Ultra Scale Playbook

Nanotron – Internal lib for distributed training

4000 scaling accelerated expt, but actually ran 16K experiments for ablations bug fixing, etc.

Outer area - OOM / Failures

Middle – Most successful runs i.e highest throughput
Measured using tokens/s or MFU

All expts useful over 2 GPU's as well.

Mostly directed towards consumer

3 key challenges:-

→ Memory usage: If training step doesn't fit on GPU, training cannot proceed

→ Compute efficiency: H/w should spend most time computing, so need to reduce time on data transfers / waiting for other GPU's to perform

→ Communication overhead: b/t comm overhead as it keeps GPU's idle. Make best use of intra-node (fast) & inter-node bandwidths & overlap comms with compute

DP becomes bad after 512 GPU's, & need to combine it with other methods like TP, etc

Mem usage:

→ OOM errors where?

→ Track layer, activation, optimizer, etc. memory usage

→ Tool is 95% accurate when predicting OOM

→ If and forward/backward peak above 75% ram of GPU, it will OOM.

Batch size is important

Bst = Batch size tokens

bst = bs * seq_len

Sweet spot is 4-60 mil tokens per batch.

Memory usage of LLMs depends on several items:

→ Model weights

→ Model gradients

→ Optimizer states

→ Activations needed to compute gradients

Number of params of an LLM is given by:

$$N = h * V + L * (12 * h^2 + 13 * h) + 2 * h$$

h - hidden dim

V - vocab size

L - no. of layers

Activation Recomp

→ Can achieve 70% mem reduction. Used in Flash Attention

→ Discard some activations during forward pass to save memory & spend extra compute to recompute it on the fly during backward pass

Gradient Accumulation

→ split batch into micro-batch

→ Compute gradients for each micro-batch, sum them up before optimizer step.

→ Allows big batch size but keeps memory footprint constant.

→ compatible with activation recomputation

→ However, gradient accumulation requires multiple FW/BW passes per opt step thereby increasing overhead & slowing down training.

Data Parallelism

→ Replicate model on multiple GPU's
→ Each GPU sees different micro batch
→ Run FW/BW passes on each micro batch
→ Sum gradients thru multiple GPUs

Parallel Prog Crash Course

Usual ops: Gather, All Gather, Reduce, All Reduce, Reduce Scatter
Use NCCL via PyTorch

→ However, naive DP has no overlap b/w FW/BW pass & AllReduce computation

First optimisation: Overlap gradient syn with backward pass

→ Gradient for curr layer can already be summed before gradient propagates to earlier layers.

Eg: As soon as last layer grad is calculated, sum & continue backprop for earlier layers

→ Achieved using all-reduce hook function for each param in Pytorch

→ This will overlap computation & comms.
This is good!!!

Second optimization: Bucketing gradients

→ GPU ops more efficient on large tensors compared to small tensors

→ Also true for comms

→ Group gradients in buckets & launch single AllReduce for each bucket.

→ Buckets are generally capped by size. Eg: 25 MB bucket

Third optimization: Interplay with gradient accumulation

Disable gradient sync on passes that don't need reduction

model.no_sync() does this

Now, global batch size (or batch size)

will be:

$$gbs = bs = mbs \times grad\_acc \times dp$$

grad_acc — gradient accumulation steps

dp — No. of parallel instances when using data parallel

→ Maximize dp ove grad-acc steps
since DP is parallel, grad-acc will be sequential

→ Add gradient accumulation over DP to achieve batch size when DP alone is not sufficient

This is parallelism over 1 dimension i.e 1D parallelism.

- Maximize dp one grad-acc steps

since DP is parallel, grad-all will be sequential

- Add gradient accumulation over DP to achieve batch size

when DP alone is not sufficient

This is parallelism over 1 dimension

i. e ID parallelism.