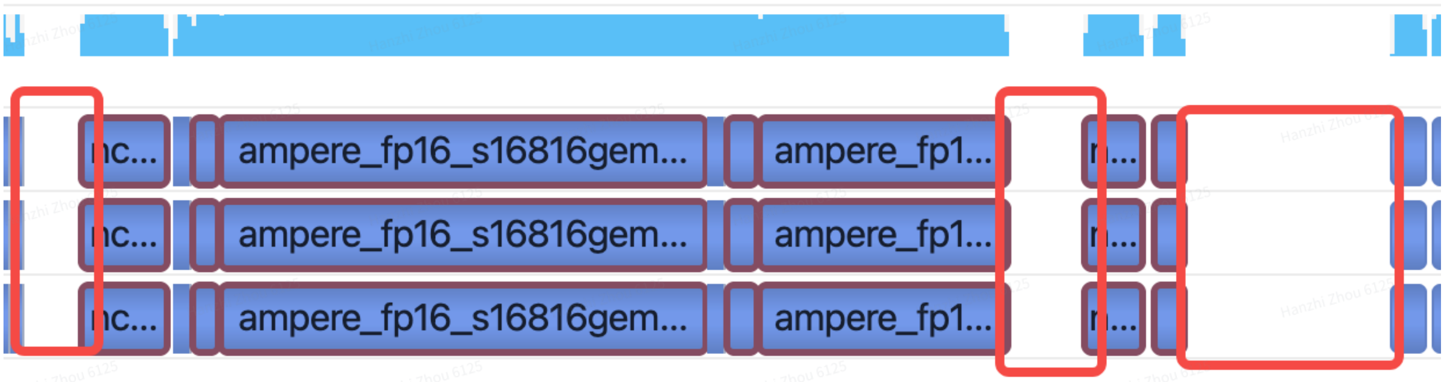
Latency-optimal allreduce and cuda graph optimization

Background

For most LLMs, LLM tensor parallel serving requires 2 allreduces per layer. During the decoding stage, the allreduce size is very small since we only decode 1 token at a time. For example, batch size 32 float16 serving of Llama 65B or Llama 2 70B only requires an allreduce size of (32 * 8192 * 2) = 512KB.

Typical implementations invoke NCCL allreduce. However, NCCL has the following problems:

1. Not **latency optimal**. NCCL implements **bandwidth optimal** tree or ring base allreduce, which have O(log n) and O(n) latency respectively. The tree method has more constant overhead and is typically not used in a single node. Given the enormous bandwidth between modern GPUs (480GB/s effective bidirectional for A100 and 1.5x for H100s), small size allreduce is **latency-bound.**
2. Unfriendly to kernel fusion. NCCL is a black box for developers and is hard to hack. If we write our own kernels, we can easily fuse subsequent elementwise operations, which are common in LLMs.
3. CUDA-graph unfriendly. NCCL's cuda graph support requires inserting synchronizing host nodes which block the GPUs, causing gaps in GPU stream, even if the stream has a lot of kernels queued.



Therefore, it is beneficial to have a custom kernel that performs allreduce with lower (preferably O(1)) latency.

Mechanism

Buffer registration

CUDA IPC support allows each rank to get hold of a pointer pointing to other GPU's memory. Therefore, during initialization step, each rank can export its buffer to an Ipc handle, and then do an allgather to collect ipc handles from all ranks. Using these pointers, each rank can simultaneously read from other GPUs.

One-shot allreduce

In one hop allreduce, each rank sends complete data to all other ranks and receives complete data from all ranks. The allreduce part is simple. The main performance trick for this part is to use custom aligned array type for loads and stores so that compiler can generate LD.128 and ST.128 instructions (load and store 16 bytes per thread per instruction)

The harder part is synchronization.

1. Allreduce can only start when all ranks have reached the allreduce kernel, or otherwise the reduce buffer may not be ready.
2. Allreduce can only end when all ranks have finished reading each other's buffer, or otherwise subsequent kernels that write to the same buffer may corrupt data.

We tried a few different ways:

1. CUDA IPC event. Each rank waits for all other ranks' events inserted before the allreduce kernel.
    a. Latency too high: Latency > 100us
2. Busy waiting on the host's shared memory
    a. Still too slow. Latency >= 10us in addition to the reduction part (3~6us)
3. In kernel synchronization
    a. This is probably as good as we can get: 8.5 ~ 9us **including the reduction part**

The details of synchronization are tricky and we suggest reading the code to understand it better. We specifically avoided using atomics to be compatible with links that don't natively support atomics (e.g. PCIe).

We think the 8.5us latency (on A100s) is optimal, because it is simply a bit more than the latency of 3 NVLink remote read/write (2.5us each)

1. 1 remote write to start sync
2. Multiple pipelined remote read for allreduce
3. 1 remote write for end sync

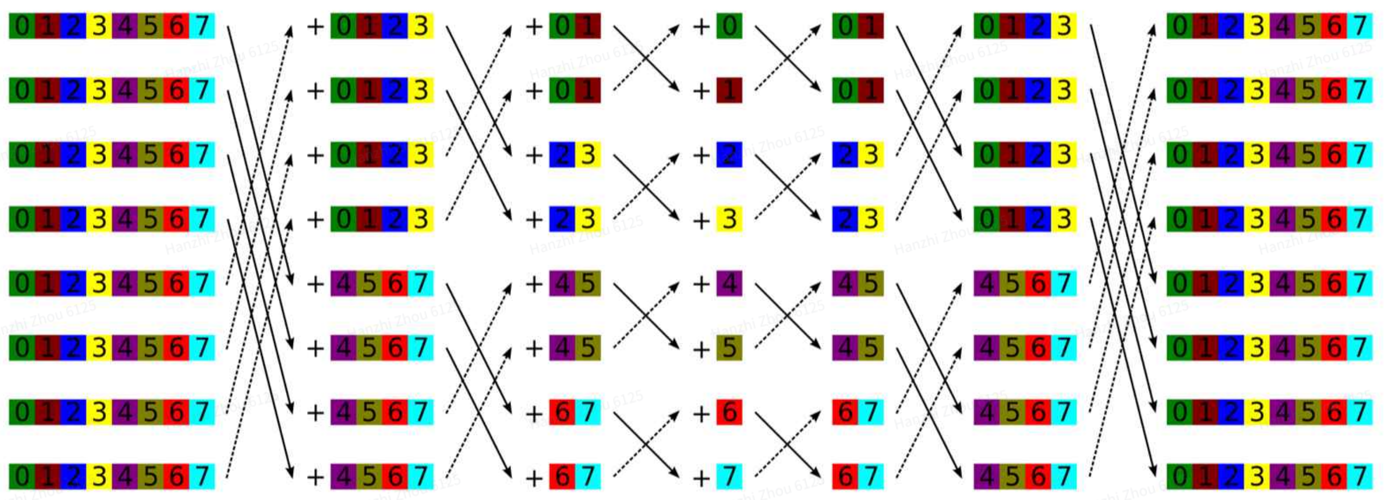
And there's probably no way to do better than this, unless we make an unsafe assumption about the later kernels so that we can elide the end sync.

Two-shot allreduce

During two shows allreduce, we first perform a reduce scatter by letting each rank reads 1/N data from all ranks and summing them. Then, we do an allgather by sending each ranks' partition to all other ranks. We use the same synchronization method described in one-shot allreduce.

Half butterfly allreduce

In PCIe topologies such as 4xA10, we observe that the all-to-all style of reduction reduces PCIe throughput significantly. The reason is that on these systems, the GPUs are directly connected to the CPUs without PCIe switches, and CPUs are often terrible PCIe switches, offering bad performance when multiple flows are present. To alleviate this problem, we designed a butterfly-style allreduce but with only half of the stages. During each stage, each rank pair exchanges complete data, thereby reducing latency by half compared to the original butterfly allreduce.



Butterfly allreduce. Note that our half butterfly allreduce only has half of the stages compared to full butterfly allreduce.

Latency and Bandwidth Analysis

N = number of nodes, P = data size per rank, L = unidirectional p2p latency

	Latency (excluding synchronization overhead)	Total communication volume (per rank)	Bandwidth optimal	Full nvlink required for >2 GPUs?
Ring	2*(N-1)*L	2(N-1/N)*P	Yes	No
One shot	L	(N-1)*P	No	Yes
Two shot	2L	2(N-1/N)*P	Yes	Yes

Half butterfly	log2(N)*L	log2(N)*P	No	No
Butterfly	2log2(N)*L - 1	2(N-1/N)*P	Yes	No

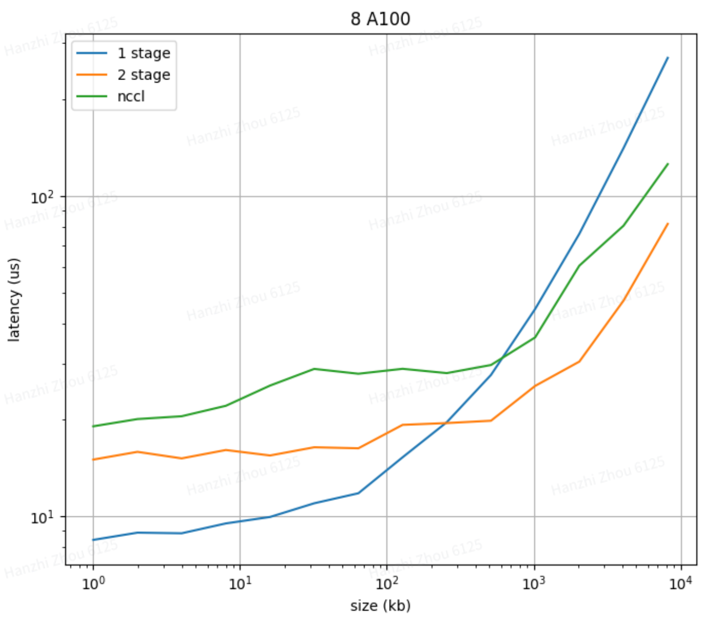
Speed vs NCCL

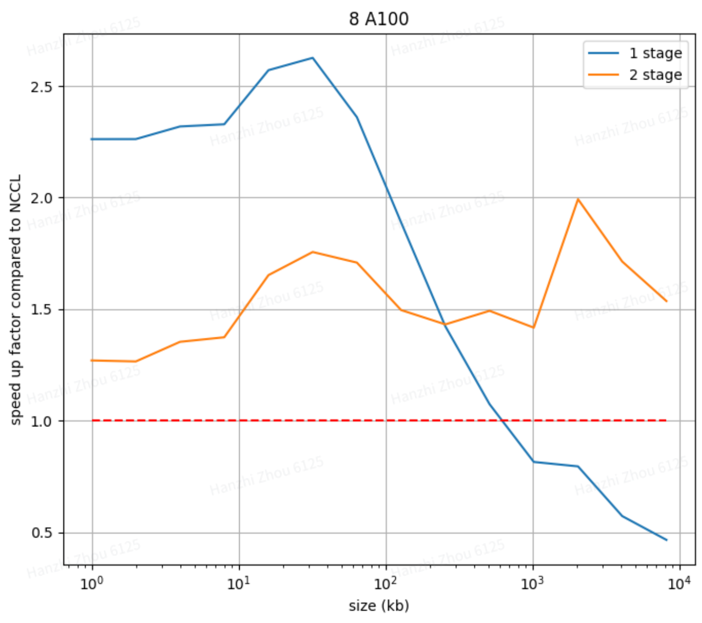
When there are only two GPUs (pcie or nvlink, doesn't matter), my one shot implementation is **faster across-the-board for all practical serving sizes.**

On nvlink fully connected systems (A100s and 4V100s), two-stage allreduce is faster than NCCL across all practical serving sizes because its bandwidth optimal and has constant latency. One hop allreduce is faster than two hop allreduce when size is small (e.g. <=512K for 4A100s and <=256k for 8A100s). Therefore, during actual application, we choose between these two kernels depending on data size.

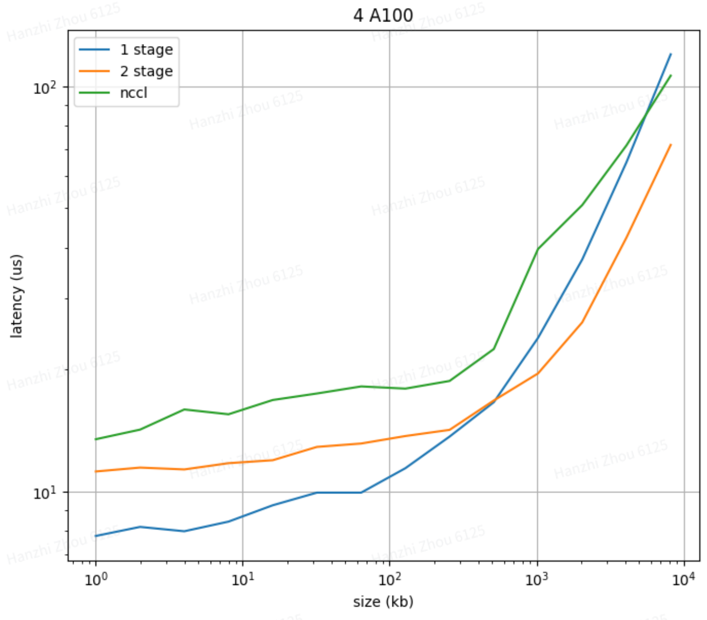
On PCIe systems such as 4A10, one shot allreduce is practically useless, being only faster than NCCL when size is <10k. Half butterfly allreduce is faster than NCCL up to 512K. This is because on 4 GPUs, half butterfly requires 2P send/recv per rank while NCCL Ring requires 1.5P send/recv per rank, so NCCL will be faster when allreduce is no longer latency bound.

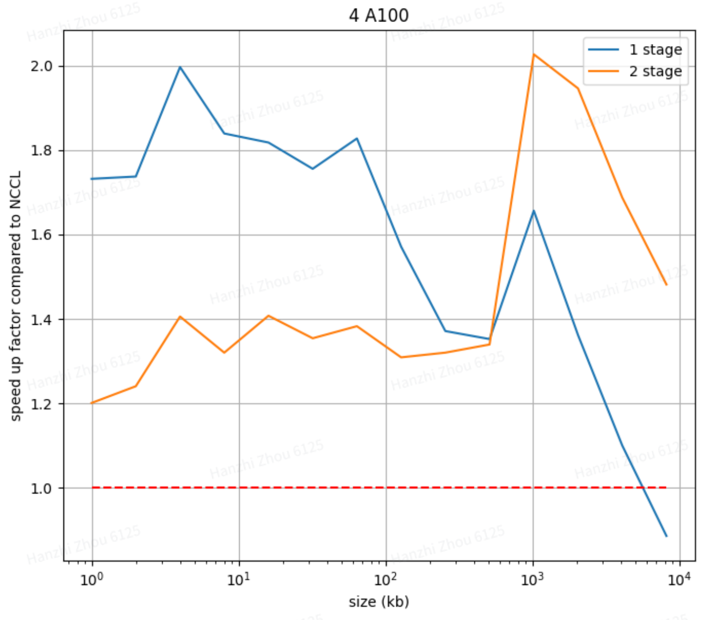
8 A100 GPUs



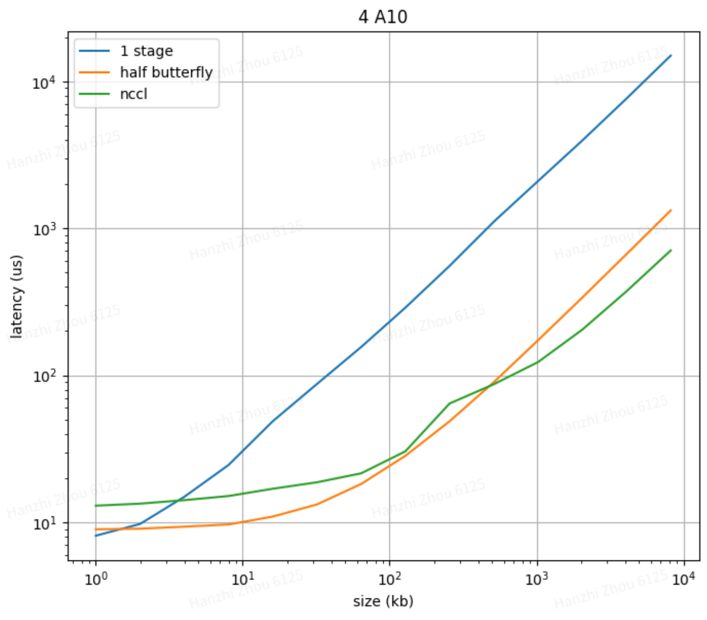


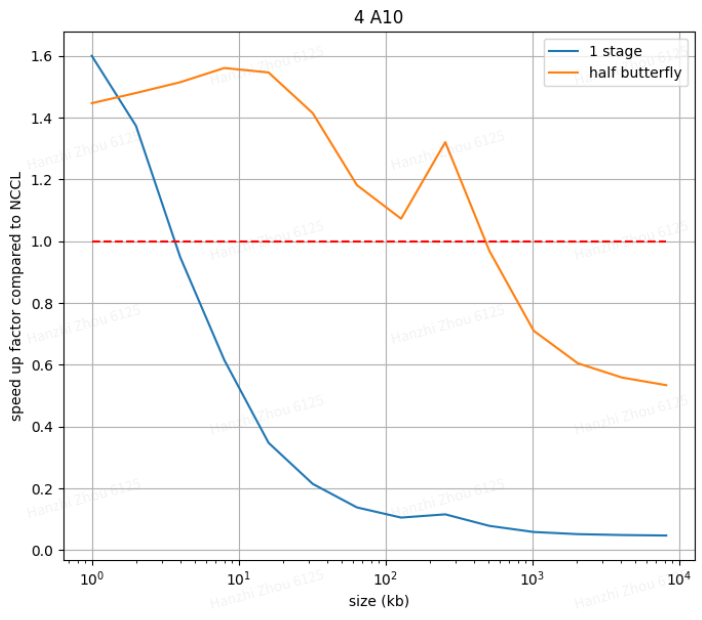
4 A100 GPUs



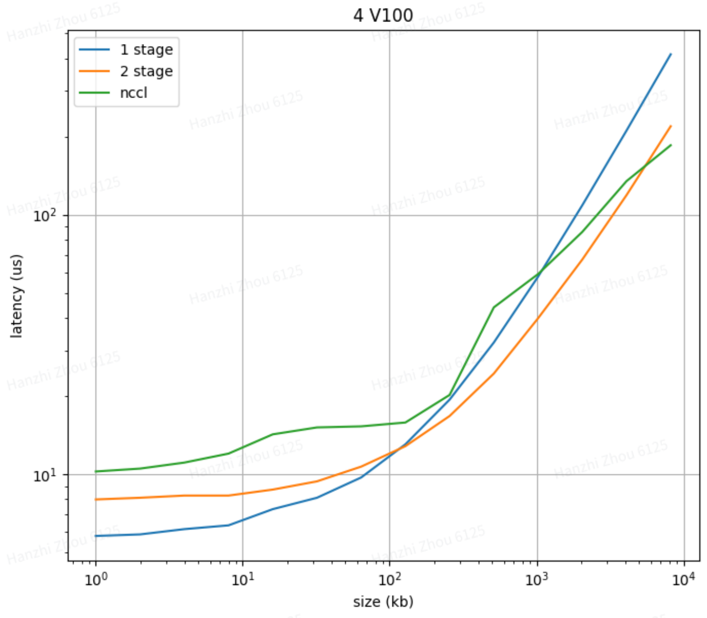


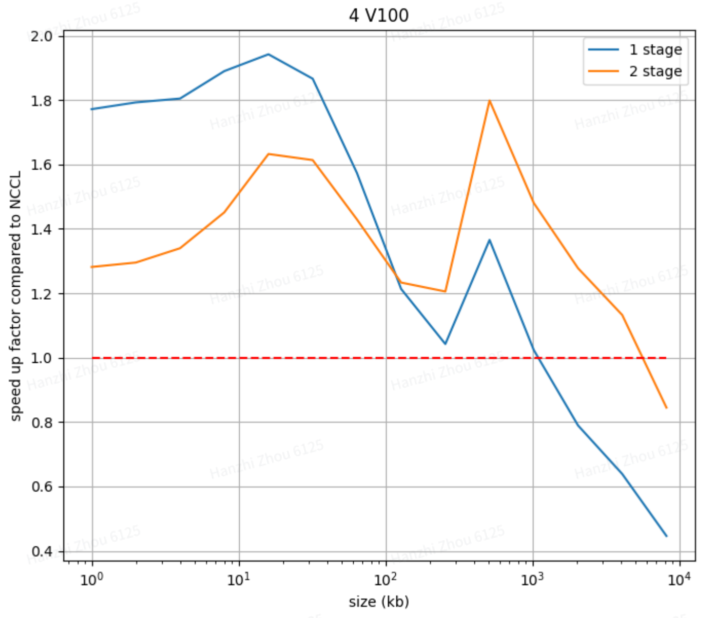
4 A10 GPUs





4 V100 GPUs





Numerical Accuracy vs NCCL

Since we can do reductions in registers, we can upcast half and bfloat16 types to float and then do the accumulation. This leads to better numerical accuracy than NCCL, which can't keep intermediate results in higher precision.

Benchmark: allreduce float16 on different number of gpus. Ground truth is calculated with double precision.

	NCCL mean abs diff	Our mean abs diff
8 GPUs	0.0396678	0.0234039
4 GPUs	0.0158518	0.0115507

End-to-end speedup

Allreduce usually takes 8 - 10% of overall execution time, and this optimization usually reduces allreduce's time by 40 - 60%, so end-to-end gain is about 3% to 5%

Additionally, our kernel is more cuda graph friendly (explained in the background section), which leads to about additional 2% end-to-end speedup

Elementwise fusion

In Llama, there is a residual connection (elementwise add) immediately after each allreduce. We can conveniently use an out-of-place allreduce that adds to result instead of writing to result to perform a fusion with an elementwise add.

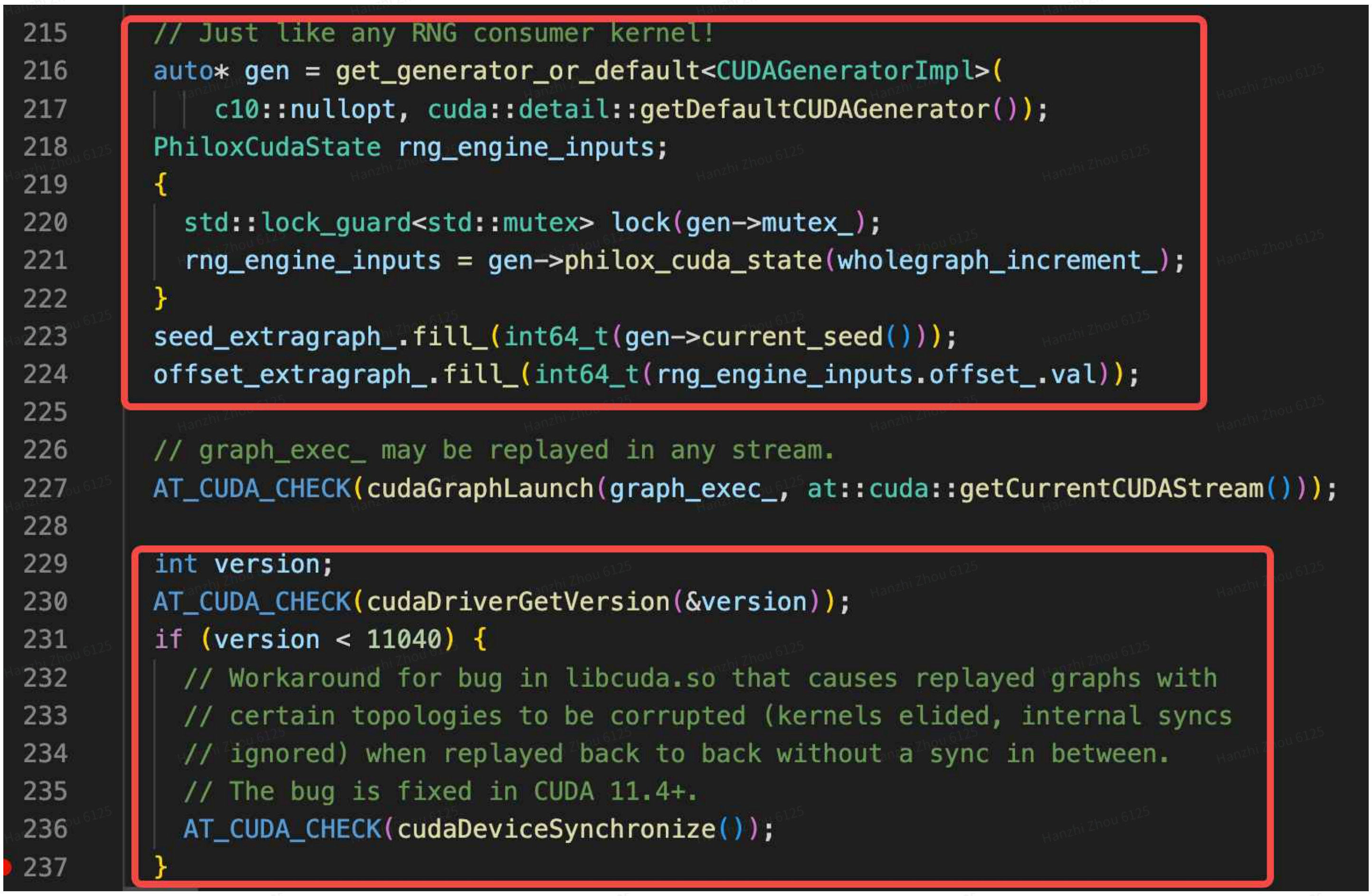
End-to-end speedup

Roughly 2.5% end-to-end speed up

CUDA graph replay optimization

Note: this is not entirely relevant to the allreduce optimization

Pytorch's cuda graph launch has extra considerations on RNG seeding that requires a few cuda API calls and 2 kernel launches to fill just 2 int64s. This is not necessary during serving where kernels don't involve any randomness. Therefore, we removed this part for performance. Additionally, pytorch has a safe guard against older versions of the cuda driver, which is also removed in this optimization.



```cpp
// Just like any RNG consumer kernel!
auto* gen = get_generator_or_default<CUDAGeneratorImpl>(
    c10::nullopt, cuda::detail::getDefaultCUDAGenerator());
PhiloxCudaState rng_engine_inputs;
{
  std::lock_guard<std::mutex> lock(gen->mutex_);
  rng_engine_inputs = gen->philox_cuda_state(wholegraph_increment_);
}
seed_extragraph_.fill_(int64_t(gen->current_seed()));
offset_extragraph_.fill_(int64_t(rng_engine_inputs.offset_.val));

// graph_exec_ may be replayed in any stream.
AT_CUDA_CHECK(cudaGraphLaunch(graph_exec_, at::cuda::getCurrentCUDAStream()));

int version;
AT_CUDA_CHECK(cudaDriverGetVersion(&version));
if (version < 11040) {
  // Workaround for bug in libcuda.so that causes replayed graphs with
  // certain topologies to be corrupted (kernels elided, internal syncs
  // ignored) when replayed back to back without a sync in between.
  // The bug is fixed in CUDA 11.4+.
  AT_CUDA_CHECK(cudaDeviceSynchronize());
}
```

End-to-end speedup

1% less kernel execution time. 1% to 5% speed up depending on how cpu-bound the model is.

Overall end-to-end speedup

On A100 SXM4 80GB

65b 256 256 tp8 bs8: 7.59 -> 6.55
65b 256 256 tp4 bs8: 9.64 -> 8.82
33b 128 128 tp4 bs1: 2.67 -> 2.30

33b 128 128 tp2 bs1: 3.80 -> 3.57

13b 128 128 tp2 bs1: 1.95 -> 1.77

On A10 24GB

13b 128 128 tp2 bs1: 4.85 -> 4.44

>10% gain most of the time across-the-board