An Informal Introduction to CoreSchedule

Jie Ren

14 June 2024

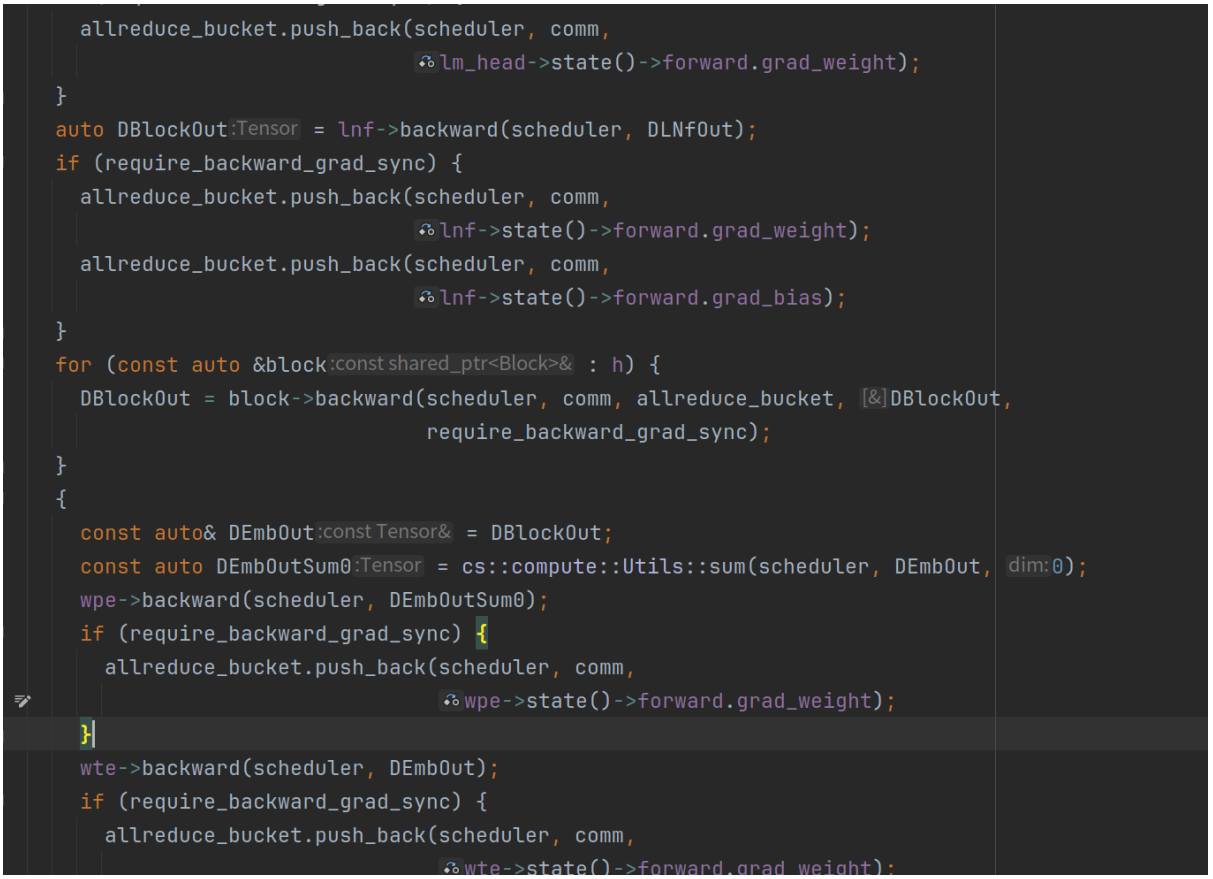
1. Asynchronous Scheduling

As deep learning models get bigger, distributed training is becoming a standard approach. Communication often lags behind computation speed, leading many libraries to adopt asynchronous communication. This approach allows overlapping of communication and computation phases. However, designing an effective asynchronous pipeline can be challenging. For instance, in Megatron (v0.7.0), a well-known distributed training framework, the following occurs:

```
420             if wgrad_compute:
421                 if ctx.sequence_parallel:
422                     world_size = get_tensor_model_parallel_world_size()
423                     dim_size = list(input.size())
424                     dim_size[0] = dim_size[0] * world_size
425
426                     all_gather_buffer = get_global_memory_buffer().get_tensor(
427                         dim_size, input.dtype, "mpu"
428                     )
429                     handle = torch.distributed._all_gather_base(
430                         all_gather_buffer, input, group=get_tensor_model_parallel_group(), async_op=True
431                     )
432
433                     # Here we rely on CUDA_DEVICE_MAX_CONNECTIONS=1 to ensure that the
434                     # gather is scheduled before the input gradient computation
435                     total_input = all_gather_buffer
436                 else:
437                     total_input = input
438             grad_input = grad_output.matmul(weight)
439
440             if ctx.sequence_parallel and wgrad_compute:
441                 handle.wait()
442
443             if wgrad_compute:
444                 grad_output, total_input = prepare_input_tensors_for_wgrad_compute(
445                     grad_output, total_input
446                 )
447
```

- Line 429: an all-gather operation returns a handle.
- Line 438: we perform a matrix multiplication (matmul) to mask the communication delay.
- Line 441: we wait for the communication to complete.

Our library can automatically handle the data dependency. The user only need to submit tasks.

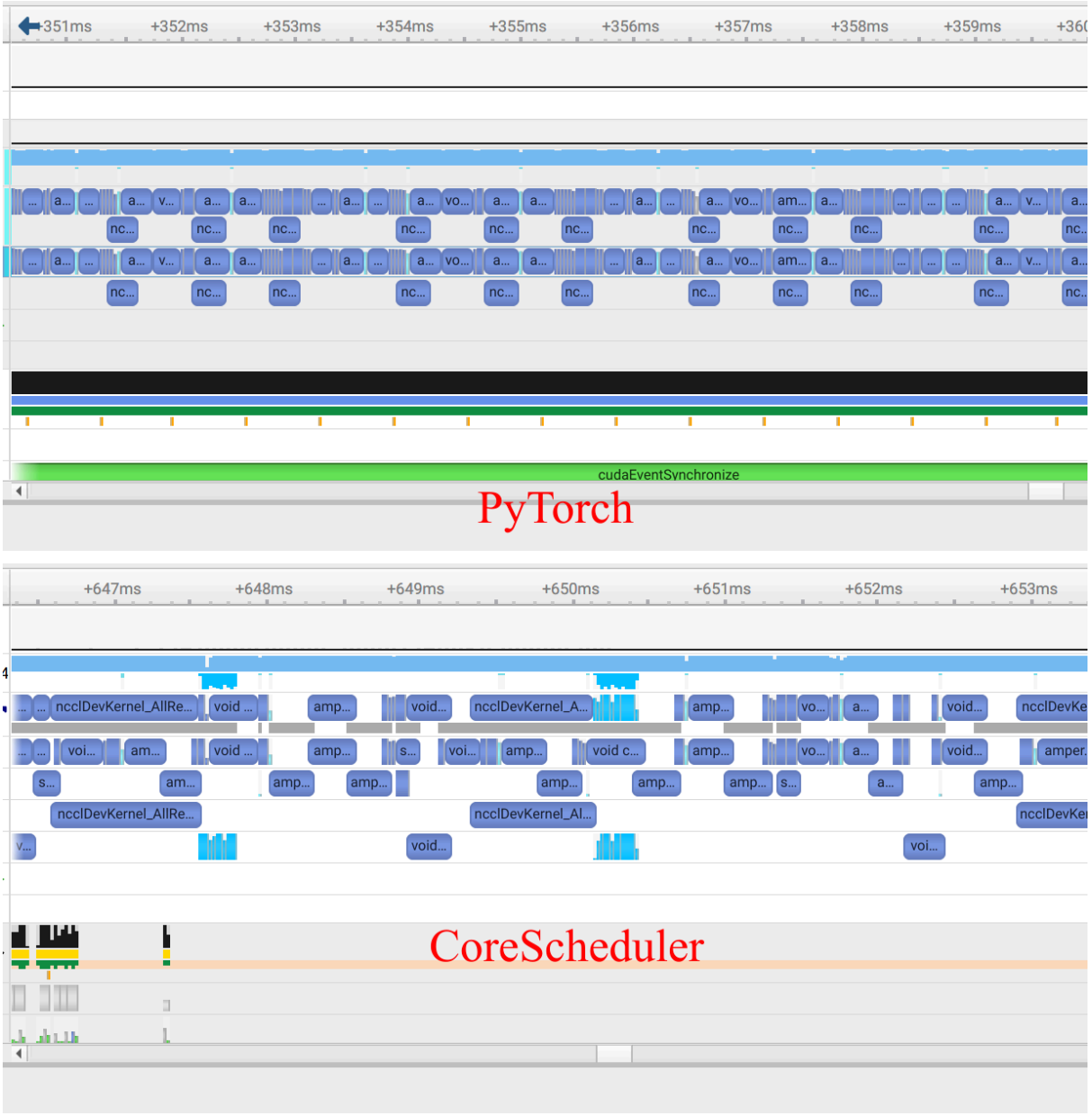


After scheduling the communication and computation tasks, we discovered the value in also scheduling different computation tasks. Below are the profiling results of PyTorch (v2.3) during the backpropagation of a GPT2 model on an A100 server.

The data shows that all computation tasks are executed sequentially. However, our scheduler is able to overlap independent tasks—yes, even GPU computations can be overlapped.

Using this method, we achieved a 1.15x speedup without needing any additional kernel fusion; we simply optimized the computation scheduling using the existing kernels.

We also implemented a dual GPU DDP GPT2 training example using both PyTorch and our CoreScheduler. PyTorch's profiling results indicate basic computation-communication scheduling.



Our results, however, reveal a more complex scheduling behavior with our scheduler. This complexity allowed us to achieve a 1.3x speedup over PyTorch on a dual A100 NVLink-enabled server.

2. Why did we choose C++?

For now, we use C++ because it offers efficient multi-threading and fine-grained lifecycle management, plus it easily interacts with hardware-close libraries like HWLOC. We may consider Rust or Python later, but our team's current expertise lies in C++.

3. What's next?

a) AMP training
b) Develop more sophisticated distributed algorithms (e.g., DeepSpeed Zero).
c) Introduce advanced kernel fusion techniques with CUTLASS/cuDNN/NV LTO JIT.
d) Fault tolerance (modify the NCCL library)
e) Create more advanced models (e.g., Llama-3).

f) Investigate static scheduling based on computation graphs.