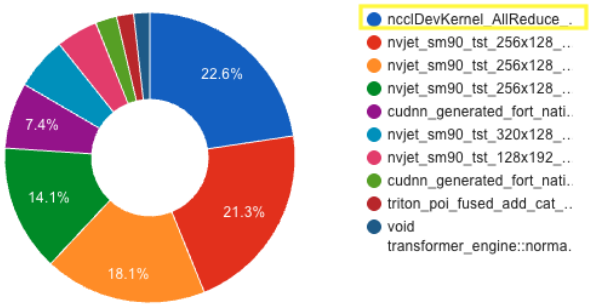
3.2. Experiment 2: Offloading vs. Recomputation across Micro-Batch Sizes

Since `--cpu-offloading-num-layers` is incompatible with pipeline parallelism and the model is too large for data parallelism alone, all runs in this experiment use TP=4. We compare two memory strategies across micro-batch sizes MBS $\in \{1, 2\}$:

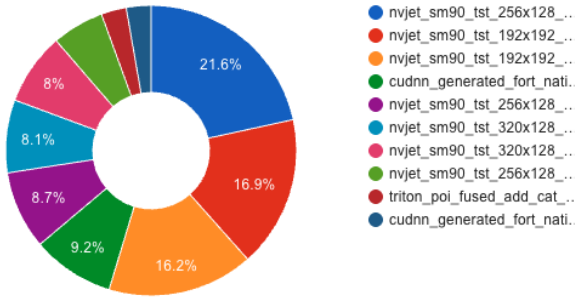
- **Full offloading**: `--cpu-offloading-num-layers 39`, offloading activations of all but the last transformer layer to CPU.
- **Full recompute**: `--recompute-granularity full`, storing only layer inputs and recomputing all activations during the backward pass.

Offloading consistently outperforms recomputation, and achieves $\sim 39\%$ MFU—a reasonable efficiency level given the memory constraints. This validates the Grace-Hopper high-bandwidth interface as an effective alternative to recomputation.

The marginal MFU gain from MBS=1 to MBS=2 is small for both strategies, suggesting the GPU is already compute-bound at MBS=1. In this regime, recomputation



(a) Kernel breakdown, TP=4.



(b) Kernel breakdown, PP=4.

Figure 2. Kernel breakdowns for TP=4 vs. PP=4 on the 8B model.

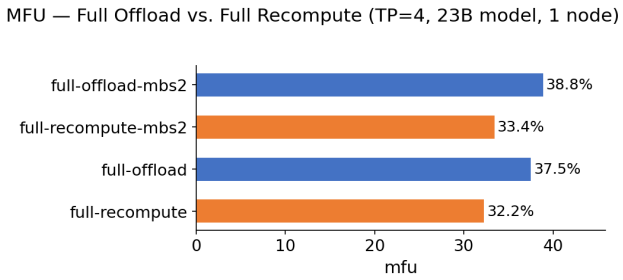


Figure 3. MFU for full offloading and full recomputation at MBS $\in \{1, 2\}$ on the 23B model with TP=4 trained for a maximum of 25 steps or 30 mins.

is particularly costly: the additional forward pass directly contends with useful computation for GPU cycles. Offloading instead externalises the memory pressure to the Grace-Hopper interconnect, which would otherwise sit idle during compute-heavy phases, explaining its advantage in this setting.

3.3. Experiment 3: Hybrid Offloading and Recomputation with Pipeline Parallelism

The two previous experiments yielded two key insights:

- **Insight 1**: Pipeline parallelism outperforms tensor parallelism in throughput.
- **Insight 2**: Offloading outperforms recomputation under TP=4.

Motivated by Insight 1, we turn to pipeline parallelism. However, the coarse layer-level offload flag is PP-incompatible, and the fine-grained offload flag cannot offload MLP activations on dense models (`expert_fc1` is MoE-only), so full offloading is not achievable under PP. Instead, guided by Insight 2, we offload everything that can be offloaded and recompute the rest:

- **Full recompute**:
  `--recompute-granularity full`
- **Mixed offload + recompute**: fine-grained offloading of attention-related activations,
  `--fine-grained-activation-offloading`
  `--offload-modules core_attn qkv_linear attn_proj attn_norm mlp_norm`
  with selective recomputation of `layernorm` and `mlp`,
  `--recompute-granularity selective`
  `--recompute-modules layernorm mlp`
  which is pipeline-parallelism friendly.

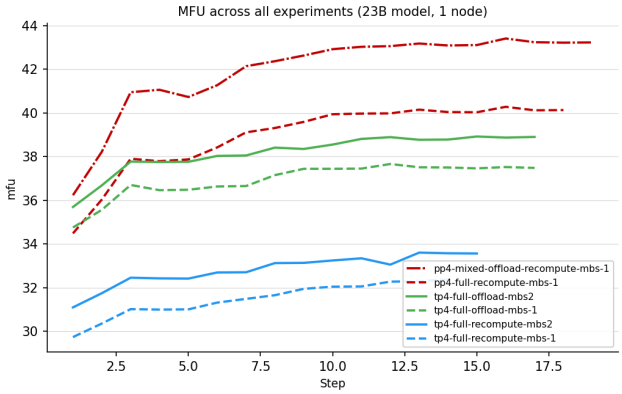


Figure 4. MFU across all experiments. The mixed offload+recompute strategy under PP=4 (pink) achieves the highest MFU at $\sim$43%, outperforming all TP=4 baselines and PP full recompute, confirming that combining PP with targeted offloading is the most effective strategy for this memory-constrained setting.

The mixed strategy achieves $\sim$43% MFU, the highest across all configurations. Notably, even PP with full recompute ($\sim$39%) surpasses the best TP=4 offloading run ($\sim$39% at MBS=2), reinforcing Insight 1. The gap between mixed and full recompute under PP quantifies the benefit of offloading attention activations even when MLP activations must be recomputed.

Table 1. Selective recomputation results (TP=4, MBS=1).

Recomputed	MLP on GPU	Fits	MFU
`core_attn`	Yes	OOM (it. 1)	$\sim$37.2%*
`core_attn+layernorm`	Yes	OOM (it. 1)	$\sim$37.6%*
`mlp`	No	Yes	$\sim$35.7%
Full (reference)	No	Yes	$\sim$32.4%

* from a single iteration before OOM.

3.4. Experiment 4: Selective Recompute Analysis (TP=4)

We use Megatron's selective recompute mode to recompute individual submodules while leaving the rest saved on GPU. All runs use TP=4, MBS=1.

Observation 1: MLP activations are the dominant memory pressure source. Both `core_attn` and `core_attn + layernorm` selective recompute OOM after a single iteration. Keeping the full MLP activation path on GPU exceeds the 96 GiB budget—MLP activations (`mlp_fc1_in`, `mlp_act_in`, `mlp_fc2_in`, $\sim$94 MB/layer combined) cannot coexist with the attention path on GPU.

Observation 2: Recomputing the MLP path is expensive. Selective MLP recompute fits but only reaches $\sim$35.7% MFU—better than full recompute ($\sim$32.4%) but still well below the no-recompute ceiling. The dense `fc1` ($h \to$ ffn) and `fc2` (ffn $\to h$) matrix multiplications are the costly part.

The two observations together imply an asymmetry: MLP activations occupy more memory than the attention path *and* are more expensive to recompute. The strategy we test in Experiment 5 is therefore: keep MLP activations resident on GPU, offload attention activations to CPU, recompute nothing.

3.5. Experiment 5: Fine-Grained Offload Without Recompute (TP=4)

Following Experiment 4, we test whether fine-grained CPU offload of attention-side activations alone is sufficient to fit the model—eliminating recomputation entirely:

```
--fine-grained-activation-offloading
--offload-modules core_attn qkv_linear
```

with *no recompute flags*. The selected attention tensors are offloaded to CPU during forward and reloaded during backward; all MLP activations remain on GPU under normal autograd.

Key finding: $\sim$43.2% MFU at MBS=1—within 1 MFU point of the best PP=4 result from Experiment 3 (44.2%), without pipeline parallelism or recompute. Fine-grained offload targets only `core_attn` and `qkv_linear` saved tensors ($\sim$46 MB/layer combined), freeing just enough GPU memory to keep the larger MLP activations ($\sim$94 MB/layer) resident, avoiding both the recompute cost and the bulk CPU transfer cost of full offload.

Table 2. Fine-grained offload results vs. baselines (TP=4). "Fine offload attn" offloads `core_attn+qkv_linear`.

Configuration	MBS	GPU mem	MFU
Fine offload attn	**1**	**$\sim$88 GB**	$\sim$43.2%
Fine offload attn	2	–	OOM
Full CPU offload	1	$\sim$77 GB	$\sim$37.5%
Full CPU offload	4	$\sim$81 GB	$\sim$39.4%

MBS=2 does not fit because doubling MBS roughly doubles activation memory. Only layer-level full CPU offload provides enough memory relief for MBS ≥ 2. Full CPU offload at MBS=4 ($\sim$39.4% MFU) shows diminishing returns consistent with Experiment 2's compute-bound conclusion.

3.6. CUDA Graph Experiments (8B)

Performance model. One training iteration can be decomposed as $t_{\text{iter}} = t_{\text{gpu}} + t_{\text{launch}} + t_{\text{comm}} + t_{\text{framework}}$, where t_{launch} is CPU-side CUDA launch overhead. CUDA Graphs target only t_{launch}; they do not reduce matmul, attention, optimizer, or NCCL time. If replay removes a fraction r of launch overhead and adds graph overhead t_{graph}, the maximum relative speedup is bounded by $t_{\text{launch}}/(t_{\text{iter}} - t_{\text{launch}})$. CUDA Graphs are therefore attractive only when $t_{\text{launch}}/t_{\text{iter}}$ is large enough to dominate t_{graph} and any extra synchronization or memory pressure introduced by capture. Graph granularity matters for the same reason: attention-only and MLP-only scopes localize capture but introduce graph/non-graph boundaries; TE full-layer captures a larger repeated block; local full-iteration in principle removes launch overhead across the whole step but interacts with optimizer, communication, and Megatron bookkeeping—the highest-risk option.

Main result. The 8B baseline already reaches approximately 10.8k tokens/sec/GPU and about 500 TFLOP/s/GPU. TransformerEngine CUDA Graph variants are approximately neutral: some runs are slightly above baseline and others are slightly below, within normal run-to-run variation.

Table 3. CUDA Graph results on the 8B model (MBS=2, DP=4).

Configuration	Avg tok/s/GPU	Δ vs. base	Conclusion
Baseline	10847	–	reference
TE full-layer	10825 / 10917	-0.2% / $+0.65\%$	neutral
TE attention	10873 / 10632	$+0.24\%$ / -1.98%	mixed
TE MLP	10705 / 10990	-1.31% / $+1.32\%$	mixed
Local full-iteration	8064 / 7376	-25.7% / -32.0%	large regression

The Megatron-local full-iteration graph result is the clearest negative result: local full-iteration capture drops to 8064 and 7376 tokens/sec/GPU, corresponding to slowdowns of approximately 25–32%. The captured region interacts with more of Megatron's iteration machinery and introduces overheads that dominate any launch savings.

Why 8B is not launch-overhead bound. Each iteration processes GBS $\times$ SEQ_LEN $= 256 \times 4096 \approx 1.05 \times 10^6$ tokens across 4 GPUs. At $\approx$10800 tokens/sec/GPU, the iteration takes $t_{\text{iter}} \approx 1.05 \times 10^6/(4 \times 10800) \approx 24.3$ s. CUDA Graphs remove CPU launch overhead, typically a few microseconds per kernel; even allowing a generous 500 ms per iteration of CPU dispatch overhead, the maximum theoretical speedup is $500\,\text{ms}/24.3\,\text{s} \approx 2.1\%$—before paying any graph replay, capture, or static-memory cost. The utilization picture agrees: 500 TFLOP/s/GPU is about half of the GH200 dense BF16 peak (989 TFLOPS), which is not the profile of a CPU-starved workload. A back-of-envelope FLOP count using the $6N$-per-token approximation gives $6\times8\times10^9\times1.05\times10^6/4 \approx 1.26\times10^{16}$ FLOPs per GPU per step, or $\approx$520 TFLOP/s—matching the logged value and confirming the iteration is dominated by real transformer compute rather than dispatch gaps. By Amdahl's law, even perfect launch-overhead elimination caps the speedup near 2%, exactly the noise band we observe for the TE variants.

Stress test: smaller low-MBS regimes. To verify that our CUDA Graph setup can show benefits when the workload is more launch-overhead sensitive, we also evaluated smaller models at MBS=1. These runs reduce work per kernel, so fixed launch overhead becomes a larger fraction of the step.

Table 4. CUDA Graph results for smaller models (MBS=1).

Model	Best graph mode	Baseline avg	Graph avg	Speedup
125M	TE attention	41271	41269	-0.00%
350M	TE MLP	40405	42565	$+5.35\%$
760M	TE attention	41750	42595	$+2.02\%$

The smaller-model results show that CUDA Graphs were not universally broken in our environment. The 350M MBS=1 run improves by 5.35% with TE MLP graph capture, and the 760M MBS=1 run improves by about 2%. These are the regimes where CUDA Graphs behave as expected: kernels are short enough that CPU launch overhead is visible, but the captured region still contains enough useful GPU work to amortize graph replay overhead. The 125M case is a useful counterexample: even though the model is small, throughput does not improve, showing that "smaller" alone is not sufficient—if the captured region is too small or other framework overheads dominate, graphing does not translate into speedup. The useful regime is a middle ground.

Engineering tradeoffs. CUDA Graphs have practical costs beyond raw throughput: (i) *static-shape requirement*—graph replay assumes stable shapes and execution paths, reducing flexibility for variable sequence length, curriculum schedules, or dynamic control flow; (ii) *memory pressure*—capture requires long-lived static buffers, which is costly for models already near the memory limit (directly relevant to Part I); (iii) *debuggability*—replay obscures where time is spent and makes failures harder to localize; (iv) *distributed interaction*—full-iteration capture can interact poorly with communication overlap, gradient reduction, and optimizer stepping, which is the most likely explanation for the local full-iteration regression; (v) *opportunity cost*—engineering time spent stabilizing a marginal speedup may be better spent on changes with larger expected payoff (memory footprint, parallelism tuning, recomputation strategy). The practical bar for adopting CUDA Graphs is therefore higher than "not slower"; for the 8B target we do not meet it.

Limitations. These results are throughput-engineering measurements rather than a microarchitectural profile. We did not directly measure CPU launch time or GPU idle gaps with Nsight Systems; the bottleneck was inferred from throughput, TFLOP/s, iteration time, and controlled ablations. A profiler trace would be the cleanest way to confirm the exact launch-overhead fraction. The observed pattern is nonetheless strong enough for the practical decision below.

4. Results

4.1. Productive GPU Memory Use Predicts Throughput

A pattern emerges across our experiments: *strategies that use the GPU memory budget for cheap-to-store, expensive-to-recompute activations achieve the highest MFU.* Crucially, it is not enough to just fill the GPU—what matters is *what* fills it.

The two best strategies (PP=4 mixed at 44.2%, TP=4 fine-grained at 43.2%) both use $\sim$85–88 GB of GPU memory. Every coarse strategy that leaves more GPU memory unused (77–81 GB) clusters below 40% MFU. But filling memory is not by itself enough: selective MLP recompute uses $\sim$85 GB and only reaches 35.7% MFU, because it spends that memory on cheap-to-recompute attention activations while paying the cost of MLP recompute.

The intuition is straightforward: every megabyte of activation memory left unused on the GPU corresponds either to a megabyte sent over the CPU$\leftrightarrow$GPU link (transfer

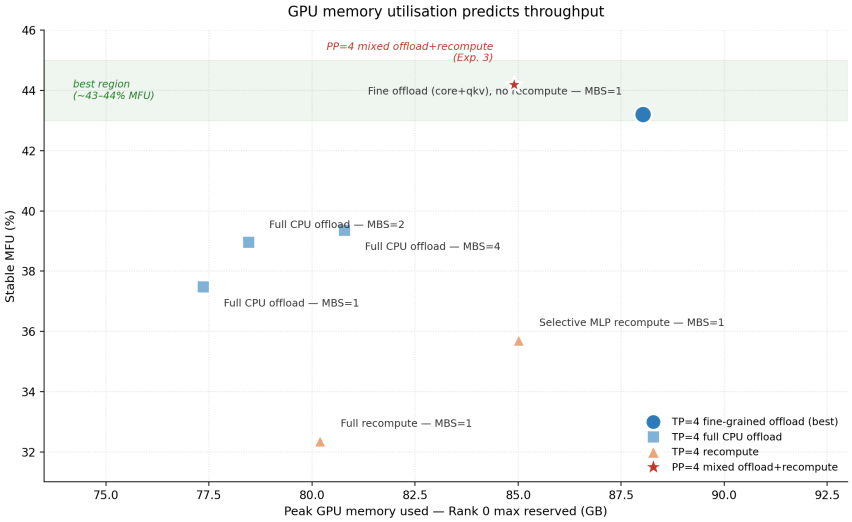


Figure 5. Stable MFU as a function of peak GPU memory used (Rank 0 max reserved). The two top strategies—TP=4 fine-grained offload and PP=4 mixed offload+recompute—both use ∼85–88 GB and both reach ∼43–44% MFU. Selective MLP recompute also uses ∼85 GB but achieves only 35.7% because it keeps the cheap-to-recompute attention path on GPU and recomputes the expensive MLP path—the wrong choice for what to retain.

Table 5. Summary of all Part I strategies (Rank 0 max reserved memory and stable MFU).

Strategy	GPU mem	MFU
PP=4 mixed offload+recompute (Exp. 3)	∼85 GB	44.2%
TP=4 fine-grained offload	**∼88 GB**	**43.2%**
TP=4 full CPU offload (MBS=4)	∼81 GB	39.4%
TP=4 full CPU offload (MBS=2)	∼79 GB	39.0%
TP=4 full CPU offload (MBS=1)	∼77 GB	37.5%
TP=4 selective MLP recompute (Exp. 4)	∼85 GB	35.7%
TP=4 full recompute (Exp. 2)	∼80 GB	32.4%

overhead) or to a recomputation step (compute overhead). Keeping the activation on GPU avoids both—*provided* the retained activation is one that would be expensive to recompute.

TP=4 fine-grained offload nearly matches PP=4 because TP shards activations across GPUs (∼448 MB/layer vs. ∼952 MB/layer under PP=4), leaving ∼17 GB of spare memory under full offload. This spare memory is what fine-grained offload exploits: it keeps the expensive-to-recompute MLP path resident on GPU, offloading only the smaller attention-side tensors needed to fit. Under PP=4, every viable strategy already fills GPU memory by construction, so PP=4 has no leftover memory to use this way.

4.2. CUDA Graphs: Engineering Decision

For the final 8B configuration, CUDA Graphs should not be enabled. The evidence is:

- TransformerEngine graph capture is approximately neutral on 8B and does not provide a reliable speedup.
- Megatron-local full-iteration capture causes a large throughput regression.
- The 8B iteration is long (≈24 s), so CPU launch overhead is a small fraction of wall-clock time.
- The 8B baseline already reaches around 500 TFLOP/s/GPU, indicating the workload is dominated by GPU-side computation rather than CPU dispatch.

CUDA Graphs are still worth reporting as a conditional optimization: they helped selected smaller low-MBS runs, especially the 350M MBS=1 TE-MLP case (+5.35%), but they are not a robust optimization for the compute-heavy 8B target. This is an important engineering lesson: an optimization can be valid for one operating point and irrelevant at another.

References

[1] 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)*, 2023. 1

[2] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[3] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. 1

A. CUDA Graph Throughput Table and Megatron Flags

Table 6. Model Performance and CUDA Graph Speedups

Model	CUDA Graph	Baseline	Graph	Speedup
125m	transformer.engine / attn	41271	41269	-0.00%
125m	transformer.engine / full-layer/default	41271	40774	-1.20%
125m	transformer.engine / mlp	41271	40777	-1.20%
350m	transformer.engine / attn	40405	40389	-0.04%
350m	transformer.engine / full-layer/default	40405	41270	2.14%
350m	transformer.engine / mlp	40405	42565	5.35%
760m	transformer.engine / attn	41750	42595	2.02%
760m	transformer.engine / full-layer/default	41750	42591	2.01%
760m	transformer.engine / mlp	41750	42528	1.86%
8b	transformer.engine / attn	10819	10873	0.50%
8b	transformer.engine / attn	10819	10632	-1.72%
8b	transformer.engine / attn mlp	10819	10670	-1.37%
8b	transformer.engine / attn mlp	10819	10789	-0.27%
8b	local / full_iteration	10819	8064	-25.46%
8b	local / full_iteration	10819	7376	-31.82%
8b	transformer.engine / mlp	10819	10705	-1.05%
8b	transformer.engine / mlp	10819	10990	1.58%
8b	transformer.engine / full-layer/default	10734	10825	0.85%
8b	transformer.engine / full-layer/default	10734	10917	1.70%
8b	transformer.engine / attn	10878	10801	-0.71%
8b	transformer.engine / attn	10878	10560	-2.93%

Table 7. Megatron built-in flags for CPU activation offloading and recomputation.

Flag	**Role & method**	**Compatibility / notes**
`--cpu-offloading-num-layers N`	Offloads inputs of *Linear* layers and the `fc1` output of the first N layers to CPU.	Incompatible with PP >0. $N \leq$ `TOTAL_NUM_LAYERS` -1. Not composable with recompute flags.
`--fine-grained-activation-offloading`	Per-submodule CPU offload of the modules listed in `--offload-modules`. Handled by `PipelineOffloadManager`.	Compatible with PP and with fine-grained recompute. Mutually exclusive with layer offloading.
`--recompute-granularity selective`	Per-submodule recompute of modules listed in `--recompute-modules`. Handled via `CheckpointWithoutOutput`.	Not composable with `--cpu-offloading-num-layers`.
`--recompute-granularity full`	Stores only layer inputs and recomputes everything during the backward pass.	Requires `--recompute-method` (`uniform`/`block`). Not composable with layer offloading.