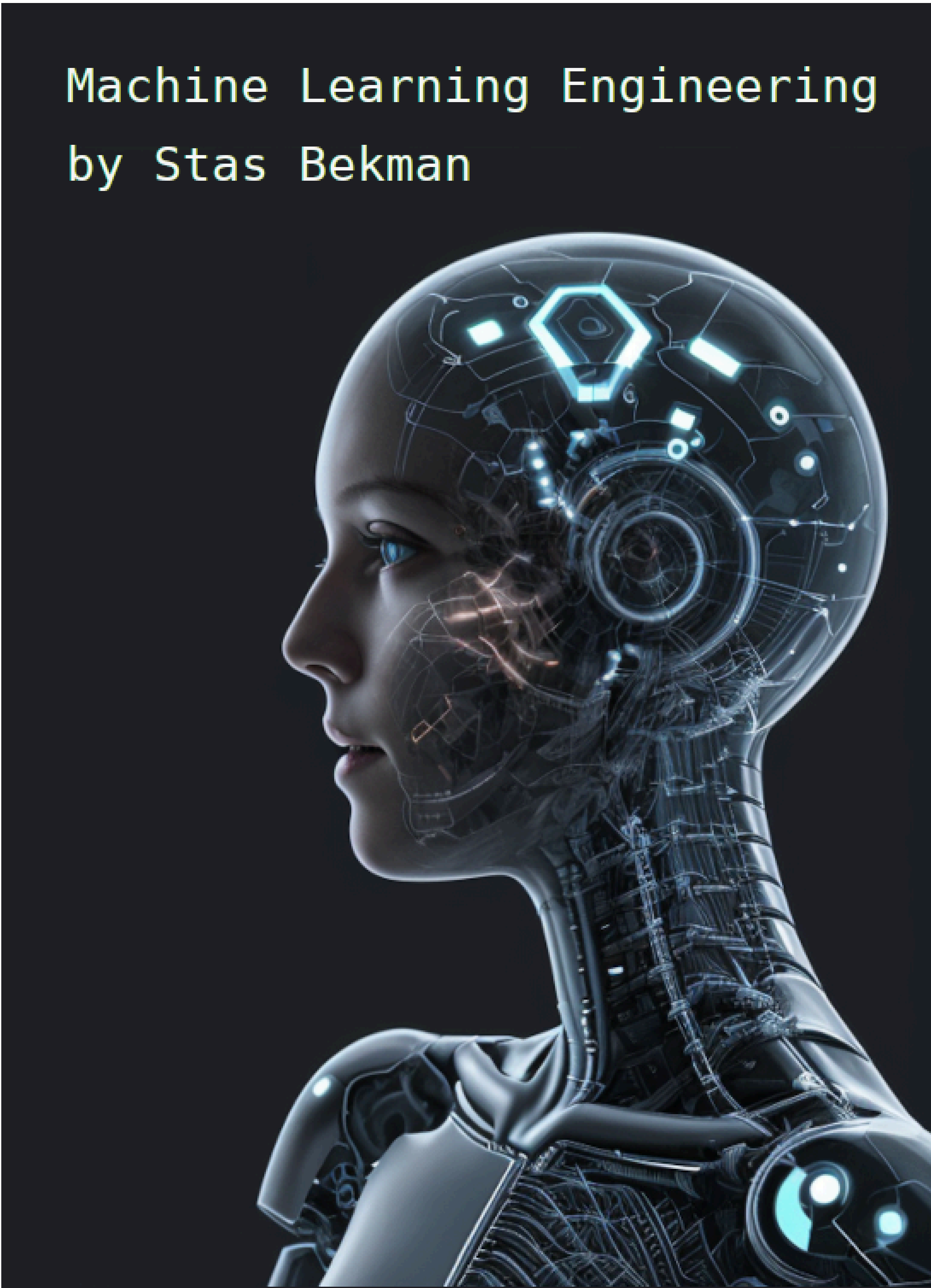


Machine Learning Engineering Open Book

This is a PDF version of Machine Learning Engineering Open Book by Stas Bekman.

As this book is an early work in progress that gets updated frequently, if you downloaded it as a pdf file, chances are that it's already outdated - make sure to check the latest version at https://github.com/stas00/ml-engineering.

This PDF was generated on 2024-01-24.

Machine Learning Engineering Open Book

An open collection of methodologies to help with successful training of large language models and multi-modal models.

This is a technical material suitable for LLM/VLM training engineers and operators. That is the content here contains lots of scripts and copy-n-paste commands to enable you to quickly address your needs.

This repo is an ongoing brain dump of my experiences training Large Language Models (LLM) (and VLMs); a lot of the know-how I acquired while training the open-source BLOOM-176B model in 2022 and IDEFICS-80B multi-modal model in 2023. Currently, I'm working on developing/training open-source Retrieval Augmented models at Contextual.AI.

I've been compiling this information mostly for myself so that I could quickly find solutions I have already researched in the past and which have worked, but as usual I'm happy to share these with the wider ML community.

Table of Contents

My apologies if the layout is a bit unstable while I'm writing new chapters and gradually re-organizing the content to be more intuitive.

Part 1. Insights

1. The AI Battlefield Engineering - What You Need To Know

Part 2. Key Hardware Components

1. Accelerator - the work horses of ML - GPUs, TPUs, IPUs, FPGAs, HPUs, QPUs, RDUs (WIP)
2. Network - intra-node and inter-node connectivity, calculating bandwidth requirements
3. Storage - local and distributed disks and file systems
4. CPU - cpus, affinities (WIP)
5. CPU Memory - how much CPU memory is enough - the shortest chapter ever.

Part 3. Performance

1. Fault Tolerance
2. Performance
3. Multi-Node networking
4. Model parallelism

Part 4. Operating

1. SLURM
2. Training hyper-parameters and model initializations
3. Instabilities

Part 5. Development

1. Debugging software and hardware failures
2. And more debugging
3. Reproducibility

4. Tensor precision / Data types
5. HF Transformers notes - making small models, tokenizers, datasets, and other tips

Part 6. Miscellaneous

1. Resources - LLM/VLM chronicles

Shortcuts

Things that you are likely to need to find quickly and often.

Tools:

- all_reduce_bench.py - a much easier way to benchmark network throughput than nccl-tests.
- torch-distributed-gpu-test.py - a tool to quickly test your inter-node connectivity

Guides:

- debugging pytorch applications - quick copy-n-paste solutions to resolve hanging or breaking pytorch applications
- slurm for users - a slurm cheatsheet and tricks
- make tiny models/datasets/tokenizers
- LLM/VLM chronicles collection

Book Building

If you want to build a PDF, check links, etc. See Book building

Gratitude

None of this would have been possible without me being entrusted with doing the specific LLM/VLM trainings I have learned this know-how from. This is a privilege that only a few enjoy due to the prohibitively expensive cost of renting huge ML compute clusters. So hopefully the rest of the ML community will vicariously learn from these notes.

Special thanks go to Thom Wolf who proposed that I lead the BLOOM-176B training back when I didn't know anything about large scale training. This was the project that catapulted me into the intense learning process. And, of course, HuggingFace for giving me the opportunity to work full time on BLOOM-176B and later on IDEFICS-80B trainings.

Contributing

If you found a bug, typo or would like to propose an improvement please don't hesitate to open an Issue or contribute a PR.

License

The content of this site is distributed under Attribution-ShareAlike 4.0 International.

My repositories map

✔ **Machine Learning:** ML Engineering Open Book | ML ways | Porting

✔ **Guides:** The Art of Debugging

✔ **Applications:** ipyexperiments

✔ Tools and Cheatsheets: bash | conda | git | jupyter-notebook | make | python | tensorboard | unix

The AI Battlefield Engineering - What You Need To Know

This chapter is one person's opinionated overview of the ML/AI Engineering reality, which may or may not be another person's reality. The intention is to help you start asking the right questions and get your ML Engineering needs met.

Basics

What's important in the AI race?

Training:

1. How fast one can train a better model (first to market advantage)
2. How much $$ was spent (do we still have money left to pay salaries to talent after training?)

Inference:

1. Fast latency (users are used to msec response times and will leave if the response takes seconds)
2. Fast throughput (how many concurrent queries can be processed)
3. How much $$ is being spent per user (can we rent more GPUs to acquire more users and/or improve (1) and (2)?)

What are the needs of LLM training?

1. Fast compute massively dominated by matrix multiplications
2. Fast enough memory, IO, network and CPU to feed the compute

Corollary: If when you buy or rent hardware you invest in the fastest accelerators, but cheap out on any of the other components you wasted $$ and you might not win the race as it'll take longer to train.

What are the workhorses of ML?

- An accelerator or a processing unit is what does most of the work.

- Since ML does a lot of parallel processing (SIMD) GPUs were used at the beginning, but now you additionally have TPUs, IPUs, FPGAs, HPUs, QPUs, RDUs, etc. Recent CPUs are becoming used as accelerators as well, especially for inference.

More details.

AI driving entities

- AI companies - train models/build products around self-trained or trained-by-others' models, in-house research.
- Academia - does massive research and write papers. Lots of new ideas are generated.
- AI enthusiasts - lots of good will available, some pull resources/talents together to train open access models, with donated compute by HPCs and an occasional cloud, or a university cluster.
- Entrepreneurs - lots of low hanging fruit to pick - creative reselling of services, making ML-driven apps, and using various ingenious combinations of available resources to create amazing outcomes.

Information sharing

- It's very surprising that almost everybody involved in the domain of AI shares a lot of the discoveries with the community.
- Surely, companies don't disclose all of their IP, but a lot of it does get shared in the form of knowledge or model

weights
- Companies that publish a lot of IP and models tend to attract higher quality talent.
- Twitter seems to be the central platform where one must be to follow what's going on

The AI bubble

- The Dot-com bubble occurred during 1995-2000. And a very similar situation is happening right now in the AI space.
- There is a lot of money available to create new startups or boost the existing companies. It's relatively easy to raise millions of dollars.
- As we are in the wild-wild-west stage of the AI industry it's very difficult to predict the future, and so pretty much anything goes as far as startup ideas go, as long as it sounds reasonable.
- What distinguishes the AI bubble from the Dot-com bubble, is that one didn't actually need much money to operate a Dot-com company - most of the raised money went to marketing and some to staff, barely any to compute. AI companies need millions of dollars because training LLMs requires an insane amount of compute, and that compute is very expensive. e.g. 1x NVIDIA H100 costs ~$30k and a company may need 512 of those, which is $15M (not counting the other hardware components and related costs)!

ML Engineer's heaven and hell

This is my personal LLM/VLM trainings-based heaven and hell. YMMV.

ML Engineer's heaven

1. A well built HPC, or a full service cloud based cluster, where someone diligently and timely takes care of the hardware and the systems.

   I just need to bring my training software and do the training, which is already an insanely complicated job requiring special skills.
2. Lots of nodes available for exclusive unlimited use
3. Fast inter-node connectivity that doesn't bottleneck the accelerators and which isn't shared with other users
4. Huge local super-fast NVME based shared filesystem that can fit datasets and checkpoints
5. Barebones Linux w/ SLURM and minimal software to be able to launch training jobs
6. `sudo`er access to ease the work with a team of people

ML Engineer's hell

1. A cloud or in-house cluster, where you have to do everything - sysadmining, replacing hardware, dealing with outages, etc. And to do the training on top of that.
2. A smallish slow shared filesystem (NFS?), with cloud to draw data from and checkpoint to
3. Slow inter-node leading to low accelerator utilization
4. Inter-node shared with other users which make the network erratic and unpredictable
5. Super-complicated cloud console with gazillion of screens and steps to set even simple things up
6. Not being able to swap out failing hardware fast
7. Needing to timeshare the nodes - with wait times between training jobs
8. Having other concurrent users who might use up the whole disk, leading to trainings crashing

9. Not being able to kill jobs others on the team started and went to sleep

Getting compute

There are 3 main choices to where one gets compute:

- Rent on the cloud
- Get a timeshare on an HPC
- Buy it

Renting on the cloud

This is currently the prevalent way of getting compute.

Pros:

- Easy to expand or contract the size of the cluster
- Easy to upgrade from the old hardware generation to the new one in a few years
- Cluster management could be easily outsourced

Cons:

- Expensive, unless you negotiate a long term (1-3 year) contract for hundreds of accelerators
- You will be tempted to buy many tools and services that you may or may not need
- You always get charged whether you use your cluster fully or not

Using HPC

There aren't that many HPCs out there and so the amount of available resources is limited.

Pros:

- Managed for you - all you need is your software to do the training and a bit of SLURM know-how to launch jobs
- Often sponsored by the local government/university - probably could get the job done for less $$ or even free (e.g. we trained BLOOM-176B for free on JeanZay HPC!)

Cons:

- needing to time share compute with other teams == short job times with possible long wait times in between - could be difficult to finish training quickly
- The inter-node network is likely to be unstable as it'll be used by other teams
- Have to abide by the HPC's rules (e.g. no `sudo` access and various other rules to follow)
- In a way the HPC cluster will be what it'll be - you can't make the network faster and often even getting some software installed can be tricky.

Buying hardware

It's mainly universities that buy and build their own clusters, and some big companies do that too.

Pros:

- If you can deploy the hardware 24/7 for more than a few years the total cost will be cheaper than renting
- Easy to provide fast local storage - a good NVME raid would be much cheaper and faster than online storage

Cons:

- You're stuck with the outdated hardware just a few years after it was purchased - might be able to resell
- Must buy more than needed - Hardware tends to break, especially when it's used 24/7, RMA could take weeks
- Have to hire talent to manage the in-house solution
- Have to figure out cooling, electric costs, insurance, etc.

The needs of technology

Can you feed the furnace fast enough?

Imagine a steam locomotive - the engine is great, but if the [fireman](https://en.wikipedia.org/wiki/Fireman_(steam_engine)) isn't fast enough to shovel the coal in, the train won't move fast.

source

This is the current state of ML hardware: The bottleneck is in moving bits and not the compute.

- Accelerators get ~2x faster every 2 years (Moore's law)
- Network and memory are not! Already now both are compute bottlenecks
- IO can be another bottleneck if your DataLoader has to pull data from the cloud
- CPU is fine as long as it has enough cpu-cores for DataLoader workers, and main processes

Corollary: research the whole machine and not just its engine.

a crazy idea: the older GPUs might do fine if you can actually feed them as fast as they can compute. And if you can get 3x of them at the same cost as the next generation GPU you might finish training sooner and a lower cost.

TFLOPS

- Once you choose the architecture and the size of the model and how many tokens you want to train the model for you immediately know how much compute will be required to accomplish this goal. Specifically you can now calculate how many floating point operations will be needed.

- All that is missing is comparing different compute providers to how many floating point operations their hardware

can computes per secs (TFLOPS) and their cost per unit and now you can tell the total approximate cost of the training.

1. Calculate the time needed to train given the TFLOPS of the considered solution:

   ```
   total_tflops_required / tflops_of_this_compute_unit = time_in_seconds
   ```

   Let's say it came to be 604800 secs or 7 days.

2. Look at the cost of using this compute solution for 7 days and now you know the total $$ to train this model.

3. Look at other proposals and calculate the same - chose the best option.

- As mentioned earlier, time is of a huge importance, so you might still choose a more expensive solution if finishing the training sooner is important because you want to be first to market.

Unfortunately, this math is only partially correct because the advertised peak TFLOPS are typically unachievable. The MFU section delves into it.

Model Flops Utilization (MFU)

As mentioned in the previous section, some (most?) vendors publish unrealistic peak performance TFLOPS - they aren't possible to achieve.

Model Flops Utilization (MFU) is the metric that tells us how well the accelerator is utilized. Here is how it is calculated:

1. Measure the actual TFLOPS by calculating how many floating point operations a single training iteration takes and dividing that number by the number of seconds this iteration took.
2. Divide the actual TFLOPS by advertised TFLOPS to get the MFU

Example: Let's say you're training in BFLOAT16 precision:

- If a single iteration requires 624 Tera floating point operations and it took 4 secs to run then we know that we get: `624/4=156` actual TFLOPS
- now BF16@A100 is [advertised as 312TFLOPS]() so `156/312=0.5` gives us 50% MFU.

Practically:

- with NVIDIA GPUs if you're above 50% MFU on a multi-node setup with a large model you're already doing fantastic
- recent advancements in more efficient scalability solutions keep on increasing MFU
- slow networks and inefficient frameworks or untuned configuration lower MFU

Therefore once you know the MFU you can now adjust the cost estimate from the previous section. In the example there we said it'll take 7 days to train, but if MFU is 50%, it means it'll take 14 days to train.

Moving bits

Why can't the advertised TFLOPS achieved? It's because it takes time to move data between accelerator memory and compute and additionally it takes even more time to move data from disk and other gpus to the accelerator's memory.

- There is not much can be done about the accelerator memory since its bandwidth is what it is - one can only write more efficient software to make data move faster to/from the accelerator - hint: fused and custom written kernels (like [torch.compile]() and [flash attention]())

- If you only have a single GPU and the model fits its memory, you don't need to worry about the network - accelerator memory is the only bottleneck. But if you have [to shard the model across multiple GPUs]() network becomes the bottleneck.

- Intra-node Network - is very fast, but difficult to take advantage of for large models - [Tensor parallelism]() and [sequence parallelism]() address part of this problem. ([more]()).

- Inter-node Network - typically is too slow on most server setups - thus this is the key component to research! Efficient frameworks succeed to partially hide the comms overhead by overlapping compute and comms. But if comms take longer than compute, the comms are still the bottleneck. more.
- Storage IO is important primarily for feeding the DataLoader workers and saving the checkpoints. more.

  1. Typically with enough DL workers the DataLoader adds very little overhead.
  2. While checkpoints are being saved the accelerators idle unless some async saving solution is used, so fast IO is crucial here

Key hardware components

Accelerators

As of this writing here are the most common accelerators that can be used for training, finetuning and inference ML models:

Widely available:

- NVIDIA A100 - huge availability across all clouds, but is already gradually being replaced by H100

Available, but locks you in:

- Google TPUs - fast! but the cost is a lock-in into a single vendor and cloud

Emerging to general availability:

- NVIDIA H100 - 2-3x faster than A100 (half precision), 6x faster for fp8
- AMD MI250 ~= A100 - very few clouds have them
- AMD MI300 ~= H100 - don't expect until 1.5-2 years from now to be GA
- Intel Gaudi2 ~= H100 - starting to slowly emerge on Intel's cloud
- GraphCore IPU - very difficult to find, paperspace has them
- Cerebras WaferScale Engine - available on Cerebras' cloud

Accelerator Interoperability

In general most (all?) accelerators are supported by major frameworks like PyTorch or TensorFlow and the same code should run everywhere with small modifications as long as it doesn't use any accelerator-specific functionality.

For example, if your PyTorch application includes custom CUDA kernels it'll only work on NVIDIA GPUs and may be on AMD MI-series.

- NVIDIA GPUs: all based on CUDA, which most training frameworks support. You can easily moved between different NVIDIA GPUs and most things would work the same.
- AMD MI250/MI300: with PyTorch using ROCm you can run most CUDA-based software as is. This is really the only inter-operable accelerator with the NVIDIA stack.
- Gaudi2: if you use HF Transformers/Diffusers you can use optimum-habana. If you use HF Trainer with NVIDIA GPUs it should be relatively easy to switch to train/infer on Gaudi2.
- GraphCore IPU: can also be run via PyTorch via poptorch
- Cerebras: is also working on PyTorch support via Cerebras Software Platform (CSoft) via XLA.

Also in general most ML code could be compiled into cross-platform formats like Open Neural Network Exchange (ONNX) which can be run on a variety of accelerators. This approach is typically used more often for inference workloads.

Network

- If you want to train a large model that doesn't fit onto a single accelerator's memory you have to rely on the intra- and inter-node networks to synchronize multiple accelerators.
- The biggest issue right now is that compute hardware advancements move faster than networking hardware, e.g. for NVIDIA NVLink intra-node:

GPU	Computefp16 TFLOPS	Compute speedup	Intra-node GBps	Intra-node speedup
V100	125	1	300	1
A100	312	2.5	600	2
H100	989	8	900	3

- You can see that A100 was 2.5 faster than V100, and H100 is ~3x faster than A100. But the intra-node speed of NVLink has only increased by 300GBps each generation.
- Moreover, all 3 generations of NVLink use identical NICs of the same 50GBps duplex throughput. They have just doubled and tripled the number of links to speed things up. So there was 0 progress in that technology.
- The inter-node situation isn't any better with most NICs there doing 100 or 200Gbps, and some 400Gbps are starting to emerge. (correspondingly in GBps: 12.5, 25 and 50). It's the same story here, some solutions provide dozens of NICs to get to higher speeds.
- Also typically with LLMs the payload is so large that network latency is often negligible for training. It's still quite important for inference.

Intra-node Network

- Pay attention to bytes vs bits. 1Byte = 8bits. 1GBps = 8Gbps.
- If you need to reduce bits (e.g. gradients) across multiple nodes, it's the slowest link (Inter-node) that defines the overall throughput, so intra-node speed doesn't matter then
- Tensor parallelism and sequence parallelism have to remain within the node to be efficient - only makes sense with fast intra-node speed

NVIDIA:

- NVIDIA-based compute nodes come with 50GBps duplex NVLInk
- Some have a lot of NVLinks, others less but typically plenty w/ at least 900GBps (5.6Tbps) duplex for H100, 600GBps for A100 nodes

Intel Gaudi2:

- 8x 21 NICs of 100GbE RoCE v2 ROMA for a total of 2.1TBps

More details

Inter-node Network

- An order of magnitude slower than Intra-node
- You will see a wide range of speeds from 50Gbps to 3200 Gbps

- You need to reduce gradients and other bits faster than compute to avoid idling accelerators
- You typically get at most 80% of advertised speed. e.g., if you are told you get 800Gbps, expect ~480Gbps.
- If moving to fp8 H100 is 18x faster than V100
- We are yet to see if 3200Gbps for H100s will be enough to keep high MFU.
- Practically less than 3x but it's a good estimate

More details.

Storage

There are 3 distinct Storage IO needs in the ML workload:

1. You need to be able to feed the DataLoader fast - (super fast read, don't care about fast write) - requires sustainable load for hours and days
2. You need to be able to write checkpoints fast - (super fast write, fastish read as you will be resuming a few times) - requires burst writing - you want super fast to not block the training for long (unless you use some sort of cpu offloading to quickly unblock the training)
3. You need to be able to load and maintain your codebase - (medium speed for both reading and writing) - this also needs to be shared since you want all nodes to see the same codebase - as it happens only during the start or resume it'll happen infrequently

- Most of the time you're being sold 80% of what you paid. If you want a reliable 100TBs you need to rent 125TBs or your application may fail to write long before the disk is full.
- Shared Distributed Filesystem:
  1. non-parallel shared file systems can be extremely slow if you have a lot of small files (=Python!)
  2. You want Parallel FS like GPFS (IBM Spectrum Scale) or Lustre (Open Source)

More details.

CPU Memory

You need enough memory for:

- 2-3 possibly DL workers per Accelerator (so 16-24 processes with 8 accelerators per node)
- Even more memory for DL workers if you pull data from the cloud
- Enough memory to load the model if you can't load to accelerator directly
- Often used for accelerator memory offloading - extends accelerator's memory by swapping out the currently unused layers - if that's the target use, then the more cpu memory is available - the better!

CPU

This is probably the least worrisome component.

- Most clouds provide beefy CPUs with plenty of cpu cores
- You need to have enough cores to run 2-3 DL workers +1 per gpu - so at least 30 cores
- Even more cores for DL workers if you have complex and/or slow DL transforms (CV)
- Most of the compute happens on GPUs

Impress others with your ML instant math

Tell how many GPUs do you need in 5 secs

- Training in half mixed-precision: `model_size_in_B * 18 * 1.25 / gpu_size_in_GB`
- Inference in half precision: `model_size_in_B * 2 * 1.25 / gpu_size_in_GB`

That's the minimum, more to have a bigger batch size and longer sequence length.

Here is the breakdown:

- Training: 8 bytes for AdamW states, 4 bytes for grads, 4+2 bytes for weights
- Inference: 2 bytes for weights (1 byte if you use quantization)
- 1.25 is 25% for activations (very very approximate)

For example: Let's take an 80B param model and 80GB GPUs and calculate how many of them we will need for:

- Training: at least 23 GPUs `80*18*1.25/80`
- Inference: at least 3 GPUs `80*2*1.25/80`

More details.

Traps to be aware of

As you navigate this very complex AI industry here are some thing to be aware of:

Say no to "will make a reasonable effort to ..." contracts

- If you contract doesn't have clear deliverables (time and performance) don't be surprised if you paid for something you won't receive in time you need it or not at all
- Be very careful before you sign a contract that includes clauses that start with "we will make a reasonable effort to ...".

  When was the last time you went to the bread section of the supermarket and found a lump of half-baked dough with a note "we made a reasonable effort to bake this bread, but alas, what you see is what you get"?

  But for whatever reason it's acceptable to create a legal contract where the provider provides neither delivery dates nor performance metrics and doesn't provide stipulations for what will they do in recompense when those promises aren't fulfilled.

Beware of hardware and software lock-in scenarios

- Some cloud providers will make you use very proprietary tools or hardware that will make it very difficult for you to leave down the road because you will have to retool everything if you leave
- Consider what would be the cost of moving to a different provider should this provider prove to be not satisfactory or if they don't have a capacity to fulfill your growing needs.
- If you rent a cluster with a generic Linux box with generic open source tools it should be trivial to move from one provider to another as almost everything would work out of the box
- Obviously if you choose compute that requires custom software that works for that hardware only and you can't rent this hardware anywhere else you're setting yourself up for a lock-in

Don't buy what you don't really need

- The cloud providers have mostly the same generic hardware, which leads to a very slim $$ margin and so in order to make big $$ they invent products and then try to convince you that you need to buy them. Sometimes you actually need those products, but very often not. See also the previous section on lock-in, since proprietary products usually mean a partial lock-in.

- Often it's easy to observe the 3 step marketing technique for solutions that seek a problem to solve:

1. Convince a couple of well respected customers to use the provider's proprietary products by giving them huge discounts or even pay them to use them
2. Use those in step 1 as the social approval lever to reel in more converts
3. Then scoop the rest of the strugglers by telling them that 80% of your customers (1+2) use these amazing products

When marketing these products it's important:

- to mention how well they work with a dozen of other products, since now you're not buying into a single product but into a whole proprietary product-sphere.
- to use really nice looking complicated diagrams of how things plug into each other, and move really fast to the next slide before someone asks a difficult question.

HPCs are probably a good group of compute providers to learn from - they have no funds to create new products and so they creatively address all their needs using mostly generic open source tools with some custom written software added when absolutely needed.

Unsolicited advice

To conclude I thought I'd share some insights to how one could slightly improve their daily AI battlefield experience.

FOMO and avoiding depression

If you read Twitter and other similar ML-related feeds you're guaranteed to feel the fear of missing out, since there is probably at least one new great model getting released weekly and multiple papers are getting published daily and your peers will publish their cool achievements hours.

We are dealing with **very** complicated technology and there is a small handful of people who can absorb that much new material and understand / integrate it.

This can be extremely depressing and discouraging.

I deal with it by looking at twitter about once or twice a week. I mostly use Twitter in broadcast mode - that is if I have something to share I post it and only watch for possible follow up questions.

Usually all the important news reach me through other people.

Don't try to know everything

The pace of innovation in the field of AI is insane. It's not possible to know all-things-AI. I'd dare to say it's not possible to know even 10% of it for most of us.

I realized this very early one and I stopped paying attention to most announcements, tutorials, keynotes, etc. Whenever I have a new need I research it and I discover what I need and I have to be careful not to try to learn other things not pertinent to the goal at hand.

So I actually know very little, but what I have researched in depth I know quite well for some time and later I forget even that (that's why I write these notes - so that I can easily find what I have already researched).

So if you ask me something, chances are that I don't know it, but the saving grace for me is that if you give me time I can

figure it out and give the answer or develop a solution.

Don't beat yourself up when using half-baked software

Because the ML field is in a huge race, a lot of the open source software is half-baked, badly documented, badly tested, at times poorly supported. So if you think you can save time by re-using software written by others expect spending hours to weeks trying to figure out how to make it work. And then keeping it working when the updates break it.

The next problem is that most of this software depends on other software which often can be just as bad. It's not uncommon where I start fixing some integration problem, just to discover a problem in a dependent package, which in its turn has another problem from another package. This can be extremely frustrating and discouraging. Once excepts to save time by reuse, but ends up spending a long time figuring out how to make it work. At least if I write my own software I have fun and it's a creative process, trying to make other people's software work is not.

So at the end of the day we are still better off re-using other people's software, except it comes at an emotional price and exhaustion.

So first of all, try to find a way not to beat yourself up if the software you didn't write doesn't work. If you think about it, those problems aren't of your creation.

Learning how to debug efficiently should also make this process much less painful.

Contributors

Mark Saroufim,

Accelerators

XXX: This chapter is a super-early WIP

Compute accelerators are the workhorses of the ML training. At the beginning there were just GPUs. But now there are also TPUs, IPUs, FPGAs, HPUs, QPUs, RDUs and more are being invented.

There exist two main ML workloads - training and inference. There is also the finetuning workload which is usually the same as training, unless a much lighter LORA-style finetuning is performed. The latter requires significantly fewer resources and time than normal finetuning.

In language models during inference the generation is performed in a sequence - one token at a time. So it has to repeat the same `forward` call thousands of times one smallish `matmul` (matrix multiplication or GEMM) at a time. And this can be done on either an accelerator, like GPU, or some of the most recent CPUs, that can handle inference quite efficiently.

During training the whole sequence length is processed in one huge `matmul` operation. So if the sequence length is 4k long, the training of the same model will require a compute unit that can handle 4k times more operations than inference and do it fast. Accelerators excel at this task. In fact the larger the matrices they have to multiply, the more efficient the compute.

The other computational difference is that while both training and inference have to perform the same total amount of `matmuls` in the `forward` pass, in the `backward` pass, which is only done for training, an additional 2x times of `matmuls` is done to calculate the gradients with regards to inputs and weights. And an additional `forward` is performed if activations recomputation is used. Therefore the training process requires at 3-4x more `matmuls` than inference.

Subsections

- Troubleshooting NVIDIA GPUs

Bird's eye view on the high end accelerator reality

While this might be changing in the future, unlike the consumer GPU market, as of this writing there aren't that many high end accelerators, and if you rent on the cloud, most providers will have more or less the same few GPUs to offer.

GPUs:

- As of today, ML clouds/HPCs started transitioning from NVIDIA A100s to H100s and this is going to take some months due to the usual shortage of NVIDIA GPUs.
- AMD's MI250 started popping up here and there, but it's unclear when it'll be easy to access those. From a recent discussion with an AMD representative MI300 is not planned to be in general availability until some time in 2025, though some HPCs already plan to get them some time in 2024.

HPU:

- Intel's Gaudi2 are starting to slowly emerge on Intel's cloud

IPU:

- And there is Graphcore with their IPU offering. You can try these out in Paperspace through their cloud notebooks.

TPU:

- Google's TPUs are, of course, available but they aren't the most desirable accelerators because you can only rent them, and the software isn't quite easily convertible between GPUs and TPUs, and so many (most?) developers remain in the GPU land, since they don't want to be locked into a hardware which is a Google monopoly.

Pods and racks:

- Cerebras' WaferScale Engine (WSE)
- SambaNova's DataScale
- dozens of different pod and rack configs that compose the afformentioned GPUs with super-fast interconnects.

That's about it as Q4-2023.

Glossary

- CPU: Central Processing Unit
- FPGA: Field Programmable Gate Arrays
- GPU: Graphics Processing Unit
- HBM: High Bandwidth Memory
- HPC: High-performance Computing
- HPU: Habana Gaudi AI Processor Unit
- IPU: Intelligence Processing Unit
- MME: Matrix Multiplication Engine
- QPU: Quantum Processing Unit
- RDU: Reconfigurable Dataflow Unit
- TPU: Tensor Processing Unit

The most important thing to understand

I will make the following statement multiple times in this book - and that it's not enough to buy/rent the most expensive accelerators and expect a high return on investment (ROI).

The two metrics for a high ROI for ML training are:

1. the speed at which the training will finish, because if the training takes 2-3x longer than planned, your model could become irrelevant before it was released - time is everything in the current super-competitive ML market.
2. the total $$ spent to train the model, because if the training takes 2-3x longer than planned, you will end up spending 2-3x times more.

Unless the rest of the purchased/rented hardware isn't chosen carefully to match the required workload chances are very high that the accelerators will idle a lot and both time and $$ will be lost. The most critical component is network, then storage, and the least critical ones are (CPU and CPU memory).

If the compute is rented one usually doesn't have the freedom to choose - the hardware is either set in stone or some components might be replaceable but with not too many choices. Thus there are times when the chosen cloud provider doesn't provide a sufficiently well matched hardware, in which case it's best to seek out a different provider.

If you purchase your servers then I recommend to perform a very indepth due diligence before buying.

Besides hardware, you, of course, need software that can efficiently deploy the hardware.

We will discuss both the hardware and the software aspects in various chapters of this book. You may want to start here and here.

What Accelerator characteristics do we care for

Let's use the NVIDIA A100 spec as a reference point in the following sections.

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s
Max Thermal Design Power (TDP)	300W	400W ***
Multi-Instance GPU	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe Dual-slot air-cooled or single-slot liquid-cooled	SXM
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600 GB/s ** PCIe Gen4: 64 GB/s	NVLink: 600 GB/s PCIe Gen4: 64 GB/s
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs	NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs

source

TFLOPS

As mentioned earlier most of the work that ML training and inference do is matrix multiplication. If you remember your algebra matrix multiplication is made of many multiplications followed by summation. Each of these computations can be counted and define how many of these operations can be performed by the chip in a single seconds.

This is one of the key characteristics that the accelerators are judged by. The term TFLOPS defines how many trillions of FloatingPointOperations the chip can perform in a second. The more the better. There is a different definition for different data types. For example, here are a few entries for A100:

Data type	TFLOPS	w/ Sparsity
FP32	19.5	n/a
Tensor Float 32 (TF32)	156	312
BFLOAT16 Tensor Core	312	624
FP16 Tensor Core	312	624
INT8 Tensor Core	624	1248

footnote: INT8 is measured in TeraOperations as it's not a floating operation.

footnote: the term FLOPS could mean either the total number of FloatingPointOperations, e.g. when counting how many FLOPS a single Transformer iteration takes, and it could also mean FloatingPointOperations per second - so watch out for the context. When you read an accelerator spec it's almost always a per second definition. When model architectures are discussed it's usually just the total number of FloatingPointOperations.

So you can see that int8 is 2x faster than bf16 which in turn is 2x faster than tf32.

Moreover, the TFLOPs depend on the matrices size as can be seen from this table:

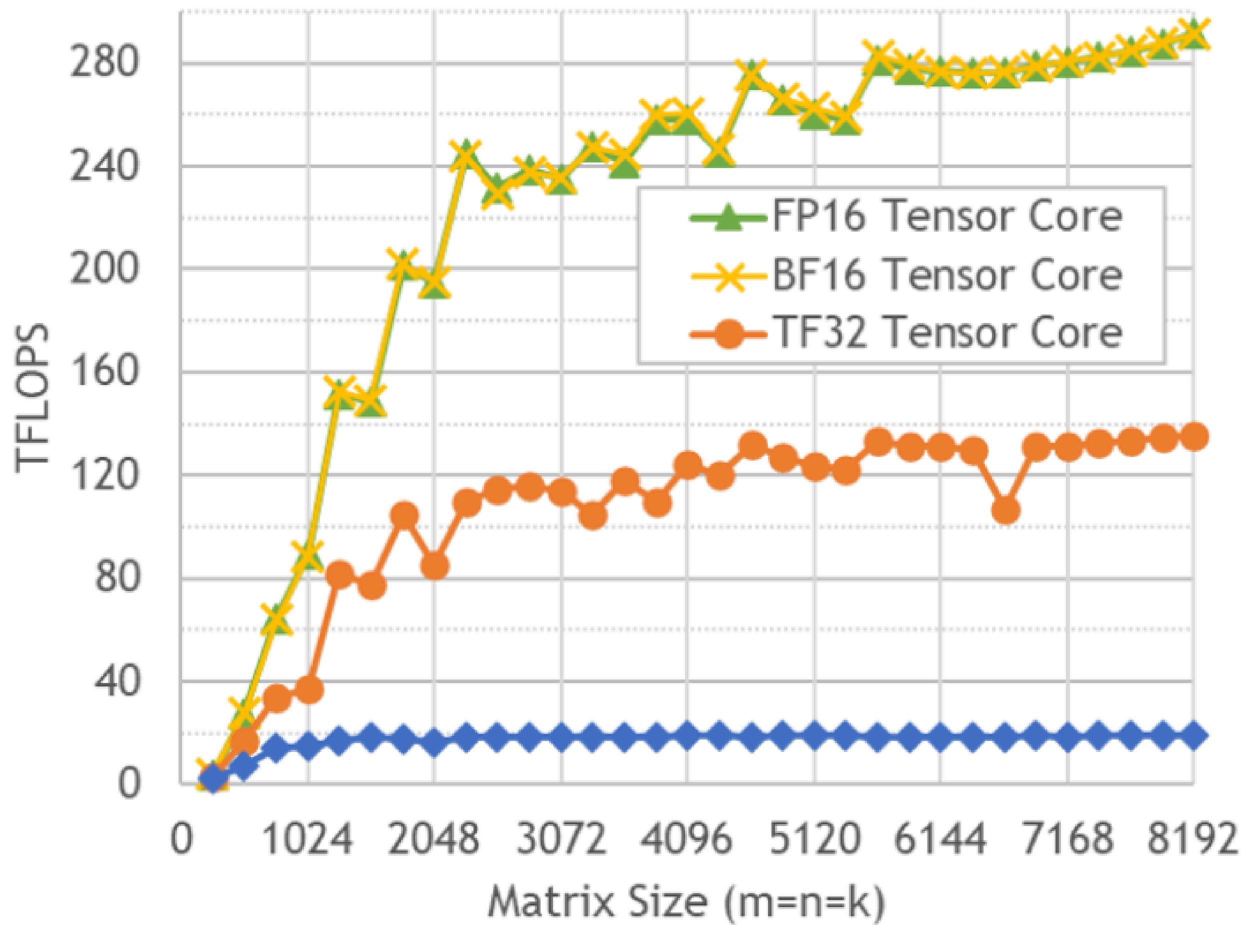


Figure 6. Mixed-precision matrix multiply on A100 with cuBLAS.

source

As you can see the difference in performance is non-linear due to the tile and wave quantization effects.

Let's look at the TFLOPS specs across the high end accelerators:

Accelerator / TFLOPS	fp32	fp16	fp8	int8
NVIDIA A100 SXM	19.5	312	624	624
AMD MI250	45.3	362	X	362
AMD MI250X	47.9	383	X	383
NVIDIA H100 SXM	67.0	989	1979	1979
NVIDIA H100 PCIe	51.0	756	1513	1513
NVIDIA H100 dual NVL	134.0	989	3958	3958

Accelerator / TFLOPS	fp32	fp16	fp8	int8
AMD MI300	?	?	?	?

- Intel Gaudi2 doesn't plan to publish TFLOPS specs as of this writing

Achievable peak TFLOPS

The problem with the advertised peak TFLOPS is that they are **very** theoretical and can't be achieved in practice even if all the perfect conditions have been provided. Each accelerator has its own realistic TFLOPS which is not advertised and there are anecdotal community reports that do their best to find the actual best value, but I'm yet to find any official reports.

If you find solid reports (papers?) showing the actual TFLOPS one can expect from one or more of the high end accelerators discussed in this chapter please kindly submit a PR with this information. The key is to have a reference to a source that the reader can validate the proposed information with.

To provide a numerical sense to what I'm talking about is let's take A100 with its 312 TFLOPS bf16 peak performance in the specs of this card. Until the invent of FlashAttention it was known that 150TFLOPS was close to the highest one could get for half precision mixed precision, with FlashAttention, it's around 180TFLOPS. This is, of course, measured for training LLMs where the network and IO are involved which create additional overheads. So here the peak performance probably lays somewhere between 200 and 300 TFLOPS.

It should be possible to calculate the actual peak TFLOPS by doing a perfectly aligned max-size matrices `matmul` measured on a single accelerator.

XXX: write a small program to do exactly dynamically figuring out the perfect shapes based on [the tile and wave quantization effects] and max sizes (how?) so that the benchmark isn't hardcoded to a particular accelerator.

Accelerator memory size and speed

Typically the more on-chip memory the accelerator has the better. At any given time usually most of the model weights aren't being used as they wait for their turn to be processed and thus large memory allows more of the model to be on the accelerator memory and immediately available for access and update. When there is not enough memory, sometimes the model has to be split across multiple accelerators, or offloaded to CPU and/or disk.

Current high end accelerators (some aren't GA yet):

Accelerator	Memory in GBs	Type	Speed in TB/s
NVIDIA A100 SXM	80	HBM2e	2
NVIDIA H100 SXM	80	HBM3	3.35
NVIDIA H100 PCIe	80	HBM3	2
NVIDIA H100 dual NVL	188	HBM3	7.8
AMD MI250	128	HBM2e	3.28
AMD MI250X	128	HBM2e	3.28
AMD MI300	192	HBM3	

- XXX: add other accelerators

Memory speed is, of course, very important since if it's not fast enough than the compute ends up idling waiting for the data to be moved to and from the memory.

The GPUs use High Bandwidth Memory (HBM) which is a 3D version of SDRAM memory. For example, A100-SXM comes with HBM2 at 1.6TB/sec, and H100-SXM comes with HBM3 at 3.35TB/s.

Heat

This is of interest when you buy your own hardware, when you rent on the cloud the provider hopefully takes care of adequate cooling.

The only important practical understanding for heat is that if the accelerators aren't kept cool they will throttle their compute clock and slow everything down (and could even crash sometimes, albeit throttling is supposed to prevent that).

High end accelerators for LLM/VLM workloads

Cloud and in-house accelerators

Most common accelerators that can be either rented on compute clouds or purchased:

NVIDIA:

- A100 - huge availability but already getting outdated.
- H100 - 2-3x faster than A100 (half precision), 6x faster for fp8, slowly emerging on all major clouds.
- GH200 - 2 chips on one card - (1) H100 w/ 96GB HBM3 or 144GB HBM3e + (2) Grace CPU w/ 624GB RAM - availability is unknown.

AMD:

- MI250 ~= A100 - very few clouds have them
- MI300 ~= H100 - don't expect until late-2024 or even 2025 to be GA

Intel:

- Gaudi2 ~= H100 - Currently there is a very low availability on cloud.google.com with a long waiting list which supposedly should be reduced in Q1-2024. AWS has the older Gaudi1 via DL1 instances.

Graphcore:

- IPU - available via Paperspace

SambaNova:

- DataScale SN30

In-house accelerator clusters

Cerebras:

- clusters
- systems based on WaferScale Engine (WSE).

Cloud-only solutions

These can be only used via clouds:

Google

- TPUs - lock-in, can't switch to another vendor like NVIDIA -> AMD

Cerebras:

- Cloud

Prices

Remember that the advertised prices are almost always open to negotiations as long as you're willing to buy/rent in bulk and if renting then for a long time (i.e. years!). When do you will discover that the actual price that you end up paying could be many times less than the original public price. Some cloud providers already include the discount as you choose a longer commitment on their website, but it's always the best to negotiate directly with their sales team. In addition or instead of a $$-discount you could be offered some useful features/upgrades for free.

For the baseline prices it should be easy to find a few good sites that provide an up-to-date public price comparisons across clouds - just search for something like cloud gpu pricing comparison.

Accelerators in detail

NVIDIA

Abbreviations:

- CUDA: Compute Unified Device Architecture (proprietary to NVIDIA)

NVIDIA-specific key GPU characteristics:

- CUDA Cores - similar to CPU cores, but unlike CPUs that typically have 10-100 powerful cores, CUDA Cores are weaker and come in thousands and allow to perform massive general purpose computations (parallelization). Like CPU cores CUDA Cores perform a single operation in each clock cycle.
- Tensor Cores - special compute units that are designed specifically to perform fast multiplication and addition operations like matrix multiplication. These perform multiple operations in each clock cycle. They can execute extremely fast computations on low or mixed precision data types with some loss (fp16, bf16, tf32, fp8, etc.). These cores are specifically designed for ML workloads.
- Streaming Multiprocessors (SM) are clusters of CUDA Cores, Tensor Cores and other components.

For example, A100-80GB has:

- 6912 CUDA Cores
- 432 Tensor Cores (Gen 3)
- 108 Streaming Multiprocessors (SM)

AMD

AMD-specific key GPU characteristics:

- Stream Processors - are similar in functionality to CUDA Cores - that is these are the parallel computation units. But they aren't the same, so one can't compare 2 gpus by just comparing the number of CUDA Cores vs the number of Stream Processors.
- Compute Units - are clusters of Stream Processors and other components

for example, AMD MI250 has:

- 13,312 Stream Processors
- 208 Compute Units

Intel Gaudi2

Architecture

- 24x 100 Gigabit Ethernet (RoCEv2) integrated on chip - 21 of which are used for intra-node and 3 for inter-node (so `21*8=168` cards for intra-node (262.5GBps per GPU), and `3*8=24` cards for inter-node (2.4Tbps between nodes)

- 96GB HBM2E memory on board w/2.45 TBps bandwidth per chip, for a total of 768GB per node

A server/node is built from 8 GPUs, which can then be expanded with racks of those servers.

There are no official TFLOPS information published (and from talking to an Intel representative they have no intention to publish any.) They publish the [following benchmarks](https://developer.habana.ai/resources/habana-models-performance/ but I'm not sure how these can be used to compare this compute to other providers.

Comparison: supposedly Gaudi2 competes with NVIDIA H100

API

NVIDIA

uses CUDA

AMD

uses ROCm

Intel Gaudi

The API is via Habana SynapseAI® SDK which supports PyTorch and TensorFlow.

Useful integrations:

- HF Optimum Habana which also includes - DeepSpeed integration.

Apples-to-apples Comparison

It's very difficult to compare specs of different offerings since marketing tricks get deployed pretty much by all competitors so that one can't compare 2 sets of specs and know the actual difference.

- MLPerf via MLCommons publishes various hardware benchmarks that measure training, inference, storage and other tasks' performance. For example, here is the most recent as of this writing training v3.0 and inference v3.1 results.

  Except I have no idea how to make use of it - it's close to impossible to make sense of or control the view. This is a great intention lost in over-engineering and not thinking about how the user will benefit from it, IMHO. For example, I don't care about CV data, I only want to quickly see the LLM rows, but I can't do it. And then the comparisons are still not apples to apples so how can you possibly make sense of which hardware is better I don't know.

Troubleshooting NVIDIA GPUs

Xid Errors

No hardware is perfect, sometimes due to the manufacturing problems or due to tear and wear (especially because of exposure to high heat), GPUs are likely to encounter various hardware issues. A lot of these issues get corrected automatically without needing to really understand what's going on. If the application continues running usually there is nothing to worry about. If the application crashes due to a hardware issue it's important to understand why this is so and how to act on it.

A normal user who uses a handful of GPUs is likely to never need to understand GPU-related hardware issues, but if you come anywhere close to massive ML training where you are likely to use hundreds to thousands of GPUs it's certain that you'd want to understand about different hardware issues.

In your system logs you are likely to see occasionally Xid Errors like:

```
NVRM: Xid (PCI:0000:10:1c): 63, pid=1896, Row Remapper: New row marked for remapping, reset gpu to
activate.
```

To get those logs one of the following ways should work:

```
sudo grep Xid /var/log/syslog
sudo dmesg -T | grep Xid
```

Typically, as long as the training doesn't crash, these errors often indicate issues that automatically get corrected by the hardware.

The full list of Xid Errors and their interpretation can be found here.

You can run `nvidia-smi -q` and see if there are any error counts reported. For example, in this case of Xid 63, you will see something like:

```
Timestamp                                 : Wed Jun  7 19:32:16 2023
Driver Version                            : 510.73.08
CUDA Version                              : 11.6

Attached GPUs                             : 8
GPU 00000000:10:1C.0
    Product Name                          : NVIDIA A100-SXM4-80GB
    [...]
    ECC Errors
        Volatile
            SRAM Correctable              : 0
            SRAM Uncorrectable            : 0
            DRAM Correctable              : 177
            DRAM Uncorrectable            : 0
        Aggregate
```

```
            SRAM Correctable              : 0
            SRAM Uncorrectable            : 0
            DRAM Correctable              : 177
            DRAM Uncorrectable            : 0
    Retired Pages
        Single Bit ECC                    : N/A
        Double Bit ECC                    : N/A
        Pending Page Blacklist            : N/A
    Remapped Rows
        Correctable Error                 : 1
        Uncorrectable Error               : 0
        Pending                           : Yes
        Remapping Failure Occurred        : No
        Bank Remap Availability Histogram
            Max                           : 639 bank(s)
            High                          : 1 bank(s)
            Partial                       : 0 bank(s)
            Low                           : 0 bank(s)
            None                          : 0 bank(s)
[...]
```

Here we can see that Xid 63 corresponds to:

```
ECC page retirement or row remapping recording event
```

which may have 3 causes: HW Error / Driver Error / FrameBuffer (FB) Corruption

This error means that one of the memory rows is malfunctioning and that upon either reboot and/or a gpu reset one of the 640 spare memory rows (in A100) will be used to replace the bad row. Therefore we see in the report above that only 639 banks remain (out of 640).

The Volatile section of the `ECC Errors` report above refers to the errors recorded since last reboot/GPU reset. The Aggregate section records the same error since the GPU was first used.

Now, there are 2 types of errors - Correctable and Uncorrectable. The correctable one is a Single Bit ECC Error (SBE) where despite memory being faulty the driver can still recover the correct value. The uncorrectable one is where more than one bit is faulty and it's called Double Bit ECC Error (DBE). Typically, the driver will retire whole memory pages if 1 DBE or 2 SBE errors occur at the same memory address. For full information see this document

A correctable error will not impact the application, a non-correctable one will crash the application. The memory page containing the uncorrectable ECC error will be blacklisted and not accessible until the GPU is reset.

If there are page scheduled to be retired you will see something like this in the output of `nvidia-smi -q`:

```
    Retired pages
        Single Bit ECC             : 2
        Double Bit ECC             : 0
        Pending Page Blacklist    : Yes
```

Each retired page decreases the total memory available to applications. But each page is only 4MB large, so it doesn't

reduce the total available GPU memory by much.

To dive even deeper into the GPU debugging, please refer to [this document](#) - it includes a useful triage chart which helps to determine when to RMA GPUs. This document has additional information about Xid 63-like errors

For example it suggests:

> If associated with XID 94, the application that encountered the error needs to be restarted. All other applications on the system can keep running as is until there is a convenient time to reboot for row remapping to activate. See below for guidelines on when to RMA GPUs based on row remapping failures.

If after a reboot the same condition occur for the same memory address, it means that memory remapping has failed and Xid 64 will be emitted again. If this continues it means you have a hardware issue that can't be auto-corrected and the GPU needs to RMA'ed.

At other times you may get Xid 63 or 64 and the application will crash. Which usually will generate additional Xid errors, but most of the time it means that the error was uncorrectable (i.e. it was a DBE sort of an error and then it'll be Xid 48).

As mentioned earlier to reset a GPU you can either simply reboot the machine, or run:

```
nvidia-smi -r -i gpu_id
```

where `gpu_id` is the sequential number of the gpu you want to reset. Without `-i` all GPUs will be reset.

Running diagnostics

If you suspect one or mode NVIDIA GPUs are broken on a given node, `dcgmi` is a great tool to quickly find any bad GPUs.

NVIDIA® Data Center GPU Manager (DCGM) is documented here and can be downloaded from here.

Here is an example slurm script that will run very in-depth diagnostics (`-r 3`), which will take about 10 minutes to complete on an 8-GPU node:

```
$ cat dcgmi-1n.slurm
#!/bin/bash
#SBATCH --job-name=dcgmi-1n
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=96
#SBATCH --gres=gpu:8
#SBATCH --exclusive
#SBATCH --output=%x-%j.out

set -x -e
echo "START TIME: $(date)"
srun --output=%x-%j-%N.out dcgmi diag -r 3
echo "END TIME: $(date)"
```

Now to run it on specific nodes of choice:

```
sbatch --nodelist=node-115 dcgmi-1n.slurm
sbatch --nodelist=node-151 dcgmi-1n.slurm
sbatch --nodelist=node-170 dcgmi-1n.slurm
```

edit the nodelist argument to point to the node name to run.

If the node is drained or downed and you can't launch a slurm job using this node, just ssh into the node and run the command directly on the node:

```
dcgmi diag -r 3
```

If the diagnostics didn't find any issue, but the application still fails to work, re-run the diagnostics with level 4, which will now take more than 1 hour to complete:

```
dcgmi diag -r 4
```

For example, if you run into a repeating Xid 64 error it's likely that the diagnostics report will include:

```
+---------------------------+------------------------------------------------+
| Diagnostic                | Result                                         |
+===========================+================================================+
|-----  Deployment  --------+------------------------------------------------|
| Error                     | GPU 3 has uncorrectable memory errors and row  |
|                           |  remappings are pending                        |
```

so you now know to RMA that problematic GPU, if remapping fails.

The `dcgmi` tool contains various other levels of diagnostics, some of which complete in a matter of a few minutes and can be run as a quick diagnostic in the epilogue of SLURM jobs to ensure that the node is ready to work for the next SLURM job, rather than discovering that after the user started their job and it crashed.

When filing an RMA report you will be asked to run `nvidia-bug-report` script, the output of which you will need to submit with the RMA request.

I usually save the log as well for posterity using one of:

```
dcgmi diag -r 3 | tee -a dcgmi-r3-`hostname`.txt
dcgmi diag -r 4 | tee -a dcgmi-r4-`hostname`.txt
```

How to detect if a node is missing GPUs

If you got a new VM, there are odd cases where there is less than expected number of GPUs. Here is how you can quickly test you have got 8 of them:

```
cat << 'EOT' >> test-gpu-count.sh
```

```
#!/bin/bash

set -e

# test the node has 8 gpus
test $(nvidia-smi -q | grep UUID | wc -l) != 8 && echo "broken node: less than 8 gpus" && false
EOT
```

and then:

```
bash test-gpu-count.sh
```

How to detect if you get the same broken node again and again

This is mostly relevant to cloud users who rent GPU nodes.

So you launched a new virtual machine and discovered it has one or more broken NVIDIA GPUs. You discarded it and launched a new and the GPUs are broken again.

Chances are that you're getting the same node with the same broken GPUs. Here is how you can know that.

Before discarding the current node, run and log:

```
$ nvidia-smi -q | grep UUID
    GPU UUID                              : GPU-2b416d09-4537-ecc1-54fd-c6c83a764be9
    GPU UUID                              : GPU-0309d0d1-8620-43a3-83d2-95074e75ec9e
    GPU UUID                              : GPU-4fa60d47-b408-6119-cf63-a1f12c6f7673
    GPU UUID                              : GPU-fc069a82-26d4-4b9b-d826-018bc040c5a2
    GPU UUID                              : GPU-187e8e75-34d1-f8c7-1708-4feb35482ae0
    GPU UUID                              : GPU-43bfd251-aad8-6e5e-ee31-308e4292bef3
    GPU UUID                              : GPU-213fa750-652a-6cf6-5295-26b38cb139fb
    GPU UUID                              : GPU-52c408aa-3982-baa3-f83d-27d047dd7653
```

These UUIDs are unique to each GPU.

When you then re-created your VM, run this command again - if the UUIDs are the same - you know you have the same broken GPUs.

Sometimes just rebooting the node will get new hardware. In some situations you get new hardware on almost every reboot, in other situations this doesn't happen. And this behavior may change from one provider to another.

If you keep on getting the same broken node - one trick to overcoming this is allocating a new VM, while holding the broken VM running and when the new VM is running - discarding the broken one. That way you will surely get new GPUs - except there is no guarantee they won't be broken as well. If the use case fits consider getting a static cluster where it's much easier to keep the good hardware.

This method is extra-crucial for when GPUs don't fail right away but after some use so it is non-trivial to see that there is a problem. Even if you reported this node to the cloud provider the technician may not notice the problem right away and put the bad node back into circulation. So if you're not using a static cluster and tend to get random VMs on demand you may want to keep a log of bad UUIDs and know you have got a lemon immediately and not 10 hours into the node's use.

Cloud providers usually have a mechanism of reporting bad nodes. Therefore other than discarding a bad node, it'd help

yourself and other users to report bad nodes. Since most of the time users just discard the bad nodes, the next user is going to get them. I have seen users getting a very high percentage of bad nodes in some situations.

Inter-node and intra-node Networking Hardware

This chapter is a WIP

It's not enough to buy/rent expensive GPUs to train/infer models fast. You need to ensure that your IO, CPU and Network are fast enough to "feed the GPU furnace". If this is not ensured then the expensive GPUs will be underutilized leading to lost $$, slower training time and inference. While it can be any other of the mentioned components, the network is most of the time what causes the bottleneck in the training (assume your DataLoader is fast).

If your model fits on a single GPU, you have little to worry about. But nowadays most models require several GPUs to load and LLM/VLM models require multiple GPU nodes for training and some even for inference.

Most GPU nodes contain 8 GPUs, some 4 and recently there are some that have one super-GPU per node.

When the model spans several GPUs and doesn't leave a single node all you need to worry about is fast Intra-node networking. As soon as the model requires several nodes, which is often the case for training as one can use multiple replicas to parallelize and speed up the training, then fast Inter-node networking becomes the key.

This article covers both types of networking hardware, reports their theoretical and effective speeds and explains how they inter-play with each other.

Glossary

- DMA: Direct Memory Access
- EFA: Elastic Fabric Adapter
- HCA: Host Channel Adapter
- IB: Infiniband
- MFU: Model Flops Utilization (e.g. `mfu=0.5` at half-precision on A100 comes from getting 156TFLOPs, because peak half-precision spec is 312TFLOPS, and thus `156/312=0.5`)
- NIC: Network Interface Card
- OPA: Omni-Path Architecture
- RoCE: RDMA over Converged Ethernet
- RoE: RDMA over Ethernet
- VPI: Virtual Protocol Interconnect
- RDMA: Remote Direct Memory Access

Speed-related:

- Bi-directional, Duplex: a transmission from one point to another in both directions A <-> B, typically 2x speed of unidirectional
- GBps, GB/s: Gigabytes per secs (1GBps = 8Gbps) transferred in a channel
- GT/s: GigaTransfers per second - the number of operations transferring data that occur in each second.
- Gbps, Gb/s: Gigabits per secs (1Gbps = 1/8GBps) transferred in a channel
- Unidirectional: a transmission from one point to another in one direction A -> B

Understanding why inter-node network speed is of a huge importance

This is probably one of the most important multi-segment section that you really want to understand well. While it seeks out to show how important the inter-node speed is, to build up the case it'll teach on the way many important training-related concepts.

The basics

First, let's get a bit of a feeling what all those Gbps/GBps practically mean.

If your model is 80B parameter large, and you need to transmit every parameter or a gradient on the network even once in float32 (fp32) format, which requires 4 bytes per parameter, so you need to send `80*4` 320GB of data, or 2560Gb (`*8`). If your network's bandwidth is 200Gbps it will take 12.8 seconds (`2560/200`) to transmit. And if you had 1600Gbps network then it'd take only 1.6 seconds. Why does it matter?

1-GPU training

Let's start with a much smaller model of say 2B params, to train it you'd need at least 18 bytes per parameter in mixed half precision. So `18*2` 36GB of memory just for model weights, optimizer states and gradients. Plus you need additional memory for activations and it'll depend on the batch size and sequence length. But with 80GB A100 GPU we can definitely train this model on a single GPU.

We then assume for the moment that the DataLoader is fast enough to be negligible in duration compared to the compute time. And thus we get a close to a perfect MFU (Model FLOPs Utilization):

```
[DL][  compute  ][DL][  compute  ][DL][  compute  ]
---------------------------------------------------> time
|<--iteration-->||<--iteration-->||<--iteration-->|
```

which means that the GPU just needs to do many matmuls and it'd do it amazing fast. In this situation you get the highest ROI (Return on Investment).

Single node training

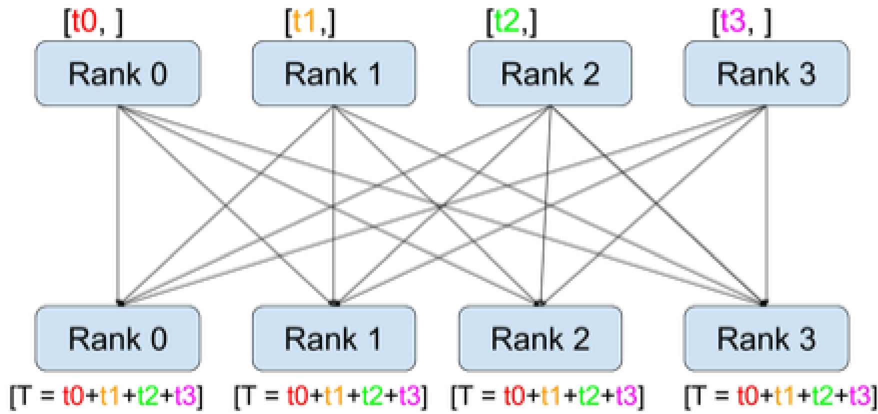
The previous situation was fantastic due to the close to perfect MFU, but you realize that the training on a single GPU is going to take quite some time, since we are in AI race you'd probably want to finish the training sooner than later. So you'd ask - can I train the model on 8 GPUs instead, and the answer would be - yes, of course. With one caveat - at the end of each iteration you'd need to sync the gradients between the 8 processes (each process for a single GPU), so that each participating process of the training can benefit from what the other 7 have learned during the last iteration.

footnote: You could, of course, use less than 8 GPUs, it is just that most NVIDIA GPU-based compute nodes these days have 8 GPUs so why not get the best return on investment.

footnote: in the ideal world the training on 1 gpu for 8 durations of time, should cost the same as training on 8 gpus for 1 duration of time. That's one would expect to spend the same $$ and to finish 8 times faster. But because of data synchronization requirements.

If the experimental model still contains 2B params like in the previous section and grads are in fp32 then the training program needs to send 8GB (`2G * 4B`) of data on every iteration. Moreover, since syncing the gradients requires an `all_reduce` collective collective - it needs to transmit the data twice - the first time sending the gradient data by each gpu, computing the sum of gradients and send this value back to each participating gpu so that each training process will benefit from the learning advancements each of its peers made in the last iteration.

Here is the all-reduce collective visualized:



(source)

So we need to send 8GB twice, which means we need to send 16GB of data.

footnote: and to be exact the 2x comms volume for all-reduce is really `2*(n-1)/n` where n is the number of participating gpus. So if n=2, the coefficient is just 1 since `2*(2-1)/2=1` and 1.75 for n=8 since `2*(8-1)/8=1.75` and it becomes already very close to 2 at n=64.

footnote: there is also the important issue of latency of the network - which is multiplied several times due to how data is gathered from all participating gpus. But, given that here we are moving a very large payload the latency contributes a very small overhead and for simplicity can be ignored.

How long will it take to send 16GB of data?

- A100 @ 300GBps: `16/300` = 0.053 secs
- H100 @ 450GBps: `16/450` = 0.035 secs

which is incredibly fast!

And here is how our timeline will look like:

```
[DL][  compute ][comms][DL][  compute ][comms][DL][  compute ][comms]|
-----------------------------------------------------------------------> time
|<---- iteration ---->||<---- iteration ---->||<---- iteration ----->|
```

oh and this whole synchronization protocol is called DDP (DistributedDataParallel) in the PyTorch lingo.

Comms and compute overlap

Even with this really fast comms the network still creates a bottleneck and leads to a short idling of the gpus. To solve this issue the advanced algorithms implement an overlap of comms and compute. Until now we approached the problem as one single transmission, but in reality each model is made of many layers and each layer can transmit the gradients it has computed, while the next layer is computing its gradients. So if you look at the level of the model, what happens in the `backward` path is:

```
[   compute   ][   compute   ][   compute   ]
               [comms]        [comms]        [comms]
---------------------------------------------> time
<- layer -1 ->|<- layer -2 ->|<- layer -3 ->|
```

so once the last layer (-1) computed its gradients it all-reduces them while the 2nd to last layer performs its `backward`, and so on, until the first layer finished with gradients and it finally sends its gradients out.

So now you understand how overlapping works, So we can now update our bigger picture diagram to be:

Now our timing diagram becomes very similar to the diagram we had for a single gpu:

```
[DL][  compute  ][DL][  compute  ][DL][  compute  ]
[  comms ]       [  comms]        [  comms]
---------------------------------------------------> time
|<--iteration-->||<--iteration-->||<--iteration-->|
```

and we hope that comms are faster than DL+compute, since if they aren't faster than we have the following gpu idling gaps:

```
[DL][  compute  ][idle][DL][  compute  ][idle][DL][  compute  ][idle]
[         comms       ][         comms       ][         comms       ]
----------------------------------------------------------------------> time
|<---  iteration  --->||<---  iteration  --->||<---  iteration  --->|
```

Calculating TFLOPS

Calculating TFLOPS answers the question of how long will it take to perform a compute.

There is a bit of nomenclature confusion here as TFLOPS as the final `s` sometimes means `sec` and at other times just `ops`.

For example, when you read, the A100 spec the TFLOPS there means TeraFloatingPointOperations per second.

So let's define these abbreviations exactly:

- TFLOPS - TeraFLoatingpointOPerations per Second (another way is TFLOP/s)
- TFLOP - TeraFLoatingpointOPerations (or TFLOPs - lower case `s` but it's already confusing)

Also see the wiki page for more clarifications.

For GPT-family of decoder transformers models we can use the math described in this BLOOM-176 docs:

Here is how many TFLOP are processed per second:

```
tflops = model_size_in_B * 4 * 2 * seqlen * global_batch_size / (time_in_sec_per_interation * total_gpus * 1e3)
```

This formula assume one uses activation recomputation which saves GPU memory while introducing a smallish overhead. If one doesn't use it then replace `4` with `3` as the model has to do only 1x compute per `forward` and 2x per `backward` (since the grads are calculated twice - once for inputs and once for weights). With activation recomputation the `forward` is done twice and thus you have an additional path which leads to a multiplier of `4` instead of `3`

footnote: activation recomputation and gradient checkpointing both refer to the same technique.

so let's remove the time component, which will give us the total TFLOP

```
tflop = model_size_in_B * 4 * 2 * seqlen * global_batch_size / (total_gpus * 1e3)
```

So let's say we have:

- `seqlen=2048` (sequence length)
- `global_batch_size=16`

and we already defined:

- `total_gpus=8`
- `model_size_in_B=2`

This gives us:

```
tflops = 2 * 4 * 2 * 2048 * 16 / (8 * 1e3) = 65.536 TFLOP
```

So if we do a mixed half-precision training and most of the operations are done in half-precision then we can roughly say that we do [312 TFLOPS on A100](https://www.nvidia.com/en-us/data-center/a100/) and usually a well optimized framework on a well-tuned hardware will do at least 50% MFU - that is it'll be able to compute at about 1/2 peak performance.

footnote: It's a ~3x [989 TFLOPS on H100](https://www.nvidia.com/en-us/data-center/h100) (scroll to the end) and also it shows a misleading 2x numbers for sparsity so you have to mentally divide it by 2.

So continuing this train of thought it means that the setup will have about 156TFLOPS - and so it'll take 0.42 secs to process a single iteration (2x `forward` and 2x `backward` compute) if we ignore the overhead of the DataLoader (which we hope is close to instant).

Earlier we said that a typical A100 node has an intra-node NVLink connection of 300GBps, and thus we said that to send 16GB of grads will take `16/300` = 0.053 secs.

And we measured our compute to be 0.42 secs, so here we have a problem as `0.053 > 0.42` so the comms will be slower than compute and the network is a bottleneck.

You can now do several thought experiments - for example if you halve the batch size or the sequence length you will halve the compute time.

footnote: this is a very rough suggestions since GPUs work the fastest when the matrices they multiple are huge. But this is good enough for a simplified thought experiment we are having here. In reality halving the dimension will not halve the compute time.

OK, but hopefully at this point it's quite clear that if you remain at the boundaries of a single node, you don't need to worry about your GPUs idling.

But what if you want to speed up the training even more and throw say 4x 8-gpu nodes at it. (and of course you don't have a choice but to use multiple nodes if you have a much larger model). Suddenly, the comms can become an even bigger bottleneck.

Multiple node training

So here we are continuing with the idea of 2B param model and we will now use 32 gpus across 4 nodes to speed up the training even more.

While each group of 8 gpus is still connected with super-fast NVLink technology, the inter-node connections are usually in an order of magnitude slower.

Let's say you have a 200Gbps connection. Let's repeat the math from the previous section of how long it'll take to reduce 16GB of gradients.

16GB is 128Gb, and so at 200Gbps this will take 0.64 seconds.

And if stick to the compute taking 0.42 seconds, here we end up with comms taking longer than compute since `0.64 > 0.42`.

Let's bring both use cases together:

nodes	comms	compute	comms is a bottleneck
1	0.027	0.42	no
4	0.64	0.42	yes

on this 200Gbps inter-node setup the comms are 23x slower than the same performed on an intra-node NVlink connections.

In this case even though we still have the much faster NVLink connection, we don't really benefit from it, since the whole ensemble communicates at the speed of the slowest link. And that slowest link is the inter-node connection.

So in this particular situation if you were able to get a 400Gbps inter-node the speed would double and the comms will finish in 0.32 secs and thus will be faster than that 0.42 secs the compute would take.

footnote: you will never be able to get the advertised speed fully on the application level, so if it's advertised as 400Gbps in the best case expect to get 320Gbps (about 80%). So make sure to take this into the account as well. Moreover, depending on the payload of each collective - the smaller the payload the smaller the actual network throughput will be.

And remember this was all handling a pretty tiny as considered these days 2B param model.

Now do the same math with 20B and 200B parameter model and you will see that you need to have a much much faster inter-node connectivity to efficiently scale.

Large model training

Of course, when we train large models we don't use DDP, because we simply can't fit the whole model on a single gpu so various other techniques are used. The details are discussed in a dedicated chapter on Model Parallelism, but the only important thing to understand immediately is that all scalability techniques incur a much larger comms overhead, because they all need to communicate a lot more than just gradients. and therefore the amount of traffic on the network can easily grow 3x and more as compared to the DDP protocol overhead we have been exploring so far.

It can be difficult to do even approximate math as we did in this chapter, because the actual compute time depends on the efficiency of the chosen framework, how well it was tuned, how fast the DataLoader can feed the batches and many other things, therefore there is no standard MFU that one can use in the math and you will discover your MFU when you configure and run the first few steps of the large model training. and then you will read the Performance chapters and improve your MFU even more.

As I have shown in these sections it should be possible to be able to do a back-of-envelope calculations once you understand the specific scalability technique and its networking costs, so that you could know ahead of time which Inter-node network speed you need to require from your acquisition manager. Of course, you also need to understand the particular model architecture and calculate how many TFLOP it will take to do a single iteration.

Unidirectional vs Bidirectional (Duplex)

Most benchmarking / bandwidth measurement tools will report a unidirectional bandwidth. So be careful when you look at unidirectional vs. bidirectional (duplex) speeds. Typically the latter is ~2x faster.

If you measure the bandwidth on your setup and it's about 40% of the advertised speed, carefully check if the advertised speed said duplex and if so half that and then your measured bandwidth should now be about 80% which is expected.

case study: for a while I couldn't understand why when I run the nccl-tests all_reduce benchmark on an A100 node with advertised 600GBps intra-node speed I was getting only 235Gbps (40%) until Horace He kindly pointed out that I should be looking at unidirectional speed which is 300GBps, and then I get 80% of the theoretical spec which checks out.

Intra-node networking

Note to the reader: my notes currently include only NVIDIA intra-node hardware, since I'm yet to find access to AMD MI* nodes - if you are with AMD and you want me to extend this writing to AMD hardware please contact me with access to such hardware as it seems impossible to find.

On the server nodes with NVIDIA GPUs there is pretty much just 2 pieces of hardware - NVLink and NVSwitch. There is of course PCIe but it's about an order of magnitude slower so it's never used on modern GPU servers to perform GPU-to-GPU communications.

NVLink

- PCIe
- NVLink
- What Is NVLink blog post.

I found the wiki pages quite difficult to follow, so I will try to help bring clarity into this.

footnote: Pay attention that 1 GBps = 8 Gbps

Effective payload rate of Intra-node GPU-to-GPU communication hardware:

Interconnect	Lane/Direction	Lanes	Links	Unidirection	Duplex
NVlink 2	6.250 GBps	4	6	150 GBps	300 GBps
NVlink 3	6.250 GBps	4	12	300 GBps	600 GBps
NVlink 4	6.250 GBps	4	18	450 GBps	900 GBps

Interconnect	Lane/Direction	Lanes	Unidirection	Duplex
PCIe 4	~2.0 GBps	16	31 GBps	62 GBps
PCIe 5	~4.0 GBps	16	63 GBps	126 GBps
PCIe 6	~7.5 GBps	16	121 GBps	241 GBps
PCIe 7	~15.0 GBps	16	242 GBps	484 GBps

NVlink 2, 3 and 4 use the same hardware of 4 lanes of 6.250 GBps each per link. Each has a unidirectional bandwidth of 25GB/s per link, and therefore 50GB/s per duplex link. The only difference is in the number of links:

- NVLink 2 has 6 links => `25* 6`=> 150 GBps unidirectional and 300 GBps bi-directional

- NVLink 3 has 12 links => `25*12`=> 300 GBps unidirectional and 600 GBps bi-directional
- NVLink 4 has 18 links => `25*18`=> 450 GBps unidirectional and 900 GBps bi-directional

The largest PCIe 16x slot has 16 lanes. Smaller slots have less lanes, 1x == 1 lane.

As of this writing NVIDIA Hopper nodes typically come equipped with PCIe 5 and NVLink 4. So there NVlink is 7x faster than PCIe.

Let's look at some actual A100 and H100 nodes and correlate the theory with reality.

- A100 topology:

```
$ nvidia-smi topo -m
      GPU0  GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7  CPU Affinity  NUMA Affinity
GPU0   X    NV12  NV12  NV12  NV12  NV12  NV12  NV12   0-23         0
GPU1  NV12   X    NV12  NV12  NV12  NV12  NV12  NV12   0-23         0
GPU2  NV12  NV12   X    NV12  NV12  NV12  NV12  NV12   0-23         0
GPU3  NV12  NV12  NV12   X    NV12  NV12  NV12  NV12   0-23         0
GPU4  NV12  NV12  NV12  NV12   X    NV12  NV12  NV12  24-47         1
GPU5  NV12  NV12  NV12  NV12  NV12   X    NV12  NV12  24-47         1
GPU6  NV12  NV12  NV12  NV12  NV12  NV12   X    NV12  24-47         1
GPU7  NV12  NV12  NV12  NV12  NV12  NV12  NV12   X    24-47         1
```

You can see there are 12 NVLinks and 2 NUMA Groups (2 CPUs w/ 24 cores each)

- H100 topology:

```
$ nvidia-smi topo -m
      GPU0  GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7  CPU Affinity  NUMA Affinity
GPU0   X    NV18  NV18  NV18  NV18  NV18  NV18  NV18   0-51         0
GPU1  NV18   X    NV18  NV18  NV18  NV18  NV18  NV18   0-51         0
GPU2  NV18  NV18   X    NV18  NV18  NV18  NV18  NV18   0-51         0
GPU3  NV18  NV18  NV18   X    NV18  NV18  NV18  NV18   0-51         0
GPU4  NV18  NV18  NV18  NV18   X    NV18  NV18  NV18  52-103        1
GPU5  NV18  NV18  NV18  NV18  NV18   X    NV18  NV18  52-103        1
GPU6  NV18  NV18  NV18  NV18  NV18  NV18   X    NV18  52-103        1
GPU7  NV18  NV18  NV18  NV18  NV18  NV18  NV18   X    52-103        1
```

You can see there are 18 NVLinks and 2 NUMA Groups (2 CPUs w/ 52 cores each)

Of course, other A100 and H100s servers may be different, e.g. different number of cpu cores.

NVSwitch

NVSwitch can connect more than 8 GPUs at the speed of NVLink. It's advertised to connect up to 256 GPUs in the future generations of the switch.

The benefit of connecting more than 8 GPUs at the speed of NVLink is that it allows all-to-all GPU communications at a much faster speed than any intra-node hardware can provide. And with ever increasing compute speeds the network is the likely bottleneck leading to underutilized super-expensive GPUs.

For example, in the universe of Tensor Parallelism (Megatron), one doesn't use TP degree of more than 8, because TP is only efficient at NVLink speed. ZeRO-DP (Depspeed/FSDP) would also run much faster if the whole cluster uses NVLink

speed and involves no slow inter-node connections.

The current NVIDIA DGX H100 has a 3.6 TBps of full-duplex NVLink Network bandwidth provided by 72 NVLinks (NVLink 4). The normal NVlink 4 has 18 NVLinks (0.9 TBps duplex). So this setup has 4 switches (`18*4=72`) and therefore `0.9*4=3.6` TBps. Note, that this server has 8 GPUs, so here we get a much faster intra-node communications as compared to the standard NVlink 4.0 which provides only 0.9 TBps all-to-all connectivity for 8 GPUs.

NVIDIA DGX A100 has 6 switches of 12 NVlinks => 72.

DGX H100 SuperPOD combines 32 DGX H100 servers, for a total of 256 GPUs. It looks like here they use only half the NVLinks they used for a single DGX H100, so only 1.8 GBps per node, for a total of 57.6 GBps in total.

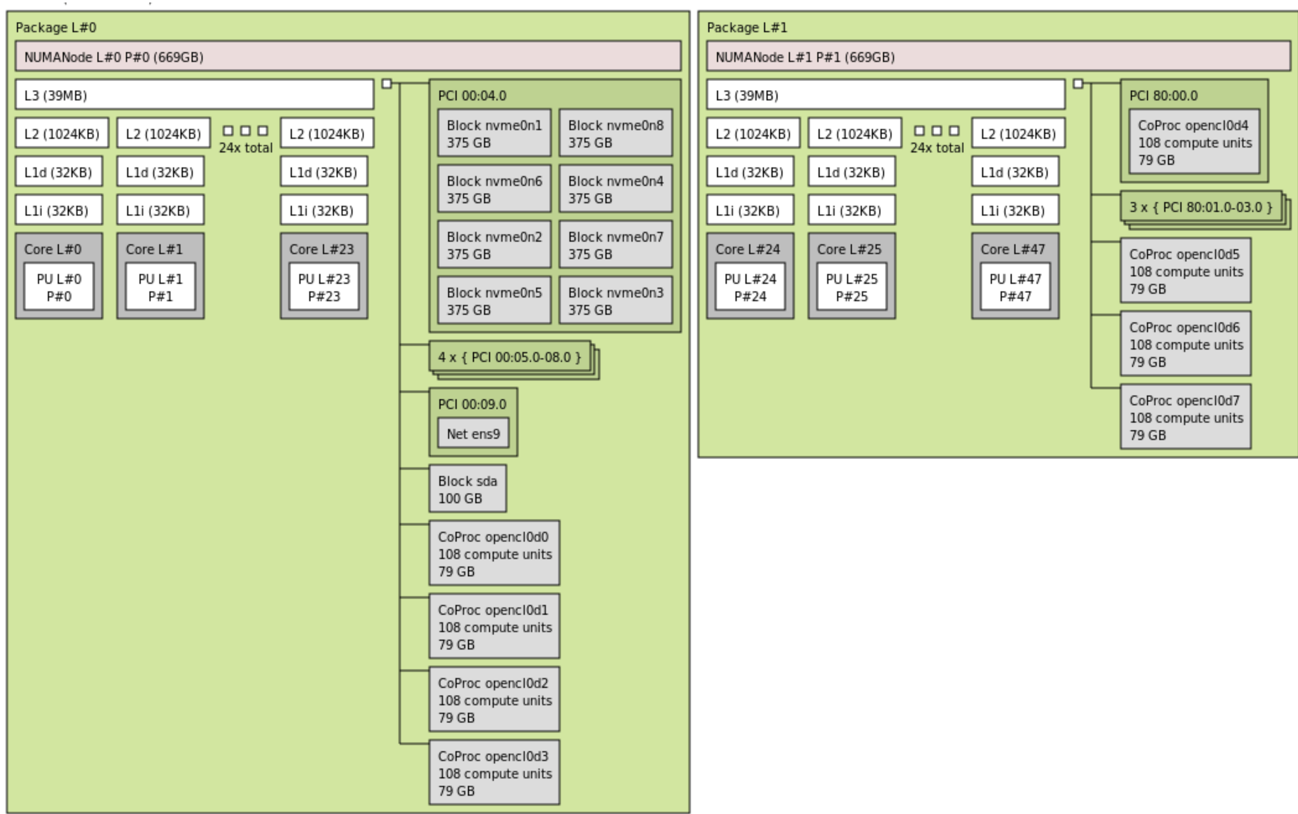
Gaudi2

According to Gaudi2 spec, these servers provide 8x 21 NICs of 100GbE RoCE v2 ROMA for a total of 2.1TBps and each card connected with each of the other 7 cards at 262.5 GBps.

NUMA Affinity

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. As modern servers have more than one CPU to get the best performance GPUs residing in the same block as the corresponding CPU should have the processes bound to that NUMA node.

Here is a typical A100 8x GPUs server, as visualized by hwloc:



As you can see it has 2 CPUs, each defining a NUMA block, and each such block contains a group of 4 GPUs. The GPUs are the grey blocks that say `CoProc` with 108 compute units (SMs) and 79GB of memory.

footnote: was generated by `lstopo a100.png`

Software Tools

note-to-self: probably belongs in its own chapter?

hwloc

https://github.com/open-mpi/hwloc

The Hardware Locality (hwloc) software project aims at easing the process of discovering hardware resources in parallel architectures. It offers command-line tools and a C API for consulting these resources, their locality, attributes, and interconnection. hwloc primarily aims at helping high-performance computing (HPC) applications, but is also applicable to any project seeking to exploit code and/or data locality on modern computing platforms.

Diagnostics: to take a snapshot of the server NUMA topology and save it as an image (supports many other formats)

```
lstopo a100.png
```

NUME node binding: `hwloc-bind` - binding processes, threads and memory

Bind an existing process to a specific numa node:

```
hwloc-bind --pid 1234 numa:0
```

similar software: `numactl`/`libnuma`

some suggestions in pytorch docs

Inter-node networking

As inter-node hardware is about of an order of magnitude slower than intra-node hardware in this universe Gbps are used instead of GBps. (1 GBps = 8 Gbps)

When it comes to inter-node networking hardware, there are the well established InfiniBand from NVIDIA and a few other players and there are many new comers that mainly are coming from compute cloud providers who can't compete on the slim margin renting out someone else's hardware so they build their own (EFA, and others not yet disclosed).

EFA

Elastic Fabric Adapter (EFA) is a recent technology created by AWS.

- EFA v1 0.4 Tbps (effective 340 Gbps for all_reduce tests) (P4 AWS instances)
- EFA v2 3.2 Tbps (since Q3-2023, P5 AWS instances)

InfiniBand

Now InfiniBand has been around for a few decades so there are many available configurations that can be found out there. So that if someone says they have InfiniBand that is insufficient information. What you need to know is the rate and the number of IB links.

Here the most recent signaling rates which you are likely to see in the current hardware offerings:

Signaling rate of uni-directional links in Gbps:

Links	EDR	HDR	NDR	XDR	GDR
1	25	50	100	200	400
4	100	200	400	800	1600
8	200	400	800	1600	3200
12	300	600	1200	2400	4800

Latency in usecs:

EDR	HDR	NDR	XDR	GDR
0.5	0.6	??	??	??

?? = NDR and later didn't publish latency data

InfiniBand provides RDMA.

Here are some examples of NVIDIA devices with the fastest IB:

- One configuration of NVIDIA DGX H100 comes with 8x NVIDIA ConnectX-7 Ethernet/InfiniBand ports each of 200Gbps, for a total of 1.6 Gbps to connect with other DGX servers.
- For DGX H100 SuperPOD the ConnectX-7s across all 32 DGX servers and associated InfiniBand switches provide 25.6 TBps of full duplex bandwidth for use within the pod or for scaling out the multiple SuperPODs - that is an equivalent of 0.8 TBps per node (6.4Tbps!).

Gaudi2

According to Gaudi2 spec, these servers provide 24 NICs of 100GbE RoCE v2 ROMA for a total of 2.4Tbps of inter-node connectivity with other Gaudi2 servers.

HPE Slingshot interconnect

HPE Slingshot interconnect seems to be used by HPCs. As of this writing it provides 200Gbps per link. Some HPCs use 4 of those links to build 800Gbps interconnects, and, of course, with more links will deliver a higher overall bandwidth.

OPA

OmniPath Architecture. Originally by Intel, the technology got sold to Cornelis Networks.

case study: I used this technology at JeanZay HPC in France in 2022. It was only 135Gbps and while the vendor tried to fix it a year later it was still the same speed. Hopefully the issue has been resolved and the speed is much faster nowadays. Because it was so slow we had to use Megatron-Deepspeed for training BLOOM-176B instead of the much easier to use DeepSpeed ZeRO).

As of this writing I see that the product comes with either 100 or 200Gbps bandwidth. So it's unlikely you will see anybody offering this solution for ML workloads, unless they manage to install many NICs perhaps?

Omni-Path provides RDMA.

Important nuances

Real network throughput

The network throughput in the advertised spec and the actual throughput will never be the same. In the best case you can expect about 80-90% of the advertised spec.

Then the network throughput will depend on the size of payload being sent during each communication. The higher the payload the higher the throughput will be.

Let's demonstrate this using nccl-tests on a single A100 node

```
$ ./build/all_reduce_perf -b 32k -e 16G -f 2 -g 8 -n 50
[...]
           size    time   algbw   busbw
            (B)    (us)  (GB/s)  (GB/s)
         32_768   43.83    0.75    1.31
         65_536   46.80    1.40    2.45
        131_072   51.76    2.53    4.43
        262_144   61.38    4.27    7.47
        524_288   80.40    6.52   11.41
       1048_576   101.9   10.29   18.00
       2097_152   101.4   20.68   36.18
      4_194_304   101.5   41.33   72.33
      8_388_608   133.5   62.82  109.93
     16_777_216   276.6   60.66  106.16
     33_554_432   424.0   79.14  138.49
     67_108_864   684.6   98.02  171.54
    134_217_728  1327.6  101.10  176.92
    268_435_456  2420.6  110.90  194.07
    536_870_912  4218.4  127.27  222.72
  1_073_741_824  8203.9  130.88  229.04
  2_147_483_648   16240  132.23  231.41
  4_294_967_296   32136  133.65  233.88
  8_589_934_592   64074  134.06  234.61
 17_179_869_184  127997  134.22  234.89
```

footnote: I massaged the output to remove unwanted columns and made the size more human readable

This benchmark run an `all_reduce` collective for various payload sizes from 32KB to 16GB. The value that we care about is the `busbw` - this column tells us the real network throughput as explained here.

As you can see for payloads smaller than 8MB the throughput is very low - and it starts saturating around payload size of 536MB. It's mostly because of latency. Reducing a single 4GB payload is much faster than 1000x 4MB payloads.

Here is a benchmark that demonstrates that: all_reduce_latency_comp.py. Let's run it on the same A100 node:

```
$ python -u -m torch.distributed.run --nproc_per_node=8 all_reduce_latency_comp.py

----------- 1x 4.0GB ----------------
 busbw: 1257.165 Gbps
```

```
----------- 1000x 0.004GB ----------------
 busbw: 374.391 Gbps
```

It's easy to see that it's about 3x slower in this particular case to send the same payload but in 1000 smaller chunks.

So when you calculate how long does it take to `all_reduce` a given payload size, you need to use the corresponding `busbw` entry (after of course you have run this benchmark on your particular hardware/environment).

Figuring out the payload can be tricky since it'd depend on the implementation of the framework. Some implementations will reduce each weight's gradient alone which obvious would lead to a very small payload and the network will be very slow. Other implementations bucket multiple gradients together before reducing those, increasing the payload and minimizing the latency impact.

But let's go back to the benchmark results table. This test was done on an A100 node that runs NVLink advertised as uni-directional 300GBs so we get about 78% of the theoretical speed with 17GB payload and more than that the benchmark crashes. It can be seen from the last few rows of the table that not much more can be squeezed.

We can also run p2pBandwidthLatencyTest which performs a low-level p2p benchmark:

```
./p2pBandwidthLatencyTest
[...]
Unidirectional P2P=Enabled Bandwidth (P2P Writes) Matrix (GB/s)
   D\D     0      1      2      3      4      5      6      7
     0 1581.48 274.55 275.92 272.02 275.35 275.28 273.62 273.20
     1 274.70 1581.48 275.33 272.83 275.38 273.70 273.45 273.70
     2 274.81 276.90 1594.39 272.66 275.39 275.79 273.97 273.94
     3 273.25 274.87 272.12 1545.50 274.38 274.37 274.22 274.38
     4 274.24 275.15 273.44 271.57 1584.69 275.76 275.04 273.49
     5 274.37 275.77 273.53 270.84 274.59 1583.08 276.04 273.74
     6 275.61 274.86 275.47 273.19 272.58 275.69 1586.29 274.76
     7 275.26 275.46 275.49 273.61 275.50 273.28 272.24 1591.14
[...]
```

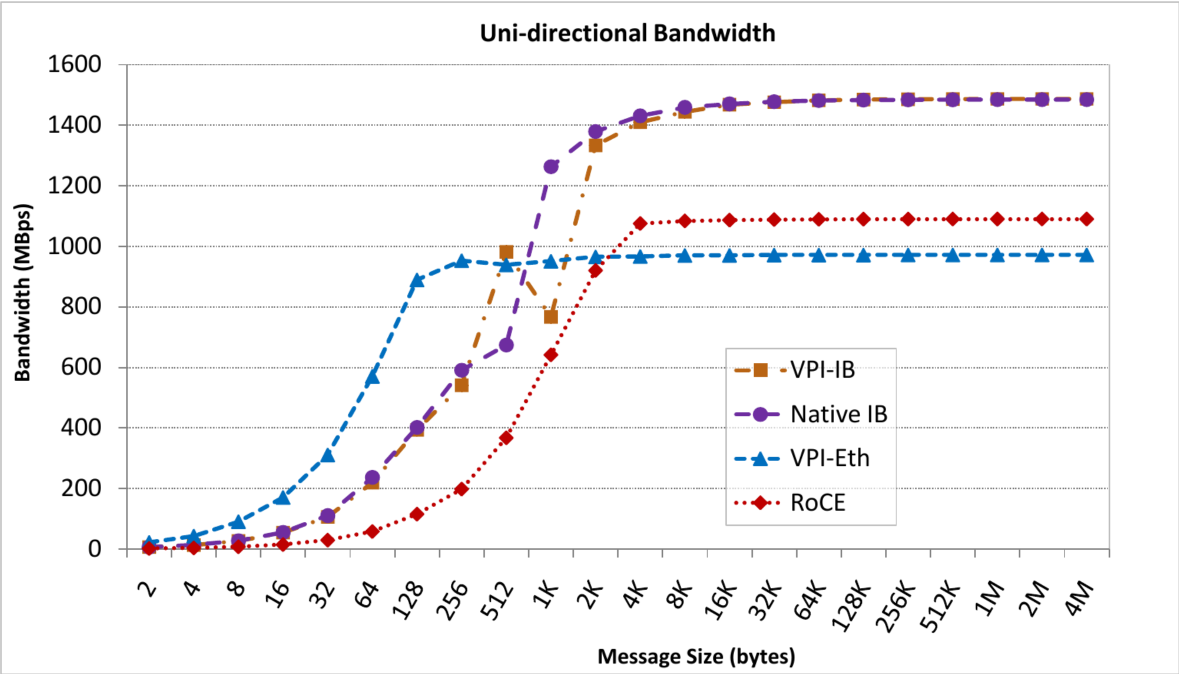
As you can see in the Unidirectional section of the report we do get 274 GBps out of the advertised 300GBps (~91%).

Bottom line - in this particular setup:

1. if you have huge payloads you will be able to use about 80% of the advertised 300GBps
2. if the payload of each communication is smallish it could be far far lower.

This graph is also helpful to demonstrate how the actual bandwidth changes with the size of the message:

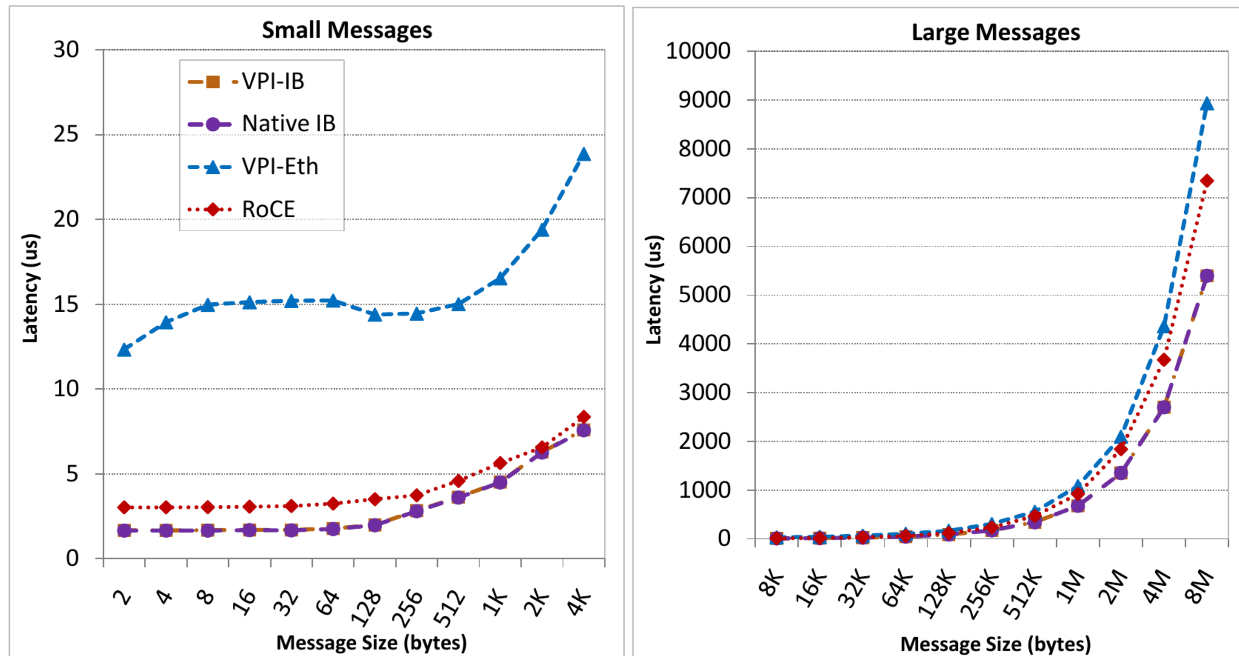
Low-level Uni-directional Bandwidth Measurements



ConnectX-DDR: 2.4 GHz Quad-core (Nehalem) Intel with IB and 10GE switches

(source)

Latency



(source)

XXX: integrate/expand

Proprietary network hardware and NCCL

Proprietary network hardware vendors like AWS (EFA) don't disclose their secrets and therefore the public libraries like nccl cannot support those out of the box. These vendors have to supply their own versions of the network collective libraries to be used by users of their hardware.

Originally proprietary hardware vendors used the trick of telling the users to use `LD_LIBRARY_PATH` and/or `LD_PRELOAD` to dynamically overload `libnccl.so` to get their custom version loaded into PyTorch or another framework. But recently NCCL developed a NCCL Net Plugin which should be used now instead. This feature was added in NCCL v2.12.

Now, when NCCL is initialized, it will look for a `libnccl-net.so` library and dynamically load it, then look for symbols inside the library. That's where proprietary hardware vendors should now put their custom APIs. This library, of course, should still be either in `LD_LIBRARY_PATH` or the `/etc/ld.so.conf` config.

For more information about dynamic library loading see this section.

Node Proximity

If you get 2 random nodes from the cloud they may not reside on the same subnet and there will be an additional latency incurred for all transmissions.

You want to make sure that the nodes used for a single training all reside on the same subnet/spine so they are all one hop away from each other.

When you plan to eventually have a large cluster but starting small make sure that your provider can expand the cluster while keeping all the nodes close to each other.

Here are the cloud-specific ways of accomplishing node proximity:

- Azure: availability set
- GCP: compact placement policies

Depending on the type of package you have or what type of machines you rent - you may or may not be able to use those.

Shared internode network

If you use a shared HPC environment, or even if you have your own cluster but sharing it with your colleagues expect the network bandwidth to be unreliable and fluctuate at different time of the day.

This situation unfortunately makes it extremely difficult to finetune the performance of your training setup. Since every time you run a test the TFLOPs will vary, so how do you do the optimization? Unfortunately I don't have a magic trick here. If you have a working solution please kindly share.

case study: we had this issue at JeanZay HPC when we were doing preliminary experiments before we started training BLOOM-176B. As that HPC has many users it was pretty much impossible to do speed optimizations, as even running the exact same setup again and again gave different throughput results. Luckily just before we launched BLOOM-176B training we were given an exclusive access to the new at that time A100 partition so we were the only users and we were able to greatly optimize the throughput.

Filesystems and IO

3 ML IO needs

There are 3 distinct IO needs in the ML workload:

1. You need to be able to feed the DataLoader fast - (super fast read, don't care about fast write) - requires sustainable load for hours and days
2. You need to be able to write checkpoints fast - (super fast write, fastish read as you will be resuming a few times) - requires burst writing - you want super fast to not block the training for long (unless you use some sort of cpu offloading to quickly unblock the training)
3. You need to be able to load and maintain your codebase - (medium speed for both reading and writing) - this also needs to be shared since you want all nodes to see the same codebase - as it happens only during the start or resume it'll happen infrequently

As you can see these 3 have very different requirements both on speed and sustainable load, and thus ideally you'd have 3 different filesystems, each optimized for the required use case.

If you have infinite funds, of course, get a single super-fast read, super-fast write, that can do that for days non-stop. But for most of us, this is not possible so getting 2 or 3 different types of partitions where you end up paying much less is a wiser choice.

Incoming suggestions from Ross Wightman to integrate:

- I'd try to separate volumes by workload, so keep the 'lots of small files', high churn like environments, code separate from bulk storage like datasets, checkpoints. Possibly even split those too since datasets are largely static and checkpoints are being rotated all the time
- When datasets are on network storage, just like bucket storage, they should consist of large files AND be read as large files (sequentially in large chunks, not mmapped!). Avoid seeking within datasets
- Setups like HF datasets can be deceiving, might look like one big file, but often being mmap'd and the IO read pattern is nuts, like 3-4x more iops than if you'd read them as individual files. Mmap loading can be turned off, but if that's the case, for a lot of datasets you move a problem into the DataLoader processes, requiring reading too much data into memory at once. Better awareness of tradeoffs for different use cases, and especially using Iterable streaming when appropriate.
- Note that once your datasets are optimally friendly for a large, distributed network filesystem, they can usually just be streamed from bucket storage in cloud systems that have that option. So better to move them off the network filesystem in that case.
- In a way, bucket storage like s3, via the interface limitations, enforces patterns that are reasonable for storage backends like this. It's ooh, it's mounted as a folder, I can do whatever I want (mmap files, write loads of little ones, delete them all, etc) that's the prob.
- One also cannot expect to treat a distributed filesystem like their local disk. If you separated volumes by workload you'd probably be able to utilize much higher % of the total storage. Don't mix high churn, small files with low churn large files.
- Also, note that once your datasets are optimally friendly for a large, distributed network filesystem, they can usually just be streamed from bucket storage in cloud systems that have that option. So better to move them off the network filesystem in that case.

Glossary

- NAS: Network Attached Storage
- SAN: Storage Area Network
- DAS: Direct-Attached storage
- NSD: Network Shared Disk
- OSS: Object storage server
- MDS: Metadata server
- MGS: Management server

Which file system to choose

Distributed Parallel File Systems are the fastest solutions

Distributed parallel file systems dramatically improve performance where hundreds to thousands of clients can access the shared storage simultaneously. They also help a lot with reducing hotspots (where some data pockets are accessed much more often than others).

The 2 excellent performing parallel file systems that I had experience with are:

- Lustre FS (Open Source) (Wiki)
- GPFS (IBM), recently renamed to IBM Storage Scale, and before that it was called IBM Spectrum Scale.

Both solutions have been around for 2+ decades. Both are POSIX-compliant. These are also not trivial to create - you have to setup a whole other cluster with multiple cpu-only VMs dedicated exclusively for those filesystems - only then you can mount those. As compared to weaker cloud-provided "built-in" solutions which take only a few screens of questions to answer in order to activate. And when creating the storage cluster there is a whole science to which VMs to choose for which functionality. For example, here is a Lustre guide on GCP.

case study: At JeanZay HPC (France) we were saving 2.3TB checkpoint in parallel on 384 processes in 40 secs! This is insanely fast - and it was GPFS over NVME drives.

NASA's cluster has a long long list of gotchas around using Lustre.

Some very useful pros of GFPS:

- If you have a lot of small files, you can easily run out of inodes (`df -i` to check). GFPS 5.x never runs out of inodes, it dynamically creates more as needed
- GPFS doesn't have the issue Lustre has where you can run out of disk space at 80% if one of the sub-disks got full and wasn't re-balanced in time - you can reliably use all 100% of the allocated storage.
- GPFS doesn't use a central metadata server (or a cluster of those) which often becomes a bottleneck when dealing with small files. Just like data, metatada is handled by each node in the storage cluster.
- GPFS comes with a native NSD client which is superior to the generic NFS client, but either can be used with it.

Other parallel file systems I don't yet have direct experience with:

- BeeGFS
- WekaIO
- DAOS (Distributed Asynchronous Object Storage) (Intel)
- NetApp

Most clouds provide at least one implementation of these, but not all. If your cloud provider doesn't provide at least one of these and they don't have a fast enough alternative to meet your needs you should reconsider.

OK'ish solutions

There are many OK'ish solutions offered by various cloud providers. Benchmark those seriously before you commit to any. Those are usually quite decent for handling large files and not so much for small files.

case study: As of this writing with GCP's Zonal FileStore over NFS solution `python -c "import torch"` takes 20 secs to execute, which is extremely slow! Once the files are cached it then takes ~2 secs. Installing a conda environment with a handful of prebuilt python packages can easily take 20-30 min! This solution we started with had been very painful and counter-productive to our work. This would impact anybody who has a lot of python packages and conda environments. But, of course, GCP provides much faster solutions as well.

Remote File System Clients

You will need to choose which client to use to connect the file system to your VM with.

The most common choice is: NFS - which has been around for 4 decades. It introduces an additional overhead and slows things down. So if there is a native client supported by your VM, you'd have an overall faster performance using it over NFS. For example, GPFS comes with an NSD client which is superior to NFS.

File Block size

If the file system you use uses a block size of 16mb, but the average size of your files is 16k, you will be using 1,000 times more disk space than the actual use. For example, you will see 100TB of disk space used when the actual disk space will be just 100MB.

footnote: On Linux the native file systems typically use a block size of 4k.

So often you might have 2 very different needs and require 2 different partitions optimized for different needs.

1. thousands to millions of tiny files - 4-8k block size
2. few large files - 2-16mb block size

case study: Python is so bad at having tens of thousand of tiny files that if you have many conda environments you are likely to run of inodes in some situations. At JeanZay HPC we had to ask for a special dedicated partition where we would install all conda environments because we kept running out of inodes on normal GPFS partitions. I think the problem is that those GPFS partitions were configured with 16MB block sizes, so this was not a suitable partition for 4KB-large files.

The good news is that modern solutions are starting to introduce a dynamic block size. For example, the most recent GPFS supports sub-blocks. So, for example, it's possible to configure GPFS with a block size of 2mb, with a sub-block of 8k, and then the tiny files get packed together as sub-blocks, thus not wasting too much disk space.

Cloud shared storage solutions

Here are shared file system storage solutions made available by various cloud providers:

- GCP
- Azure
- AWS

Local storage beats cloud storage

While cloud storage is cheaper the whole idea of fetching and processing your training data stream dynamically at training time is very problematic with a huge number of issues around it.

Same goes for dynamic offloading of checkpoints to the cloud.

It's so much better to have enough disk space locally for data loading.

For checkpointing there should be enough local disk space for saving a checkpoint in a fast and reliable way and then having a crontab job or a slurm job to offload it to the cloud. Always keep the last few checkpoints locally for a quick

resume, should your job crash, as it'd be very expensive to wait to fetch the checkpoint from the cloud for a resume.

case study: we didn't have a choice and had to use cloud storage for dataloading during IDEFICS-80B training as we had barely any local storage and since it was multimodal data it was many TBs of data. We spent many weeks trying to make this solution robust and it sucked at the end. The biggest issue was that it was very difficult at the time to keep track of RNG state for the DataSampler because the solution we used, well, didn't bother to take care of it. So a lot of data that took a lot of time to create was wasted (not used) and a lot of data was repeated, so we didn't have a single epoch of unique data.

Beware that you're often being sold only 80% of the storage you pay for

There is a subtle problem with distributed shared storage used on compute nodes. Since most physical disks used to build the large file systems are only 0.3-2TB large, any of these physical disks can get full before the combined storage gets full. And thus they require constant rebalancing so that there will be no situation where one disk is 99% full and others are only 50% full. Since rebalancing is a costly operation, like most programming languages' garbage collection, it happens infrequently. And so if you run `df` and it reports 90% full, it's very likely that any of the programs can fail at any given time.

From talking to IO engineers, the accepted reality (that for some reason is not being communicated to customers) is that only about 80% of distributed large storage is reliable.

Which means that if you want to have 100TB of reliable cloud storage you actually need to buy 125TB of storage, since 80% of that will be 100TB. So you need to plan to pay 25% more than what you provisioned for your actual needs. I'm not sure why the customer should pay for the technology deficiency but that's how it is.

For example, GCP states that only 89% can be used reliably, albeit more than once the storage failed already at 83% for me there. Kudos to Google to even disclosing this as a known issue, albeit not at the point of where a person buys the storage. As in - we recommend you buy 12% more storage than you actually plan to use, since we can only reliably deliver 89% of it.

I also talked to Sycomp engineers who provide managed IBM Storage Scale (GPFS) solutions, and according to them GPFS doesn't have this issue and the whole 100% can be reliably used.

Also on some setups if you do backups via the cloud provider API (not directly on the filesystem), they might end up using the same partition, and, of course, consume the disk space, but when you run `df` it will not show the real disk usage - it may show usage not including the backups. So if your backups consume 50% of the partition.

Whatever storage solution you pick, ask the provider how much of the storage can be reliably used, so that there will be no surprises later.

Don't forget the checksums

When you sync data to and from the cloud make sure to research whether the tool you use checks the checksums, otherwise you may end up with corrupt during transmission data. Some tools do it automatically, others you have to enable this feature (since it usually comes at additional compute cost and transmission slowdown). Better slow, but safe.

These are typically MD5 and SHA256 checksums. Usually MD5 is sufficient if your environment is safe, but if you want the additional security do SHA256 checksums.

Concepts

Here are a few key storage-related concepts that you likely need to be familiar with:

Queue Depth

Queue depth (or **IO depth**) is the number of IO requests that can be queued at one time on a storage device controller. If more IO requests than the controller can queue are being sent the OS will usually put those into its own queue.

On Linux the local block devices' queue depth is usually pre-configured by the kernel. For example, if you want to check the max queue depth set for `/dev/sda` you can `cat /sys/block/sda/queue/nr_requests`. To see the current queue depth of a local device run `iostat -x` and watch for `aqu-sz` column. (`apt install sysstat` to get `iostat`.)

Typically the more IO requests get buffered the bigger the latency will be, and the better the throughput will be. This is because if a request can't be acted upon immediately it'll prolong the response time as it has to wait before being served. But having multiple requests awaiting to be served in a device's queue would typically speed up the total throughput as there is less waiting time between issuing individual requests.

Direct vs Buffered IO

Direct IO refers to IO that bypasses the operating system's caching buffers. This corresponds to `O_DIRECT` flag in `open(2)` system call.

The opposite is the **buffered** IO, which is usually the default way most applications do IO since caching typically makes things faster.

When we run an IO benchmark it's critical to turn the caching/buffering off, because otherwise the benchmark's results will most likely be invalid. You normally won't be reading or writing the same file hundreds of times in a row. Hence most likely you'd want to turn the direct mode on in the benchmark's flags if it provides such.

In certain situation opening files with `O_DIRECT` may actually help to overcome delays. For example, if the training program logs to a log file (especially on a slow shared file system), you might not be able to see the logs for many seconds if both the application and the file system buffering are in the way. Opening the log file with `O_DIRECT` by the writer typically helps to get the reader see the logged lines much sooner.

Synchronous vs asynchronous IO

In synchronous IO the client submits an IO request and wait for it to be finished before submitting the next IO request to the same target device.

In asynchronous IO the client may submit multiple IO requests one after another without waiting for any to finish first. This requires that the target device can queue up multiple IO requests.

Sequential vs Random access IO

Sequential access IO is when you read blocks of data one by one sequentially (think a movie). Here are some examples:

- reading or writing a model's checkpoint file all at once
- loading a python program
- installing a package

Random access IO is when you're accessing part of a file at random. Here are some examples:

- database querying
- reading samples from a pre-processed dataset in a random fashion
- moving around a file using `seek`

Benchmarks

Time is money both in terms of a developer's time and model's training time, so it's crucial that storage IO isn't a bottleneck in your human and compute workflows.

In the following sections we will discuss various approaches to figuring out whether the proposed storage solution satisfies your work needs.

Metrics

The three main storage IO metrics one typically cares for are:

1. Network Throughput or Bandwidth (bytes per second - can be MBps, GBps, etc.)
2. IOPS (Input/output operations per second)
3. Latency (msecs or usecs)

- *Network Throughput* refers to the rate of message delivery over a network.
- *IOPS* measures how many input and/or output operations a given storage device can perform per second. Whether it's just read or write will depend on the use case - it can be just read or just write or both.

Latency: is the delay between the moment the instruction to transfer data is issued and the data starting to arrive.

It might be easier to understand using a commuter experience. If it takes you 1h to commute to work, then your commute latency is 2h per day - as you have to travel to work and back. Regardless if you spent at work 8h or you stayed for just an hour, it'll still take you approximately the same amount of time to return home (minus the fluctuation in traffic jams).

So if you have a local NVME drive your read or write latency will be much shorter as compared to reading or writing to a storage device that is located on another continent.

fio

fio - Flexible I/O tester is a commonly used IO benchmarking tool, which is relatively easy to operate. It has many options which allow you to emulate pretty much any type of a load and it provides a very detailed performance report.

First install `fio` with `apt install fio` or however your package manager does it.

Here is an example of a read benchmark:

```
base_path=/path/to/partition/
fio --ioengine=libaio --filesize=16k --ramp_time=2s --time_based --runtime=3m --numjobs=16 \
--direct=1 --verify=0 --randrepeat=0 --group_reporting --unlink=1 --directory=$base_path  \
--name=read-test --blocksize=4k --iodepth=64 --readwrite=read
```

Here 16 concurrent read threads will run for 3 minutes. The benchmark uses a block size of 4k (typical for most OSes) with the file size of 16k (a common size of most Python files) in a sequential reading style using non-buffered IO. So this particular set of flags will create a good benchmark to show how fast you can import Python modules on 16 concurrent processes.

case study: on one NFS setup we had `python -c "import torch"` taking 20 seconds the first time it was run, which is about 20x slower than the same test on a normal NVME drive. Granted once the files were cached the loading was much faster but it made for a very painful development process since everything was slow.

good read: Fio Output Explained - it's an oldie but is still a goodie - if you have a more up-to-date write up please send me a link or a PR.

Important: if you don't use the `--unlink=1` flag make sure to delete `fio`'s work files between different benchmarks - not doing so can lead to seriously wrong reports as `fio` will reuse files it prepared for a different benchmark which must not be re-used if the benchmark parameters have changed. Apparently this reuse is an `fio` feature, but to me it's a bug since I didn't know this nuance and got a whole lot of invalid reports because of it and it took awhile to realize they were wrong.

Going back to the benchmark - the parameters will need to change to fit the type of the IO operation you care to be fast - is it doing a lot of pip installs or writing a checkpoint on 512 processes, or doing a random read from a parquet file - each benchmark will have to be adapted to measure the right thing.

At the beginning I was manually fishing out the bits I was after, so I automated it resulting in fio-scan benchmark that will run a pair of read/write benchmarks on 16KB, 1MB and 1GB file sizes each using a fixed 4k block size (6 benchmarks in total). It uses a helper fio-json-extract.py to parse the log files and pull out the average latency, bandwidth and iops and report them in a nicely formatted markdown table.

Here is how to run it:

```
git clone https://github.com/stas00/ml-engineering/
cd ml-engineering
cd storage

path_to_test=/path/to/partition/to/test
./fio-scan $path_to_test
```

Adapt `path_to_test` to point to the partition path you want to benchmark.

note: the log parser uses python3. if `fio-scan` fails it's most likely because you run it on a system with python2 installed by default. It expects `python --version` to be some python 3.x version. You can edit `fio-scan` to point to the right `python`.

Here is an example of this IO scan on my Samsung SSD 980 PRO 2TB NVME drive (summary):

- filesize=16k read

lat msec	bw MBps	IOPS	jobs
4.0	1006.3	257614	16

- filesize=16k write

lat msec	bw MBps	IOPS	jobs
3.2	1239.1	317200	16

- filesize=1m read

lat msec	bw MBps	IOPS	jobs
1.7	2400.1	614419	16

- filesize=1m write

lat msec	bw MBps	IOPS	jobs
2.1	1940.5	496765	16

- filesize=1g read

lat msec	bw MBps	IOPS	jobs
1.4	2762.0	707062	16

- filesize=1g write

lat msec	bw MBps	IOPS	jobs
2.1	1943.9	497638	16

As you can see as of this writing this is a pretty fast NVMe drive if you want to use it as a base-line against, say, a network shared file system.

Poor man's storage IO benchmark

Besides properly designed performance benchmarks which give you some numbers that you may or may not be able to appreciate there is a perception benchmark, and that is how does a certain functionality or a service feel. For example, when going to a website, does it feel like it's taking too long to load a webpage? or when going to a video service, does it take too long for the video to start playing and does it stop every few seconds to buffer the stream?

So with file system the questions are very simple - does it feel that it takes too long to install or launch a program? Since a lot of us live in the Python world, python is known to have thousands of tiny files which are usually installed into a virtual environment, with conda being the choice of many as of this writing.

In one of the environments we have noticed that our developers' productivity was really bad on a shared filesystem because it was taking up to 30min to install a conda environment with various packages needed for using a certain ML-training framework, and we also noticed that `python -c "import torch'` could take more than 20 seconds. This is about 5-10x slower than a fast local NVME-based filesystem would deliver. Obviously, this is bad. So I devised a perception test using `time` to measure the common activities. That way we could quickly tell if the proposed shared file system solution that we contemplated to switch to were significantly better. We didn't want a solution that was 2x faster, we wanted a solution that was 10x better, because having an expensive developer wait for proverbial paint to dry is not a good thing for a business.

So here is the poor man's benchmark that we used, so this is just an example. Surely if you think about the workflow of your developers you would quickly identify where things are slow and devise yours best fitting your needs.

note: To have a baseline to compare to do these timing tests on a recently manufactured local NVME. This way you know what the ceiling is, but with beware that many shared file systems won't be able to match that.

Step 1. Install conda onto the shared file system you want to test if it's not there already.

```
export target_partition_path=/mnt/weka  # edit me!!!
mkdir -p $target_partition_path/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O $target_partition_path/
miniconda3/miniconda.sh
bash $target_partition_path/miniconda3/miniconda.sh -b -u -p $target_partition_path/miniconda3
rm -rf $target_partition_path/miniconda3/miniconda.sh
$target_partition_path/miniconda3/bin/conda init bash
bash
```

notes:

- adapt `target_partition_path` and the miniconda download link if you aren't on the x86 platform.
- at the end we launch a new `bash` shell for conda setup to take an effect, you might need to tweak things further if you're not a `bash` user - I trust you will know what to do.

Step 2. Measure conda install time (write test)

Time the creation of a new conda environment:

```
time conda create -y -n install-test python=3.9
```

```
real    0m29.657s
user    0m9.141s
sys     0m2.861s
```

Time the installation of some heavy pip packages:

```
conda deactivate
conda activate install-test
time pip install torch torchvision torchaudio
```

```
real    2m10.355s
user    0m50.547s
sys     0m12.144s
```

Please note that this test is somewhat skewed since it also includes the packages download in it and depending on your incoming network speed it could be super fast or super slow and could impact the outcome. But once the downloaded packages are cached, in the case of conda they are also untarred, so if you try to install the packages the 2nd time the benchmark will no longer be fair as on a slow shared file system the untarring could be very slow and we want to catch that.

I don't worry about it because usually when the file system is very slow usually you can tell it's very slow even if the downloads are slow, you just watch the progress and you can just tell.

If you do want to make this benchmark precise, you probably could keep the pre-downloaded conda packages and just deleting their untar'ed dirs:

```
find $target_partition_path/miniconda3/pkgs -mindepth 1 -type d -exec rm -rf {} +
```

in the case of `pip` it doesn't untar anything, but just caches the wheels it downloaded, so the `time pip install` benchmark can definitely be more precise if you run it the 2nd time (the first time it's downloaded, cached and installed, the second time it's installed from cache. So you could do:

```
conda create -y -n install-test python=3.9
conda activate install-test
pip install torch torchvision torchaudio
conda create -y -n install-test2 python=3.9
```

```
conda activate install-test2
time pip install torch torchvision torchaudio
```

As you can see here we time only the 2nd time we install the pip packages.

Step 3. Measure loading time after flushing the memory and file system caches (read test)

```
sudo sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
time python -c "import torch"
```

As you can see before we do the measurement we have to tell the OS to flush its memory and file system caches.

If you don't have `sudo` access you can skip the command involving `sudo`, also sometimes the system is setup to work w/o `sudo`. If you can't run the syncing and flushing of the file system caches you will just get incorrect results as the benchmark will be measuring the time to load already cached file system objects. To overcome this either ask your sysadmin to do it for you or simply come back in the morning while hopefully your file system caches other things and evicts the python packages, and then repeat the python one liner then with the hope those files are no longer in the cache.

Here is how to see the caching effect:

```
$ time python -c "import torch"

real    0m5.404s
user    0m1.761s
sys     0m0.751s

$ time python -c "import torch"

real    0m1.977s
user    0m1.623s
sys     0m0.519s

$ sudo sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ time python -c "import torch"

real    0m5.698s
user    0m1.712s
sys     0m0.734s
```

You can see that the first time it wasn't cached and took ~3x longer, then when I run it the second time. And then I told the system to flush memory and file system caches and you can see it was 3x longer again.

I think it might be a good idea to do the memory and file system caching in the write tests again, since even there caching will make the benchmark appear faster than what it would be like in the real world where a new package is installed for the first time.

other tools

-

- HPC IO Benchmark Repository (`mdtest` has been merged into `ior` in 2017)
- DLIO

XXX: expand on how these are used when I get a chance to try those

Published benchmarks

Here are some published IO benchmarks:

- MLPerf via MLCommons publishes various hardware benchmarks that measure training, inference, storage and other tasks' performance. For example, here is the most recent as of this writing storage v0.5 results. Though I find the results are very difficult to make sense of - too many columns and no control whatsoever by the user, and each test uses different parameters - so how do you compare things.

Then various benchmarks that you can run yourself:

Contributors

Ross Wightman

fio benchmark results for hope on 2023-12-20-14:37:02

partition /mnt/nvme0/fio/fio-test

- filesize=16k read

lat msec	bw MBps	IOPS	jobs
4.0	1006.3	257614	16

- filesize=16k write

lat msec	bw MBps	IOPS	jobs
3.2	1239.1	317200	16

- filesize=1m read

lat msec	bw MBps	IOPS	jobs
1.7	2400.1	614419	16

- filesize=1m write

lat msec	bw MBps	IOPS	jobs
2.1	1940.5	496765	16

- filesize=1g read

lat msec	bw MBps	IOPS	jobs
1.4	2762.0	707062	16

- filesize=1g write

lat msec	bw MBps	IOPS	jobs
2.1	1943.9	497638	16

CPU

XXX: This chapter needs a lot more work

How many cpu cores do you need

Per 1 gpu you need:

1. 1 cpu core per process that is tied to the gpu
2. 1 cpu core for each DataLoader worker process - and you need 2-4 workers.

2 workers is usually plenty for NLP, especially if the data is preprocessed

If you need to do dynamic transforms, which is often the case with computer vision models, you may need 3-4 and sometimes more workers.

The goal is to be able to pull from the DataLoader instantly, and not block the GPU's compute, which means that you need to pre-process a bunch of samples for the next iteration, while the current iteration is running. In other words your next batch needs to take no longer than a single iteration GPU compute of the batch of the same size.

Besides preprocessing if you're pulling dynamically from the cloud instead of local storage you also need to make sure that the data is pre-fetched fast enough to feed the workers that feed the gpu furnace.

Multiply that by the number of GPUs, add a few cores for the Operation system (let's say 4).

If the node has 8 gpus, and you have n_workers, then you need `8*(num_workers+1)+4`. If you're doing NLP, it'd be usually about 2 workers per gpu, so `8*(2+1)+4` => 28 cpu cores. If you do CV training, and, say, you need 4 workers per gpu, then it'd be `8(4+1)+4` => 44 cpu cores.

What happens if you have more very active processes than the total number of cpu cores? Some processes will get preempted (put in the queue for when cpu cores become available) and you absolutely want to avoid any context switching.

But modern cloud offerings typically have 48+ cpu-cores so usually there is no problem to have enough cores to go around.

CPU offload

Some frameworks, like [Deepspeed](https://www.deepspeed.ai/tutorials/zero-offload/) can offload some compute work to CPU without creating an bottleneck. In which case you'd want additional cpu-cores.

Hyperthreads

Doubles the cpu cores number

XXX:

CPU memory

This is a tiny chapter, since usually there are very few nuances one needs to know about CPU memory - which is a good thing!

Most of the ML workload compute happens on GPUs, but typically there should be at least as much CPU memory on each node as there is on the GPUs. So, for example, if you're on a H100 node with 8x 80GB GPUs, you have 640GB of GPU memory. Thus you want at least as much of CPU memory. But most recent high end cloud packages usually come with 1-2TBs of CPU memory.

What CPU memory is needed for in ML workloads

- Loading the model weights, unless they are loaded directly onto the GPUs - this is usually a transitory memory usage that goes back to zero once the model has been moved to GPUs.
- Saving the model weights. In some situations each GPU writes its own checkpoint directly to the disk, in other cases the model is recomposed on the CPU before it's written to disk - this too is a transitory memory usage.
- Possible parameter and optimizer state offloading when using frameworks like [Deepspeed](). In which case quite a lot of CPU memory might be needed.
- Activations calculated in the `forward` pass, and which need to be available for the `backward` path can also be offloaded to CPU, rather than discarded and then recomputed during the backward pass to save the unnecessary overhead
- `DataLoader` is usually one of the main users of CPU memory and at times it may consume very large amounts of memory. Typically there are at least 2x 8 DL workers running on each node, so you need enough memory to support at least 16 processes each holding some data. For example, in the case of streaming data from the cloud, if the data shards are large, these processes could easily eat up hundreds of GBs of CPU memory.
- The software itself and its dependent libraries uses a bit of CPU memory, but this amount is usually negligible.

Things to know

- If the `DataLoader` uses HF `datasets` in `mmap` mode the Resident memory usage may appear to be using a huge amount of CPU memory as it'll try to map out the whole datasets to the memory. Except this is misleading, since if the memory is needed elsewhere the OS will page out any unneeded mmap'ed pages back to the system. You can read more about it [here](). This awareness, of course, applies to any dataset using `mmap`, I was using HF `datasets` as an example since it's very widely used.

Performance and Acceleration

- Tuning ML software for best performance - tweaking software for the best performance.
- Tuning ML hardware for best performance - choosing and configuring machine learning hardware for best performance.

Choosing and Configuring Machine Learning Hardware for Best Performance

Notes on getting the best performance from the perspective of what hardware is being used. And how to properly set up and configure it for the peak performance.

GPUs

Power and Cooling

If you bought an expensive high end GPU make sure you give it the correct power and sufficient cooling.

Power

Some high end consumer GPU cards have 2 and sometimes 3 PCI-E 8-Pin power sockets. Make sure you have as many independent 12V PCI-E 8-Pin cables plugged into the card as there are sockets. Do not use the 2 splits at one end of the same cable (also known as pigtail cable). That is if you have 2 sockets on the GPU, you want 2 PCI-E 8-Pin cables going from your PSU to the card and not one that has 2 PCI-E 8-Pin connectors at the end! You won't get the full performance out of your card otherwise.

Each PCI-E 8-Pin power cable needs to be plugged into a 12V rail on the PSU side and can supply up to 150W of power.

Some other cards may use a PCI-E 12-Pin connectors, and these can deliver up to 500-600W of power.

Low end cards may use 6-Pin connectors, which supply up to 75W of power.

Additionally you want the high-end PSU that has stable voltage. Some lower quality ones may not give the card the stable voltage it needs to function at its peak.

And of course the PSU needs to have enough unused Watts to power the card.

Cooling

When a GPU gets overheated it will start throttling down and will not deliver full performance and it can even shutdown if it gets too hot.

It's hard to tell the exact best temperature to strive for when a GPU is heavily loaded, but probably anything under +80C is good, but lower is better - perhaps 70-75C is an excellent range to be in. The throttling down is likely to start at around 84-90C. But other than throttling performance a prolonged very high temperature is likely to reduce the lifespan of a GPU.

Intra-connects and Inter-connects

Multi-GPU Connectivity

If you use multiple GPUs the way cards are inter-connected can have a huge impact on the total training time. If the GPUs are on the same physical node, you can run:

```
nvidia-smi topo -m
```

and it will tell you how the GPUs are inter-connected. On a machine with dual-GPU and which are connected with NVLink,

you will most likely see something like:

```
        GPU0    GPU1    CPU Affinity    NUMA Affinity
GPU0     X      NV2     0-23            N/A
GPU1    NV2      X      0-23            N/A
```

on a different machine w/o NVLink we may see:

```
        GPU0    GPU1    CPU Affinity    NUMA Affinity
GPU0     X      PHB     0-11            N/A
GPU1    PHB      X      0-11            N/A
```

The report includes this legend:

```
  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA
node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

So the first report `NV2` tells us the GPUs are interconnected with 2 NVLinks, and the second report `PHB` we have a typical consumer-level PCIe+Bridge setup.

Check what type of connectivity you have on your setup. Some of these will make the communication between cards faster (e.g. NVLink), others slower (e.g. PHB).

Depending on the type of scalability solution used, the connectivity speed could have a major or a minor impact. If the GPUs need to sync rarely, as in DDP, the impact of a slower connection will be less significant. If the GPUs need to send messages to each other often, as in ZeRO-DP, then faster connectivity becomes super important to achieve faster training.

NVlink

[NVLink](https://en.wikipedia.org/wiki/NVLink) is a wire-based serial multi-lane near-range communications link developed by Nvidia.

Each new generation provides a faster bandwidth, e.g. here is a quote from [Nvidia Ampere GA102 GPU Architecture]():

> Third-Generation NVLink® GA102 GPUs utilize NVIDIA's third-generation NVLink interface, which includes four x4 links, with each link providing 14.0625 GB/sec bandwidth in each direction between two GPUs. Four links provide 56.25 GB/sec bandwidth in each direction, and 112.5 GB/sec total bandwidth between two GPUs. Two RTX 3090 GPUs can be connected together for SLI using NVLink. (Note that 3-Way and 4-Way SLI configurations are not supported.)

So the higher `X` you get in the report of `NVX` in the output of `nvidia-smi topo -m` the better. The generation will depend on your GPU architecture.

Let's compare the execution of a gpt2 language model training over a small sample of wikitext.

The results are:

NVlink	Time
Y	101s
N	131s

You can see that NVLink completes the training ~23% faster. In the second benchmark we use `NCCL_P2P_DISABLE=1` to tell the GPUs not to use NVLink.

Here is the full benchmark code and outputs:

```
# DDP w/ NVLink

rm -r /tmp/test-clm; CUDA_VISIBLE_DEVICES=0,1 python -m torch.distributed.launch \
--nproc_per_node 2 examples/pytorch/language-modeling/run_clm.py --model_name_or_path gpt2 \
--dataset_name wikitext --dataset_config_name wikitext-2-raw-v1 --do_train \
--output_dir /tmp/test-clm --per_device_train_batch_size 4 --max_steps 200

{'train_runtime': 101.9003, 'train_samples_per_second': 1.963, 'epoch': 0.69}

# DDP w/o NVLink

rm -r /tmp/test-clm; CUDA_VISIBLE_DEVICES=0,1 NCCL_P2P_DISABLE=1 python -m torch.distributed.launch \
--nproc_per_node 2 examples/pytorch/language-modeling/run_clm.py --model_name_or_path gpt2 \
--dataset_name wikitext --dataset_config_name wikitext-2-raw-v1 --do_train
--output_dir /tmp/test-clm --per_device_train_batch_size 4 --max_steps 200

{'train_runtime': 131.4367, 'train_samples_per_second': 1.522, 'epoch': 0.69}
```

Hardware: 2x TITAN RTX 24GB each + NVlink with 2 NVLinks (`NV2` in `nvidia-smi topo -m`) Software: `pytorch-1.8-to-be` + `cuda-11.0` / `transformers==4.3.0.dev0`

NVSwitch

InfiniBand (IB)

Infiniband

Intel Omni-Path (OPA)

EFA

AWS-based hardware

Performance measurements

- all_reduce_bench.py

- `all_gather_perf` from nccl-tests

Software Tune Up For The Best Performance

The faster you can make your model to train the sooner the model will finish training, which is important not only to being first to publish something, but also potentially saving a lot of money.

In general maximizing throughput is all about running many experiments and measuring the outcome and choosing the one that is superior.

In certain situations your modeling team may ask you to choose some hyper parameters that will be detrimental to throughput but overall beneficial for the overall model's success.

Crucial reproducibility requirements

The most important requirements for a series of successful experiments is to be able to reproduce the experiment environment again and again while changing only one or a few setup variables.

Therefore when you try to figure out whether some change will improve performance or make it worse, you must figure out how to keep things stable.

For example, you need to find a way to prevent the network usage from fluctuations. When we were doing performance optimizations for 108B pre-BLOOM experiments it was close to impossible to perform, since we were on a shared internode network and the exact same setup would yield different throughput depending on how many other users used the network. It was not working. During BLOOM-176B we were given a dedicated SLURM partition with an isolated network where the only traffic was ours. Doing the performance optimization in such environment was just perfect.

Network throughput

It's critical to understand your particular model size and framework requirements with regard to network bandwidth, throughput and latency. If you underpay for network you will end up having idle gpus and thus you wasted money and time. If you overpay for very fast network, but your gpus are slow, then again you wasted money and time.

If your network is very slow, your training is likely to be network-bound and many improvements in the training setup will not help with the improving performance.

Here is a simple all-reduce benchmark that you can use to quickly measure the throughput of your internode network:

all_reduce_bench.py

Usually benchmarking at least 4 nodes is recommended, but, of course, if you already have access to all the nodes you will be using during the training, benchmark using all of the nodes.

To run it on 4 nodes:

```
GPUS_PER_NODE=8
NNODES=4
MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=6000
python -u -m torch.distributed.run \
    --nproc_per_node $GPUS_PER_NODE \
    --nnodes $NNODES \
    --rdzv_endpoint $MASTER_ADDR:$MASTER_PORT \
    --rdzv_backend c10d \
```

```
    --max_restarts 0 \
    --role `hostname -s`: \
    --tee 3 \
    all_reduce_bench.py
```

Notes:

- adapt `MASTER_ADDR` to rank 0 hostname if it's not a SLURM environment where it's derived automatically.

Here is how to run launch it in a SLURM env with 4 nodes:

```
salloc --partition=mypartition --nodes=4 --ntasks-per-node=1 --cpus-per-task=48 --gres=gpu:8
--time=1:00:00 bash
srun --gres=gpu:8 --nodes=4 --tasks-per-node=1 python -u -m torch.distributed.run --nproc_per_node=8
--nnodes 4 --rdzv_endpoint $(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1):6000 --rdzv_backend
c10d all_reduce_bench.py
```

Notes:

- You are likely to need to adapt `--cpus-per-task` and `--partition` arguments there.
- You do `salloc` once and then can repeat `srun` multiple times on the same allocation.

You may get results anywhere between 5Gbps and 1600Gbps (as of this writing). The minimal speed to prevent being network bound will depend on your particular training framework, but typically you'd want at least 400Gbps or higher. Though we trained BLOOM on 50Gbps.

Frameworks that shard weights and optim stages like Deepspeed w/ ZeRO Stage-3 do a lot more traffic than frameworks like Megatron-Deepspeed which do tensor and pipeline parallelism in addition to data parallelism. The latter ones only send activations across and thus don't need as much bandwidth. But they are much more complicated to set up and run.

Of course, an efficient framework will overlap communications and compute, so that while one stage is fetching data, the other stage in parallel runs computations. So as long as the communication overhead is smaller than compute the network requirements are satisfied and don't have to be super fantastic.

To get reasonable GPU throughput when training at scale (64+GPUs) with DeepSpeed ZeRO Stage 3 with V100s

1. 100Gbps is not enough
2. 200-400 Gbps is ok
3. 800-1000 Gbps is ideal

full details

Of course, the requirements are higher for A100 gpu nodes and even higher for H100s (but no such benchmark information has been shared yet).

MACs vs FLOP vs FLOPS vs FLOP/s

This section is here to try to disambiguate the common performance metric definitions and their relationship to each other.

MAC vs FLOP:

- 1 FLOP (FLoating point OPeration) can be one of addition, subtraction, multiplication, or division operation.
- 1 MAC (Multiply-ACCumulate) operation is a multiplication followed by an addition, that is: `a * b + c`

Thus 1 MAC = 2 FLOPs. It's also quite common for modern hardware to perform 1 MAC in a single clock cycle.

Please note that to calculate the number of MACs in relationship to FLOPs the reverse logic applies, that is MACs = 0.5 FLOPs - it's somewhat confusing since we have just said that 1 MAC = 2 FLOPs, but it checks out - observe: 100 FLOPs = 50 MACs - because there are 2 FLOPs in each MAC.

Moreover, while 1 MAC = 2 FLOPs, the reverse isn't necessarily true. That is 2 FLOPs isn't necessarily equal to 1 MAC. For example, if you did `.5*.6` 100 times it'd be 100 FLOPs, which here would equal to 100 MACs, because here only the multiply part of the MAC is executed.

FLOP vs FLOPS vs FLOP/s

- 1 FLOP (FLoating point OPeration) is any floating point addition, subtraction, multiplication, or division operation.
- 1 FLOPS (FLoating point OPeration per Second) is how many floating point operations were performed in 1 second - see FLOPS

Further you will find the following abbreviations: GFLOPS = Giga FLOPS, TFLOPS = Tera FLOPS, etc., since it's much easier to quickly grasp 150TFLOPS rather than 150000000000000FLOPS.

There is an ambiguity when FLOPS is used in writing - sometimes people use it to indicate the total quantity of operations, at other times it refers to operations per second. The latter is the most common usage and that is the definition used in this book.

In scientific writing FLOP/s is often used to clearly tell the reader that it's operations per second. Though this particular approach is hard to convert to a variable name since it still becomes `flops` when illegal characters are removed.

In some places you might also see FLOPs, which again could mean either, since it's too easy to flip lower and upper case `s`.

If the definition is ambiguous try to search for context which should help to derive what is meant:

- If it's a math equation and there is a division by time you know it's operations per second.
- If speed or performance is being discussed it usually refers to operations per second.
- If it talks about the amount of compute required to do something it refers to the total amount of operations.

TFLOPS as a performance metric

Before you start optimizing the performance of your training setup you need a metric that you can use to see whether the throughput is improving or not. You can measure seconds per iteration, or iterations per second, or some other such timing, but there is a more useful metric that measures TFLOPS.

Measuring TFLOPS is superior because without it you don't know whether you are close to the best performance that can be achieved or not. This measurement gives you an indication of how far you're from the peak performance reported by the hardware manufacturer.

In this section I will use BLOOM's training for the exemplification. We used 80GB A100 NVIDIA GPUs and we trained in mixed bf16 regime. So let's look at the A100 spec which tells us:

```
BFLOAT16 Tensor Core        312 TFLOPS
```

Therefore we now know that if we were to only run `matmul` on huge bf16 matrices of very specific dimensions without copying to and from the device we should get around 312 TFLOPS max.

Practically though, due to disk IO, communications and copying data from the GPU's memory to its computing unit overhead and because we can't do everything in bf16 and at times we have to do math in fp32 (or tf32) we can really expect much less than that. The realistic value will vary from accelerator to accelerator, but for A100 in 2022 getting above 50% (155 TFLOPS) was an amazing sustainable throughput for a complex 384 GPUs training setup.

footnote: in 2023 the invention of flash attention and other techniques have pushed the bar to more than 50%.

When we first started tuning things up we were at <100 TFLOPS and a few weeks later when we launched the training we managed to get 150 TFLOPS.

The important thing to notice here is that we knew that we can't push it further by much and we knew that there was no more point to try and optimize it even more.

So a general rule of thumb for when you prepare for a massive model training - ask around what's the top TFLOPS one can expect to get with a given accelerator on a multi-node setup with the specified precision - and optimize until you get close to that. Once you did stop optimizing and start training.

footnote: For 80GB A100s in 2022 that was 155, in 2023 it has been pushed to about 180 TFLOPS.

footnote: When calculating TFLOPS it's important to remember that the math is different if [Gradient checkpointing]() are enabled, since when it's activated more compute is used and it needs to be taken into an account. Usually the cost is of an additional forward path, but recently better methods have been found that saves some of that recomputation.

For decoder transformer models the following is an estimation formula which slightly under-reports the real TFLOPS:

TFLOPS: `model_size_in_B * 4 * 2 * seqlen * global_batch_size / (time_in_sec_per_interation * total_gpus * 1e3)`

The factor of 4 is used with activation/gradient checkpointing, otherwise it will be 3. For 100B+ models, activation checkpointing will almost always be on.

So the `3*2` is often called "model FLOPs" and `4*2` - "hardware FLOPs", correlating to MFU and HFU (model and hardware FLOPS per second divided by the accelerator's theoretical peak FLOPS)

```
perl -le '$ng=64; $ms=52; $gbs=1024; $sp=127; $seqlen=2048; print $ms*4*2*$seqlen*$gbs / ( $sp * $ng *
1e3)'
```

(ng = total gpus, ms = model size in B, gbs = global batch size, sp = throughput in seconds)

Here is the same formula using `bash` env vars and which breaks down GBS into `MBS*DP*GAS` (GAS in this case corresponded to `pp_chunks` which was the number of chunks in the pipeline, but normally GAS just stands for Gradient Accumulation Steps):

```
echo "($MSIZE*4*2*SEQLEN*$MICRO_BATCH_SIZE*$DP_SIZE*$GAS)/($THROUGHPUT*$NNODES*4*1000)" | bc -l
```

The exact formula is in Equation 3 of Section 5.1 of the Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM paper. You can see the code here.

footnote: For Inference only it'd be: `24Bsh^2 + 4Bs^2h` floating point operations per layer.

How To Improve Speed and Save Memory

The more GPU memory you have for your batch size (BS) the more efficient the GPUs will be at performing compute, and the faster you will complete your task since you will be able to go through data faster.

Of course, this section is crucial for when you get GPU OOM with even BS=1 and you don't want to rent/buy more hardware.

Here is an overview of what features can help to either improve speed or save memory

Method	Speed	Memory
Gradient accumulation	Yes	Yes
Gradient checkpointing	Yes	Yes
Mixed precision training	Yes	No
Batch size	Yes	Yes
Optimizer choice	Yes	Yes
DataLoader	Yes	No
DeepSpeed Zero	No	Yes

Anatomy of Model's Operations

Transformers architecture includes 3 main groups of operations grouped below by compute-intensity.

1. **Tensor Contractions**

   Linear layers and components of Multi-Head Attention all do batched **matrix-matrix multiplications**. These operations are the most compute-intensive part of training a transformer.

2. **Statistical Normalizations**

   Softmax and layer normalization are less compute-intensive than tensor contractions, and involve one or more **reduction operations**, the result of which is then applied via a map.

3. **Element-wise Operators**

   These are the remaining operators: **biases, dropout, activations, and residual connections**. These are the least compute-intensive operations.

This knowledge can be helpful to know when analyzing performance bottlenecks.

This summary is derived from [Data Movement Is All You Need: A Case Study on Optimizing Transformers 2020]

Anatomy of Model's Memory Usage

We've seen that training the model uses much more memory than just putting the model on the GPU. This is because there are many components during training that use GPU memory. The components on GPU memory are the following:

1. model weights
2. optimizer states
3. gradients
4. forward activations saved for gradient computation
5. temporary buffers
6. functionality-specific memory

A typical model trained in mixed precision with AdamW requires 18 bytes per model parameter plus activation memory and temp memory.

Let's look at the details.

Model Weights:

- 4 bytes * number of parameters for fp32 training
- 6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16/bf16 in

memory)

Optimizer States:

- 8 bytes * number of parameters for normal AdamW (maintains 2 states)
- 4 bytes * number of parameters for AdamW running at bf16. See [this work](#) that uses `AnyPrecisionAdamW`.
- 4 bytes * number of parameters for optimizers like SGD with momentum (maintains only 1 state) or LION, or Adafactor (and others) (Adafactor uses some additional memory beside 4 bytes)
- 2 bytes * number of parameters for 8-bit AdamW optimizers like [bitsandbytes](#)

Gradients

- 4 bytes * number of parameters for either fp32 or mixed precision training (gradients are almost always kept in fp32).
- 2 bytes * number of parameters for more recent works where half-precision is used

Forward Activations

- size depends on many factors, the key ones being sequence length, hidden size and batch size.

There are the input and output that are being passed and returned by the forward and the backward functions and the forward activations saved for gradient computation.

Temporary Memory

Additionally there are all kinds of temporary variables which get released once the calculation is done, but in the moment these could require additional memory and could push to OOM. Therefore when coding it's crucial to think strategically about such temporary variables and sometimes to explicitly free those as soon as they are no longer needed.

Functionality-specific memory

Then your software could have special memory needs. For example, when generating text using beam search, the software needs to maintain multiple copies of inputs and outputs.

For **inference**, the math is very similar to training, minus optimizer states and gradients. And for model weights there is just a single multiplier of the number of parameters:

- 6 bytes in mixed precision (4+2)
- 4 bytes in fp32
- 2 bytes in half precision
- 1 byte in quantized int8 precision

Another excellent resource that takes you through the memory needs and other requirements is [Transformer Math 101](#).

Additional GPU memory usage

In addition to the memory usage described in the previous section, there are other consumers of the GPU memory - so you never get the full memory for your model's use.

Preloaded CUDA kernels memory usage

When PyTorch uses CUDA for the first time, it may use up 0.5-2GB of GPU memory, reducing the GPU's total available memory.

The size of allocated memory for cuda kernels varies between different GPUs, and also it can be different between pytorch versions. Let's allocate a 4-byte tensor on cuda and check how much GPU memory is used up upfront.

With `pytorch==1.10.2`:

```
$ CUDA_MODULE_LOADING=EAGER python -c "import torch; x=torch.ones(1).cuda(); free, total = map(lambda x:
```

```
x/2**30, torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GB, {free=:0.2f}GB, {total=:0.2f}GB')"
pt=1.10.2: used=1.78GB, free=77.43GB, total=79.21GB
```

With `pytorch==1.13.1`:

```
$ CUDA_MODULE_LOADING=EAGER python -c "import torch; x=torch.ones(1).cuda(); free, total = map(lambda x:
x/2**30, torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GB, {free=:0.2f}GB, {total=:0.2f}GB')"
pt=1.13.1: used=0.90GB, free=78.31GB, total=79.21GB
```

The older pytorch "wasted" 1.78GB of A100, the newer only 0.9GB, thus saving a whooping 0.9GB, which can be the saving grace for the OOM situations.

`CUDA_MODULE_LOADING=EAGER` is needed in the recent pytorch version if we want to force cuda kernels pre-loading, which are otherwise lazy-loaded on demand. But do not use this setting in production since it's likely to use more memory than needed. The whole point of lazy-loading is to load only the kernels that are needed.

With `pytorch==2.1.1`:

```
$ CUDA_MODULE_LOADING=EAGER python -c "import torch; x=torch.ones(1).cuda(); free, total = map(lambda x:
x/2**30, torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GB, {free=:0.2f}GB, {total=:0.2f}GB')"
pt=2.1.1+cu121: used=0.92GB, free=78.23GB, total=79.15GB
```

As compared to the lazy mode:

```
$ python -c "import torch; x=torch.ones(1).cuda(); free, total = map(lambda x: x/2**30,
torch.cuda.mem_get_info()); \
used=total-free; print(f'pt={torch.__version__}: {used=:0.2f}GB, {free=:0.2f}GB, {total=:0.2f}GB')"
pt=2.1.1+cu121: used=0.47GB, free=78.68GB, total=79.15GB
```

There is a 450MB difference, but here we only loaded kernels to do `torch.ones` - the actual memory allocated at run time with other code using torch API will be somewhere between 0.47 and 0.92GB.

Memory fragmentation

As the model allocates and frees tensors, the memory could fragment. That is there could be enough free memory to allocate, say, 1GB of contiguous memory, but it could be available in 100s of small segments spread out through the memory and thus even though the memory is available it can't be used unless very small allocations are made.

Environment variable `PYTORCH_CUDA_ALLOC_CONF` comes to help and allows you to replace the default memory allocation mechanisms with more efficient ones. For more information see Memory management.

Batch sizes

First, there are usually two batch sizes:

1. micro batch size (MBS), also known as batch size per gpu - this is how many samples a single gpu consumes during a

model's single `forward` call.

2. global batch size (GBS) - this is the total amount of samples consumed between two optimizer steps across all participating GPUs.

Model replica is how many gpus are needed to fit the full model.

- If the model fits into a single GPU a model replica takes 1 GPU. Usually then one can use multiple GPUs to perform Data Parallelism
- If the model doesn't fit into a single GPU, it'd usually require some sort of sharding technique - it can be Tensor Parallelism (TP), Pipeline Parallelism (PP), or ZeRO Data Parallelism (ZeRO-DP).

You can have as many data streams as there are replicas. Which is the same as the value of DP.

- So in the simple case of a model fitting into a single GPU. There are as many data streams as there are GPUs. DP=N_GPUS
- when the model doesn't fit onto a single GPU, then `DP=N_GPUs/(TP*PP)` in the case of 3D parallelism and DP=ZeRO-DP in the case of ZeRO parallelism.

Going back to our global batch size (GBS) it's calculated as:

```
GBS = MBS*DP
```

So if you have 8 gpus (N_GPUS=8) and your MBS=4 and you do DP you end up with having GBS=32 because:

```
GBS = MBS*DP = 4*8 = 32
```

If you use TP with a degree of 2 (TP=2) and PP with a degree of 2 (PP=2) this means each model replica takes 4 gpus (`TP*PP`), and thus with N_GPUS=8

```
DP = N_GPUS/(TP*PP) = 8 / (2*2) = 2
```

and now GBS becomes:

```
GBS = MBS*DP = 4*2 = 8
```

If your training setup requires Gradient Accumulation, one usually defines the interval of how many steps to wait before performing a gradient accumulation. The term is usually Gradient Accumulation Steps (GAS). If GAS=4 (i.e. sync grads every 4 steps) and TP=1, PP=1 and DP=8:

```
DP = N_GPUS/(TP*PP) = 8 / (1*1) = 8
GBS = MBS*DP*GAS = 4*8*4 = 128
```

Typically you want to make the micro batch size as large as possible so that the GPU memory is close to being full, but not too full.

With large models usually there is not much free GPU memory left to have a large micro batch size, therefore every

additional sample you can fit is important.

While it's super important that sequence length and hidden size and various other hyper parameters are high multiples of 2 (64, 128 and higher) to achieve the highest performance, because in most models the batch dimension is flattened with the sequence length dimension during the compute the micro batch size alignment usually has little to no impact on performance.

Therefore if you tried to fit a micro batch size of 8 and it OOM'ed, but 7 fits - use the latter rather than 4. The higher the batch size the more samples you will be able to fit into a single step.

Of course, when using hundreds of GPUs your global batch size may become very large. In that case you might use a smaller micro batch size or use less GPUs or switch to a different form of data parallelism so that the GPUs work more efficiently.

Gradient Accumulation

The idea behind gradient accumulation is to instead of calculating the gradients for the whole batch at once to do it in smaller steps. The way we do that is to calculate the gradients iteratively in smaller batches by doing a forward and backward pass through the model and accumulating the gradients in the process. When enough gradients are accumulated we run the model's optimization step. This way we can easily increase the overall batch size to numbers that would never fit into the GPU's memory. In turn, however, the added forward and backward passes can slow down the training a bit.

Gradient Accumulation Steps (GAS) is the definition of how many steps are done w/o updating the model weights.

When using Pipeline parallelism a very large Gradient Accumulation is a must to keep the pipeline's bubble to the minimum.

Since the optimizer step isn't performed as often with gradient accumulation there is an additional speed up here as well.

The following benchmarks demonstrate how increasing the gradient accumulation steps improves the overall throughput (20-30% speedup):

- RTX-3090
- A100

When data parallelism is used gradient accumulation further improves the training throughput because it reduces the number of gradient reduction calls, which is typically done via the `all_reduce` collective which costs a 2x size of gradients to be reduced. So for example, if one goes from GAS=1 to GAS=8 in `DistributedDataParallelism` (DDP) the network overhead is reduced by 8x times, which on a slow inter-node network can lead to a noticeable improvement in the training throughput.

Gradient checkpointing

Gradient Checkpointing is also known as Activation Recompution, Activation Checkpointing and Checkpoint Activations.

This methodology is only relevant for training, and not during inference.

Enabling gradient checkpointing allows one to trade training throughput for accelerator memory. When this feature is activated instead of remembering the outputs of, say, transformer blocks until the `backward` pass is done, these outputs are dropped. This frees up huge amounts of accelerator memory. But, of course, a `backward` pass is not possible without having the outputs of `forward` pass, and thus they have to be recalculated.

This, of course, can vary from model to model, but typically one pays with about 20-25% decrease in throughput, but since a huge amount of gpu memory is liberated, one can now increase the batch size per gpu and thus overall improve the effective throughput of the system. In some cases this allows you to double or quadruple the batch size if you were already able to do a small batch size w/o OOM. (Recent papers report as high as 30-40% additional overhead.)

Activation checkpointing and gradient checkpointing are 2 terms for the same methodology.

For example, in HF Transformers models you do `model.gradient_checkpointing_enable()` to activate it in your custom

Trainer or if you use the HF Trainer then you'd activate it with `--gradient_checkpointing 1`.

XXX: expand on new tech from the paper: [Reducing Activation Recomputation in Large Transformer Models] which found a way to avoid most activation recomputations and thus save both memory and compute.

Memory-efficient optimizers

The most common optimizer is Adam. It and its derivatives all use 8 bytes per param (2x fp32 tensors - one for each momentum), which account for almost half the memory allocation for the model, optimizer and gradients. So at times using other optimizers may save the day, if they successfully train that is. Not all optimizers are suitable for all training tasks.

4-byte optimizers:

- There are optimizers like Adafactor that need only 4 bytes. Same goes for the recently invented LION optimizer.
- `AnyPrecisionAdamW`. Some courageous souls try to do the whole training in BF16 (not mixed precision!), including the optimizer and thus need only 4 bytes per parameter for optim states. See this work. Hint: this optimizer requires Kahan summation and/or stochastic rounding, see Revisiting BFloat16 Training (2020). You need only 8 bytes per parameter for weights, optim states and gradients here! Instead of 18!

2-byte optimizers:

- There are quantized solutions like `bnb.optim.Adam8bit` which uses only 2 bytes instead of 8 (1 byte per momentum). You can get it from here. Once installed, if you're using HF Trainer, you can enable it on with just passing `--optim adamw_bnb_8bit`!

For speed comparisons see this benchmark Speed-wise:`apex`'s `apex.optimizers.FusedAdam` optimizer is so far the fastest implementation of Adam. Since pytorch-2.0 torch.optim.AdamW added support for `fused=True` option, which brings it almost on par with `apex.optimizers.FusedAdam`.

Model execution speed

`forward` vs `backward` Execution Speed

For convolutions and linear layers there are 2x flops in the backward compared to the forward, which generally translates into ~2x slower (sometimes more, because sizes in the backward tend to be more awkward). Activations are usually bandwidth-limited, and it's typical for an activation to have to read more data in the backward than in the forward (e.g. activation forward reads once, writes once, activation backward reads twice, `gradOutput` and output of the forward, and writes once, `gradInput`).

Memory profiler tools

In this chapter we discussed the theoretical math of how much this or that feature should consume in MBs of memory. But often in reality things aren't exactly the same. So you plan for a certain model size and batch sizes but when you come to use it suddenly there is not enough memory. So you need to work with your actual code and model and see which part takes how much memory and where things got either miscalculated or some additional missed overhead hasn't been accounted for.

You'd want to use some sort of memory profiler for that purpose. There are various memory profilers out there.

One useful tool that I developed for quick and easy profiling of each line or block of code is IPyExperiments. You just need to load your code into a jupyter notebook and it'll automatically tell you how much CPU/GPU memory each block allocates/frees. So e.g. if you want to see how much memory loading a model took, and then how much extra memory a single inference step took - including peak memory reporting.

Vector and matrix size divisibility

One gets the most efficient performance when batch sizes and input/output neuron counts are divisible by a certain number, which typically starts at 8, but can be much higher as well. That number varies a lot depending on the specific hardware being used and the dtype of the model.

For fully connected layers (which correspond to GEMMs), NVIDIA provides recommendations for input/output neuron counts and batch size.

Tensor Core Requirements define the multiplier based on the dtype and the hardware. For example, for fp16 a multiple of 8 is recommended, but on A100 it's 64!

For parameters that are small, there is also Dimension Quantization Effects to consider, this is where tiling happens and the right multiplier can have a significant speedup.

Tile and wave quantization

XXX

Number/size of Attention heads

XXX

Fault Tolerance

Regardless of whether you own the ML training hardware or renting it by the hour, in this ever speeding up domain of ML, finishing the training in a timely matter is important. As such if while you were asleep one of the GPUs failed or the checkpoint storage run out of space which led to your training crashing, you'd have discovered upon waking that many training hours were lost.

Due the prohibitively high cost of ML hardware, it'd be very difficult to provide redundancy fail-over solutions as it's done in Web-services. Nevertheless making your training fault-tolerant is achievable with just a few simple recipes.

As most serious training jobs are performed in a SLURM environment, it'll be mentioned a lot, but most of this chapter's insights are applicable to any other training environments.

Always plan to have more nodes than needed

The reality of the GPU devices is that they tend to fail. Sometimes they just overheat and shut down, but can recover, at other times they just break and require a replacement.

The situation tends to ameliorate as you use the same nodes for some weeks/months as the bad apples get gradually replaced, but if you are lucky to get a new shipment of GPUs and especially the early GPUs when the technology has just come out, expect a sizeable proportion of those to fail.

Therefore, if you need 64 nodes to do your training, make sure that you have a few spare nodes and study how quickly you can replace failing nodes should the spares that you have not be enough.

It's hard to predict what the exact redundancy percentage should be, but 5-10% shouldn't be unreasonable. The more you're in a crunch to complete the training on time, the higher the safety margin should be.

Once you have the spare nodes available, validate that your SLURM environment will automatically remove any problematic nodes from the pool of available nodes so that it can automatically replace the bad nodes with the good ones.

If you use a non-SLURM scheduler validate that it too can do unmanned bad node replacements.

You also need at least one additional node for running various preventative watchdogs (discussed later in this chapter), possibly offloading the checkpoints and doing cleanup jobs.

Queue up multiple training jobs

The next crucial step is to ensure that if the training crashed, there is a new job lined up to take place of the previous one.

Therefore, when a training is started, instead of using:

```
sbatch train.slurm
```

You'd want to replace that with:

```
sbatch --array=1-10%1 train.slurm
```

This tells SLURM to book a job array of 10 jobs, and if one of the job completes normally or it crashes, it'll immediately schedule the next one.

footnote: `%1` in `--array=1-10%1` tells SLURM to launch the job array serially - one job at a time.

If you have already started a training without this provision, it's easy to fix without aborting the current job by using the `--dependency` argument:

```
sbatch --array=1-10%1 --dependency=CURRENTLY_RUNNING_JOB_ID train.slurm
```

So if your launched job looked like this:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
     JOBID PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME
NODELIST(REASON)
       87    prod    my-training-10b  RUNNING 2-15:52:19 1-16:00:00   64    2023-10-07T01:26:28
node-[1-63]
```

You will not that the current's `JOBID=87` and now you can use it in:

```
sbatch --array=1-10%1 --dependency=87 train.slurm
```

and then the new status will appear as:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
     JOBID PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME
NODELIST(REASON)
       87    prod    my-training-10b  RUNNING 2-15:52:19 1-16:00:00   64    2023-10-07T01:26:28
node-[1-63]
 88_[10%1]   prod    my-training-10b  PENDING       0:00 1-16:00:00   64                   N/A
(Dependency)
```

So you can see that an array of 10 jobs (`88_[10%1]`) was appended to be started immediately after the current job (`87`) completes or fails.

Granted that if the condition that lead to the crash is still there the subsequent job will fail as well. For example, if the storage device is full, no amount of restarts will allow the training to proceed. And we will discuss shortly how to avoid this situation.

But since the main reason for training crashes is failing GPUs, ensuring that faulty nodes are automatically removed and the new job starts with a new set of nodes makes for a smooth recovery from the crash.

In the SLURM lingo, the removed nodes are given a new status called `drained`. Here is an example of a hypothetical SLURM cluster:

```
$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
prod*       up   infinite       4  drain node-[0-3]
prod*       up   infinite      47  alloc node-[4-51]
```

```
prod*        up   infinite       23   idle node-[52-73]
```

Here we have 47 nodes being used (`alloc`), 23 available (`idle`) and 4 unavailable (`drained`).

The sysadmin is expected to periodically check the drained nodes, fix or replace them and then make them again available to be used by changing their state to `idle`.

The other approach is to daisy-chain jobs via `--dependency` as explained here. Both of these approaches could also be combined.

How do you know when the job array or a daisy chain should not resume - well, normally the training loop will exit immediately if it knows the job is done. But you could also add features like kill switch which are even easier to use to prevent a job array from running.

Frequent checkpoint saving

Whenever the training job fails, many hours of training can be lost. This problem is mitigated by a frequent checkpoint saving. When the training is resumed it'll continue from the last checkpoint saved. If the failure occurred 12 hours after the last checkpoint has been saved, 12 hours of training is lost and needs to be re-done. This can be very expensive if the training uses hundreds of GPUs.

In theory one could save a checkpoint every 10 minutes and only ever lose 10 minutes of training time, but this too would dramatically delay the reaching of the finish line because large models can't be saved quickly and if the saving time starts to create a bottleneck for the training this approach becomes counterproductive.

Depending on your checkpointing methodology and the speed of your IO storage partition the saving of a large model can take from dozens of seconds to several minutes. Therefore, the optimal approach to saving frequency lies somewhere in the middle.

The math is quite simple - measure the amount of time it takes to save the checkpoint, multiply it by how many times you'd want to save it and see how much of an additional delay the checkpoint saving will contribute to the total training time.

Use case: While training BLOOM-176B we had an incredibly fast GPFS over NVME filesystem and it took only 40 seconds to save a 2.3TB checkpoint written concurrently on 384 processes. We saved a checkpoint approximately every 3 hours. As we trained for about 3 months, that means that we saved about 720 checkpoints (`90 days * 24h / 3h`) - that is an additional 8 hours was spent just saving the checkpoints (`720 times * 40 secs / 3600 secs`) - or ~0.37% of the total training time (`8h / (90 days * 24 hours)`). Now say if the IO were to be 5 times slower, which is not uncommon on the cloud unless one pays for premium IO, that would have become 2% of the training time, which would be quite significant.

footnote: If you don't have a large local storage and you have to offload the checkpoints to the cloud, make sure that the 2 most frequent checkpoints remain local to allow for a quick resume. The reason for 2 and not 1, is that it's possible that the very last checkpoint got corrupted or didn't finish saving if a crash occurred during its saving.

Kill switch

In many SLURM environments users have no `sudo` access and when one user started a training and went to sleep, and then a problem has been discovered, the other users can't easily stop the training and restart it again.

This was the situation during BLOOM-176B training and we implemented a kill-switch to handle that. The mechanism is very simple. The training loop polls for a specific file to appear before starting a new iteration and if the file is there the program saves the checkpoint and exits, allowing users other than the one who started the previous training to change things and restart it again. An additional poll was added at the very beginning of `main` so that if there was a long job array queued by the user who is asleep they could be "burned through" quickly by getting each job exit quickly on start.

This is also discussed here.

This facility helps to minimize the amount of wasted training time.

Save switch

While mentioning the kill switch, it might be good to quickly mention its cousin, a save switch. Similarly to the kill switch the save switch is a variation of the former where instead of stopping the training, if the training loop discovers that a save-switch file appears - it will save a checkpoint, but will continue training. It'll also automatically remove the save-switch from the file-system, so that it won't accidentally start saving a checkpoint after every iteration.

This feature can be very useful for those who watch the training charts. If one sees an interesting or critical situation in the training loss or some other training metric one can quickly ask the training program to save the checkpoint of interest and be able to later reproduce the current situation at will.

The main use of this feature is around observing training loss spikes and divergences.

(note-to-self: better belongs to instabilities chapter)

Prevention

The easiest way to avoid losing training time is to prevent certain types of problems from happening. While one can't prevent a GPU from failing, other than ensuring that adequate cooling is provided, one can certainly ensure that there is enough of disk space remaining for the next few days of training. This is typically done by running scheduled watchdogs that monitor various resources and alert the operator of possible problems long before they occur.

Scheduled Watchdogs

Before we discuss the various watchdogs it's critical that you have a mechanism that allows you to run scheduled jobs. In the Unix world this is implemented by the `crontab` facility.

Here is an example of how `~/bin/watch-fs.sh` can be launched every hour:

```
0 * * * * ~/bin/watch-fs.sh
```

The link above explains how to configure a crontab job to run at various other frequencies.

To setup a crontab, execute `crontab -e` and check which jobs are scheduled `crontab -l`.

The reason I don't go into many details is because many SLURM environments don't provide access to the `crontab` facility. And therefore one needs to use other approaches to scheduling jobs.

The section on Crontab Emulation discusses how to implement crontab-like SLURM emulation and also Self-perpetuating SLURM jobs.

Notification facility

Then you need to have one or more notification facilities.

The simplest one is to use email to send alerts. To make this one work you need to ensure that you have a way to send an email from the SLURM job. If it isn't already available you can request this feature from your sysadmin or alternatively you might be able to use an external SMTP server provider.

In addition to email you could probably also setup other notifications, such as SMS alerting and/or if you use Slack to send slack-notifications to a channel of your choice.

Once you understand how to schedule watchdogs and you have a notification facility working let's next discuss the critical watchdogs.

Is-job-running watchdog

The most obvious watchdog is one which checks that there is a training SLURM job running or more are scheduled to run.

Here is an example slurm-status.py that was used during BLOOM-176B training. This watchdog was sending an email if a job was detected to be neither running nor scheduled and it was also piping its check results into the main training's log file. As we used Crontab Emulation, we simply needed to drop slurm-status.slurm into the `cron/cron.hourly/` folder and the previously launched SLURM crontab emulating scheduler would launch this check approximately once an hour.

The key part of the SLURM job is:

```
tools/slurm-status.py --job-name $WATCH_SLURM_NAME 2>&1 | tee -a $MAIN_LOG_FILE
```

which tells the script which job name to watch for, and you can also see that it logs into a log file.

For example, if you launched the script with:

```
tools/slurm-status.py --job-name my-training-10b
```

and the current status report shows:

```
$ squeue -u `whoami` -o "%.10i %9P %20j %.8T %.10M %.8l %.6D %.20S %R"
  JOBID    PARTITION NAME             STATE       TIME   TIME_LIM    NODES  START_TIME
NODELIST(REASON)
    87     prod      my-training-10b  RUNNING 2-15:52:19 1-16:00:00  64    2023-10-07T01:26:28 node-[1-63]
```

then all is good. But if `my-training-10b` job doesn't show the alert will be sent.

You can now adapt these scripts to your needs with minimal changes of editing the path and email addresses. And if it wasn't you who launched the job then replace `whoami` with the name of the user who launched it. `whoami` only works if it was you who launched it.

Is-job-hanging watchdog

If the application is doing `torch.distributed` or alike and a hanging occurs during one of the collectives, it'll eventually timeout and throw an exception, which would restart the training and one could send an alert that the job got restarted.

However, if the hanging happens during another syscall which may have no timeout, e.g. reading from the disk, the application could easily hang there for hours and nobody will be the wiser.

Most applications do periodic logging, e.g., most training log the stats of the last N steps every few minutes. Then one could check if the log file has been updated during the expected time-frame - and if it didn't - send an alert. You could write your own, or use io-watchdog for that.

Low disk space alerts

The next biggest issue is running out of disk space. If your checkpoints are large and are saved frequently and aren't offloaded elsewhere it's easy to quickly run out of disk space. Moreover, typically multiple team members share the

same cluster and it could be that your colleagues could quickly consume a lot of disk space. Ideally, you'd have a storage partition that is dedicated to your training only, but often this is difficult to accomplish. In either case you need to know when disk space is low and space making action is to be performed.

Now what should be the threshold at which the alerts are triggered. They need to be made not too soon as users will start ignoring these alerts if you start sending those at say, 50% usage. But also the percentage isn't always applicable, because if you have a huge disk space shared with others, 5% of that disk space could translate to many TBs of free disk space. But on a small partition even 25% might be just a few TBs. Therefore really you should know how often you write your checkpoints and how many TBs of disk space you need daily and how much disk space is available.

Use case: During BLOOM training we wrote a 2.3TB checkpoint every 3 hours, therefore we were consuming 2.6TB a day!

Moreover, often there will be multiple partitions - faster IO partitions dedicated to checkpoint writing, and slower partitions dedicated to code and libraries, and possibly various other partitions that could be in use and all of those need to be monitored if their availability is required for the training not crashing.

Here is another caveat - when it comes to distributed file systems not all filesystems can reliably give you a 100% of disk space you acquired. In fact with some of those types you can only reliably use at most ~80% of the allocated storage space. The problem is that these systems use physical discs that they re-balance at the scheduled periods or triggered events, and thus any of these individual discs can reach 100% of their capacity and lead to a failed write, which would crash a training process, even though `df` would report only 80% space usage on the partition. We didn't have this problem while training BLOOM-176B, but we had it when we trained IDEFICS-80B - 80% there was the new 100%. How do you know if you have this issue - well, usually you discover it while you prepare for the training.

And this is not all. There is another issue of inodes availability and some storage partitions don't have very large inode quotas. Python packages are notorious for having hundreds to thousands of small files, which combined take very little total space, but which add up to tens of thousands of files in one's virtual environment and suddenly while one has TBs of free disk space available, but runs out of free inodes and discovering their training crashing.

Finally, many distributed partitions don't show you the disk usage stats in real time and could take up to 30min to update.

footnote: Use `df -ih` to see the inodes quota and the current usage.

footnote: Some filesystems use internal compression and so the reported disk usage can be less than reality if copied elsewhere, which can be confusing.

So here is fs-watchdog.py that was used during BLOOM-176B training. This watchdog was sending an email if any of the storage requirements thresholds hasn't been met and here is the corresponding fs-watchdog.slurm that was driving it.

If you study the watchdog code you can see that for each partition we were monitoring both the disk usage and inodes. We used special quota tools provided by the HPC to get instant stats for some partitions, but these tools didn't work for all partitions and there we had to fallback to using `df` and even a much slower `du`. As such it should be easy to adapt to your usecase.

Dealing with slow memory leaks

Some programs develop tiny memory leaks which can be very difficult to debug. Do not confuse those with the usage of MMAP where the program uses the CPU memory to read quickly read data from and where the memory usage could appear to grow over time, but this is not real as this memory gets freed when needed. You can read A Deep Investigation into MMAP Not Leaking Memory to understand why.

Of course, ideally one would analyze their software and fix the leak, but at times the leak could be coming from a 3rd party package or can be very difficult to diagnose and there isn't often the time to do that.

When it comes to GPU memory, there is the possible issue of memory fragmentation, where over time more and more tiny unused memory segments add up and make the GPU appear to have a good amount of free memory, but when the program tries to allocate a large tensor from this memory it fails with the OOM error like:

```
RuntimeError: CUDA out of memory. Tried to allocate 304.00 MiB (GPU 0; 8.00 GiB total capacity;
142.76 MiB already allocated; 6.32 GiB free; 158.00 MiB reserved in total by PyTorch)
```

In this example if there are 6.32GB free, how comes that 304MB couldn't be allocated.

One of the approaches my team developed during IDEFICS-80B training where there was some tiny CPU memory leak that would often take multiple days to lead to running out of CPU memory was to install a watchdog inside the training loop that would check the memory usage and if a threshold was reached it'd voluntarily exit the training loop. The next training job would then resume with all the CPU memory reclaimed.

footnote: The reality of machine learning trainings is that not all problems can be fixed with limited resources and often times a solid workaround provides for a quicker finish line, as compared to "stopping the presses" and potentially delaying the training for weeks, while trying to figure out where the problem is. For example we trained BLOOM-176B with `CUDA_LAUNCH_BLOCKING=1` because the training would hang without it and after multiple failed attempts to diagnose that we couldn't afford any more waiting and had to proceed as is. Luckily this environment variable that normally is used for debug purposes and which in theory should make some CUDA operations slower didn't actually make any difference to our throughput. But we have never figured out what the problem was and today it doesn't matter at all that we haven't, as we moved on with other projects which aren't impacted by that issue.

The idea is similar to the kill and save switches discussed earlier, but here instead of polling for a specific file appearance we simply watch how much resident memory is used. For example here is how you'd auto-exit if the OS shows only 5% of the virtual cpu memory remain:

```
import psutil
for batch in iterator:
    total_used_percent = psutil.virtual_memory().percent
    if total_used_percent > 0.95:
        print(f"Exiting early since the cpu memory is almost full: ({total_used_percent}%)")
        save_checkpoint()
        sys.exit()

    train_step(batch)
```

Similar heuristics could be used for setting a threshold for GPU memory usage, except one needs to be aware of cuda tensor caching and python garbage collection scheduling, so to get the actual memory usage you'd need to do first run the garbage collector then empty the cuda cache and only then you will get real memory usage stats and then gracefully exit the training if the GPU is too close to being full.

```
import gc
import torch

for batch in iterator:
    gc.collect()
    torch.cuda.empty_cache()

    # get mem usage in GBs and exit if less than 2GB of free GPU memory remain
    free, total = map(lambda x: x/2**30, torch.cuda.mem_get_info());
    if free < 2:
        print(f"Exiting early since the GPU memory is almost full: ({free}GB remain)")
```

```
        save_checkpoint()
        sys.exit()

    train_step(batch)
```

footnote: don't do this unless you really have to, since caching makes things faster. Ideally figure out the fragmentation issue instead. For example, look up `max_split_size_mb` in the doc for [`PYTORCH_CUDA_ALLOC_CONF`] as it controls how memory is allocated. Some frameworks like [Deepspeed] solve this by pre-allocating tensors at start time and then reuse them again and again preventing the issue of fragmentation altogether.

footnote: this simplified example would work for a single node. For multiple nodes you'd need to gather the stats from all participating nodes and find the one that has the least amount of memory left and act upon that.

Dealing with forced resource preemption

Earlier you have seen how the training can be gracefully stopped with a [kill switch solution] and it's useful when you need to stop or pause the training on demand.

On HPC clusters SLURM jobs have a maximum runtime. A typical one is 20 hours. This is because on HPCs resources are shared between multiple users/groups and so each is given a time slice to do compute and then the job is forcefully stopped, so that other jobs could use the shared resources.

footnote: this also means that you can't plan how long the training will take unless your jobs run with the highest priority on the cluster. If your priority is not the highest it's not uncommon to have to wait for hours and sometimes days before your job resumes.

One could, of course, let the job killed and hope that not many cycles were spent since [the last checkpoint was saved] and then let the job resume from this checkpoint, but that's quite wasteful and best avoided.

The efficient solution is to gracefully exit before the hard tile limit is hit and the job is killed by SLURM.

First, you need to figure out how much time your program needs to gracefully finish. This typically requires 2 durations:

1. how long does it take for a single iteration to finish if you have just started a new iteration
2. how long does it take to save the checkpoint

If, for example, the iteration takes 2 minutes at most and the checkpoint saving another 2 minutes, then you need at least 4 minutes of that grace time. To be safe I'd at least double it. There is no harm at exiting a bit earlier, as no resources are wasted.

So, for example, let's say your HPC allows 100 hour jobs, and then your slurm script will say:

```
#SBATCH --time=100:00:00
```

Approach A. Tell the program at launch time when it should start the exiting process:

```
srun ... torchrun ... --exit-duration-in-mins 5990
```

100h is 6000 minutes and so here we give the program 10 mins to gracefully exit.

And when you start the program you create a timer and then before every new iteration starts you check if the time limit is reached. If it is you save the checkpoint and exit.

case study: you can see how this was set [in the BLOOM training job](#) and then acted upon [here](#):

```
        # Exiting based on duration
        if args.exit_duration_in_mins:
            train_time = (time.time() - _TRAIN_START_TIME) / 60.0
            done_cuda = torch.cuda.IntTensor(
                [train_time > args.exit_duration_in_mins])
            torch.distributed.all_reduce(
                done_cuda, op=torch.distributed.ReduceOp.MAX)
            done = done_cuda.item()
            if done:
                if not saved_checkpoint:
                    save_checkpoint_and_time(iteration, model, optimizer,
                                             lr_scheduler)
                print_datetime('exiting program after {} minutes'.format(train_time))
                sys.exit()
```

As you can see since the training is distributed we have to synchronize the exiting event across all ranks

You could also automate the derivation, by retrieving the `EndTime` for the running job:

```
$ scontrol show -d job $SLURM_JOB_ID | grep Time
   RunTime=00:00:42 TimeLimit=00:11:00 TimeMin=N/A
   SubmitTime=2023-10-26T15:18:01 EligibleTime=2023-10-26T15:18:01
   AccrueTime=2023-10-26T15:18:01
   StartTime=2023-10-26T15:18:01 EndTime=2023-10-26T15:18:43 Deadline=N/A
```

and then comparing with the current time in the program and instead setting the graceful exit period. There are other timestamps and durations that can be retrieved as it can be seen from the output.

Approach B. Sending a signal X minutes before the end

In your sbatch script you could set

```
#SBATCH --signal=USR1@600
```

and then SLURM will send a `SIGUSR1` signal to your program 10min before job's end time.

footnote: normally SLURM schedulers send a `SIGTERM` signal about 30-60 seconds before the job's time is up, and just as the time is up it will send a `SIGKILL` signal if the job is still running. `SIGTERM` can be caught and acted upon but 30 seconds is not enough time to gracefully exit a large model training program.

Let's demonstrate how the signal sending and trapping works. In terminal A, run:

```
python -c "
import time, os, signal

def sighandler(signum, frame):
```

```
    print('Signal handler called with signal', signum)
    exit(0)

signal.signal(signal.SIGUSR1, sighandler)
print(os.getpid())
time.sleep(1000)
"
```

it will print the pid of the process, e.g., `4034989` and will go to sleep (emulating real work). In terminal B now send `SIGUSR1` signal to the python program in terminal A with:

```
kill -s USR1 4034989
```

The program will trap this signal, call the `sighandler` which will now print that it was called and exit.

```
Signal handler called with signal 10
```

`10` is the numerical value of `SIGUSR1`.

So here is the same thing with the SLURM setup:

```
$ cat sigusr1.slurm
#SBATCH --job-name=sigusr1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=0:03:00
#SBATCH --partition=mypartition
#SBATCH --output=%x-%j.out
#SBATCH --signal=USR1@170

srun python -c "
import time, os, signal

def sighandler(signum, frame):
    print('Signal handler called with signal', signum)
    exit(0)

signal.signal(signal.SIGUSR1, sighandler)
print(os.getpid())
time.sleep(1000)
"
```

In the SLURM script we told SLURM to send the program a signal 170 seconds before its end and the job itself was set to run for 180 secs (3 mins).

When this job has been scheduled:

```
sbatch sigusr1.slurm
```

10 seconds (`180-170`) after the job started, it will exit with the log:

```
58307
Signal handler called with signal 10
```

which means the job had a pid `58307` and it caught `SIGUSR1` (`10`) and it exited.

Now that you understand how this machinery works, instead of immediate `exit(0)` you can set exit-asap flag, finish the currently run iteration, check that the flag is up, save the checkpoint and exit. This is very similar to the code shown in Approach A above.

Contributors

Adam Moody,

Multi-node

Guides

- emulate-multi-node.md - instructions on how to emulate a multi-node setup using just a single node - we use the `deepspeed` launcher here.

Tools

- printflock.py - a tiny library that makes your `print` calls non-interleaved in a multi-gpu environment.
- multi-gpu-non-interleaved-print.py - a `flock`-based wrapper around `print` that prevents messages from getting interleaved when multiple processes print at the same time - which is the case with `torch.distributed` used with multiple-gpus.
- all_reduce_bench.py - a tool to benchmark the real network bandwidth while performing all_reduce on a largish amount of data. This is useful to find out what one gets in reality as compared to the promised spec.
- all_gather_object_vs_all_reduce.py - a quick benchmark showing 23x speed up when moving from `all_gather_object` to `all_reduce` when collecting completion status from the process group. e.g. when implementing some sort of all-processes-are-done flag. This technique is usually used for synchronizing gpus when they may complete at different number of iterations - which one needs for inference over multiple DP channels, or when one wants to sync a `StopIteration` event in `DataLoader`. See also all_gather_object_vs_all_gather.py.

Emulate a multi-node setup using just a single node

The goal is to emulate a 2-node environment using a single node with 2 GPUs (for testing purposes). This, of course, can be further expanded to larger set ups.

We use the `deepspeed` launcher here. There is no need to actually use any of the deepspeed code, it's just easier to use its more advanced capabilities. You will just need to install `pip install deepspeed`.

The full setup instructions follow:

1. Create a `hostfile`:

```
$ cat hostfile
worker-0 slots=1
worker-1 slots=1
```

2. Add a matching config to your ssh client

```
$ cat ~/.ssh/config
[...]

Host worker-0
    HostName localhost
    Port 22
Host worker-1
    HostName localhost
    Port 22
```

Adapt the port if it's not 22 and the hostname if `localhost` isn't it.

3. As your local setup is probably password protected ensure to add your public key to `~/.ssh/authorized_keys`

The `deepspeed` launcher explicitly uses no-password connection, e.g. on worker0 it'd run: `ssh -o PasswordAuthentication=no worker-0 hostname`, so you can always debug ssh setup using:

```
$ ssh -vvv -o PasswordAuthentication=no worker-0 hostname
```

4. Create a test script to check both GPUs are used.

```
$ cat test1.py
import os
import time
import torch
import deepspeed
import torch.distributed as dist
```

```
# critical hack to use the 2nd gpu (otherwise both processes will use gpu0)
if os.environ["RANK"] == "1":
    os.environ["CUDA_VISIBLE_DEVICES"] = "1"

dist.init_process_group("nccl")
local_rank = int(os.environ.get("LOCAL_RANK"))
print(f'{dist.get_rank()=}, {local_rank=}')

x = torch.ones(2**30, device=f"cuda:{local_rank}")
time.sleep(100)
```

Run:

```
$ deepspeed -H hostfile test1.py
[2022-09-08 12:02:15,192] [INFO] [runner.py:415:main] Using IP address of 192.168.0.17 for node worker-0
[2022-09-08 12:02:15,192] [INFO] [multinode_runner.py:65:get_cmd] Running on the following workers:
worker-0,worker-1
[2022-09-08 12:02:15,192] [INFO] [runner.py:504:main] cmd = pdsh -S -f 1024 -w worker-0,worker-1 export
PYTHONPATH=/mnt/nvme0/code/huggingface/multi-node-emulate-ds;  cd /mnt/nvme0/code/huggingface/
multi-node-emulate-ds; /home/stas/anaconda3/envs/py38-pt112/bin/python -u -m deepspeed.launcher.launch
--world_info=eyJ3b3JrZXItMCI6IFswXSwgIndvcmtlci0xIjogWzBdfQ== --node_rank=%n --master_addr=192.168.0.17
--master_port=29500 test1.py
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0': [0],
'worker-1': [0]}
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1, node_rank=0
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:155:main] global_rank_mapping=defaultdict(<class
'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:156:main] dist_world_size=2
worker-0: [2022-09-08 12:02:16,517] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0': [0],
'worker-1': [0]}
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1, node_rank=1
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:155:main] global_rank_mapping=defaultdict(<class
'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:156:main] dist_world_size=2
worker-1: [2022-09-08 12:02:16,518] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-1: torch.distributed.get_rank()=1, local_rank=0
worker-0: torch.distributed.get_rank()=0, local_rank=0
worker-1: tensor([1., 1., 1.,  ..., 1., 1., 1.], device='cuda:0')
worker-0: tensor([1., 1., 1.,  ..., 1., 1., 1.], device='cuda:0')
```

If the ssh set up works you can run `nvidia-smi` in parallel and observe that both GPUs allocated ~4GB of memory from `torch.ones` call.

Note that the script hacks in `CUDA_VISIBLE_DEVICES` to tell the 2nd process to use gpu1, but it'll be seen as `local_rank==0` in both cases.

5. Finally, let's test that NCCL collectives work as well

Script adapted from torch-distributed-gpu-test.py to just tweak `os.environ["CUDA_VISIBLE_DEVICES"]`

```
$ cat test2.py
import deepspeed
import fcntl
import os
import socket
import time
import torch
import torch.distributed as dist

# a critical hack to use the 2nd GPU by the 2nd process (otherwise both processes will use gpu0)
if os.environ["RANK"] == "1":
    os.environ["CUDA_VISIBLE_DEVICES"] = "1"

def printflock(*msgs):
    """ solves multi-process interleaved print problem """
    with open(__file__, "r") as fh:
        fcntl.flock(fh, fcntl.LOCK_EX)
        try:
            print(*msgs)
        finally:
            fcntl.flock(fh, fcntl.LOCK_UN)

local_rank = int(os.environ["LOCAL_RANK"])
torch.cuda.set_device(local_rank)
device = torch.device("cuda", local_rank)
hostname = socket.gethostname()

gpu = f"[{hostname}-{local_rank}]"

try:
    # test distributed
    dist.init_process_group("nccl")
    dist.all_reduce(torch.ones(1).to(device), op=dist.ReduceOp.SUM)
    dist.barrier()
    print(f'{dist.get_rank()=}, {local_rank=}')

    # test cuda is available and can allocate memory
    torch.cuda.is_available()
    torch.ones(1).cuda(local_rank)

    # global rank
    rank = dist.get_rank()
    world_size = dist.get_world_size()

    printflock(f"{gpu} is OK (global rank: {rank}/{world_size})")

    dist.barrier()
    if rank == 0:
```

```
        printflock(f"pt={torch.__version__}, cuda={torch.version.cuda}, nccl={torch.cuda.nccl.version()}")
        printflock(f"device compute capabilities={torch.cuda.get_device_capability()}")
        printflock(f"pytorch compute capabilities={torch.cuda.get_arch_list()}")

except Exception:
    printflock(f"{gpu} is broken")
    raise
```

Run:

```
$ deepspeed -H hostfile test2.py
[2022-09-08 12:07:09,336] [INFO] [runner.py:415:main] Using IP address of 192.168.0.17 for node worker-0
[2022-09-08 12:07:09,337] [INFO] [multinode_runner.py:65:get_cmd] Running on the following workers:
worker-0,worker-1
[2022-09-08 12:07:09,337] [INFO] [runner.py:504:main] cmd = pdsh -S -f 1024 -w worker-0,worker-1 export
PYTHONPATH=/mnt/nvme0/code/huggingface/multi-node-emulate-ds;  cd /mnt/nvme0/code/huggingface/
multi-node-emulate-ds; /home/stas/anaconda3/envs/py38-pt112/bin/python -u -m deepspeed.launcher.launch
--world_info=eyJ3b3JrZXItMCI6IFswXSwgIndvcmtlci0xIjogWzBdfQ== --node_rank=%n --master_addr=192.168.0.17
--master_port=29500 test2.py
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0': [0],
'worker-1': [0]}
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1, node_rank=0
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:155:main] global_rank_mapping=defaultdict(<class
'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:156:main] dist_world_size=2
worker-0: [2022-09-08 12:07:10,635] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:136:main] WORLD INFO DICT: {'worker-0': [0],
'worker-1': [0]}
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:142:main] nnodes=2, num_local_procs=1, node_rank=1
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:155:main] global_rank_mapping=defaultdict(<class
'list'>, {'worker-0': [0], 'worker-1': [1]})
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:156:main] dist_world_size=2
worker-1: [2022-09-08 12:07:10,635] [INFO] [launch.py:158:main] Setting CUDA_VISIBLE_DEVICES=0
worker-0: dist.get_rank()=0, local_rank=0
worker-1: dist.get_rank()=1, local_rank=0
worker-0: [hope-0] is OK (global rank: 0/2)
worker-1: [hope-0] is OK (global rank: 1/2)
worker-0: pt=1.12.1+cu116, cuda=11.6, nccl=(2, 10, 3)
worker-0: device compute capabilities=(8, 0)
worker-0: pytorch compute capabilities=['sm_37', 'sm_50', 'sm_60', 'sm_70', 'sm_75', 'sm_80', 'sm_86']
worker-1: [2022-09-08 12:07:13,642] [INFO] [launch.py:318:main] Process 576485 exits successfully.
worker-0: [2022-09-08 12:07:13,642] [INFO] [launch.py:318:main] Process 576484 exits successfully.
```

Voila, missing accomplished.

We tested that the NCCL collectives work, but they use local NVLink/PCIe and not the IB/ETH connections like in real multi-node, so it may or may not be good enough for testing depending on what needs to be tested.

Larger set ups

Now, let's say you have 4 GPUs and you want to emulate 2x2 nodes. Then simply change the `hostfile` to be:

```
$ cat hostfile
worker-0 slots=2
worker-1 slots=2
```

and the `CUDA_VISIBLE_DEVICES` hack to:

```
if os.environ["RANK"] in ["2", "3"]:
    os.environ["CUDA_VISIBLE_DEVICES"] = "2,3"
```

Everything else should be the same.

Automating the process

If you want an automatic approach to handle any shape of topology, you could use something like this:

```
def set_cuda_visible_devices():
    """
    automatically assign the correct groups of gpus for each emulated node by tweaking the
    CUDA_VISIBLE_DEVICES env var
    """

    global_rank = int(os.environ["RANK"])
    world_size = int(os.environ["WORLD_SIZE"])
    emulated_node_size = int(os.environ["LOCAL_SIZE"])
    emulated_node_rank = int(global_rank // emulated_node_size)
    gpus = list(map(str, range(world_size)))
    emulated_node_gpus =
",".join(gpus[emulated_node_rank*emulated_node_size:(emulated_node_rank+1)*emulated_node_size])
    print(f"Setting CUDA_VISIBLE_DEVICES={emulated_node_gpus}")
    os.environ["CUDA_VISIBLE_DEVICES"] = emulated_node_gpus

set_cuda_visible_devices()
```

Emulating multiple GPUs with a single GPU

The following is an orthogonal need to the one discussed in this document, but it's related so I thought it'd be useful to share some insights here:

With NVIDIA A100 you can use MIG to emulate up to 7 instances of GPUs on just one real GPU, but alas you can't use those instances for anything but standalone use - e.g. you can't do DDP or any NCCL comms over those GPUs. I hoped I could use my A100 to emulate 7 instances and add one more real GPU and to have 8x GPUs to do development with - but nope it doesn't work. Asking NVIDIA engineers about it, there are no plans to have this use-case supported.

Acknowledgements

Many thanks to Jeff Rasley for helping me to set this up.

Model Parallelism

Parallelism overview

In the modern machine learning the various approaches to parallelism are used to:

1. Overcome GPU memory limitations. Examples:
   - fit very large models - e.g., t5-11b is 45GB in just model params
   - fit very long sequences - e.g.,
2. significantly speed up training - finish training that would take a year in hours

We will first discuss in depth various 1D parallelism techniques and their pros and cons and then look at how they can be combined into 2D and 3D parallelism to enable an even faster training and to support even bigger models. Various other powerful alternative approaches will be presented.

While the main concepts most likely will apply to any other framework, this article is focused on PyTorch-based implementations.

Two main approaches are used to enable training and inferring models that are bigger than the accelerator's memory:

1. 3D parallelism - very network efficient, but can be very invasive into the modeling code and require a lot more work to make it work correctly
2. ZeRO parallelism - not very network efficient, but requires close to zero changes to the modeling code and very easy to make to work.

Scalability concepts

The following is the brief description of the main concepts that will be described later in depth in this document.

1. Data Parallelism (DP) - the same setup is replicated multiple times, and each being fed a slice of the data. The processing is done in parallel and all setups are synchronized at the end of each training step.
2. TensorParallelism (TP) - each tensor is split up into multiple chunks, so instead of having the whole tensor reside on a single gpu, each shard of the tensor resides on its designated gpu. During processing each shard gets processed separately and in parallel on different GPUs and the results are synced at the end of the step. This is what one may call horizontal parallelism, as the splitting happens on horizontal level.
3. PipelineParallelism (PP) - the model is split up vertically (layer-level) across multiple GPUs, so that only one or several layers of the model are places on a single gpu. Each gpu processes in parallel different stages of the pipeline and working on a small chunk of the batch.
4. Zero Redundancy Optimizer (ZeRO) - Also performs sharding of the tensors somewhat similar to TP, except the whole tensor gets reconstructed in time for a forward or backward computation, therefore the model doesn't need to be modified. It also supports various offloading techniques to compensate for limited GPU memory. Sharded DDP is another name for the foundational ZeRO concept as used by various other implementations of ZeRO.
5. Sequence Parallelism - training on long input sequences requires huge amounts of GPU memory. This technique splits the processing of a single sequence across multiple GPUs.

The introduction sections of this paper is probably one of the best explanations I have found on most common parallelism techniques Breadth-First Pipeline Parallelism.

Data Parallelism

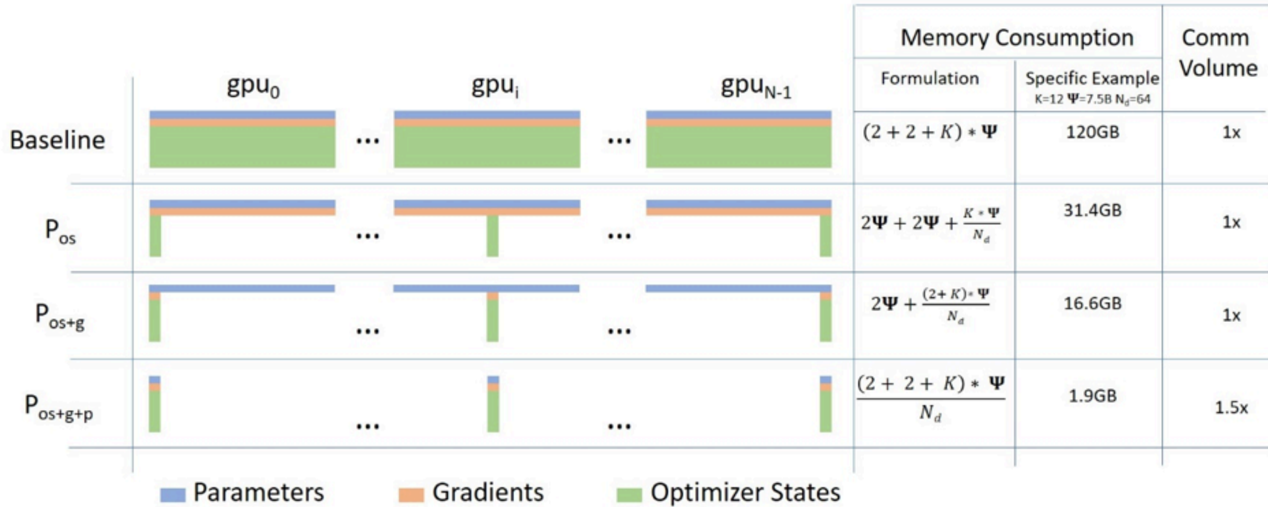
DDP

Most users with just 2 GPUs already enjoy the increased training speed up thanks to `DataParallel` (DP) and `DistributedDataParallel` (DDP) that are almost trivial to use. This is a built-in feature of Pytorch.

For details see DistributedDataParallel

ZeRO Data Parallelism

ZeRO-powered data parallelism (ZeRO-DP) is described on the following diagram from this blog post



It can be difficult to wrap one's head around it, but in reality the concept is quite simple. This is just the usual `DataParallel` (DP), except, instead of replicating the full model params, gradients and optimizer states, each GPU stores only a slice of it. And then at run-time when the full layer params are needed just for the given layer, all GPUs synchronize to give each other parts that they miss - this is it.

Consider this simple model with 3 layers, where each layer has 3 params:

```
La | Lb | Lc
---|----|---
a0 | b0 | c0
a1 | b1 | c1
a2 | b2 | c2
```

Layer La has weights a0, a1 and a2.

If we have 3 GPUs, the Sharded DDP (= Zero-DP) splits the model onto 3 GPUs like so:

```
GPU0:
La | Lb | Lc
---|----|---
a0 | b0 | c0

GPU1:
```

```
La | Lb | Lc
---|----|---
a1 | b1 | c1

GPU2:
La | Lb | Lc
---|----|---
a2 | b2 | c2
```

In a way this is the same horizontal slicing, as tensor parallelism, if you imagine the typical DNN diagram. Vertical slicing is where one puts whole layer-groups on different GPUs. But it's just the starting point.

Now each of these GPUs will get the usual mini-batch as it works in DP:

```
x0 => GPU0
x1 => GPU1
x2 => GPU2
```

The inputs are unmodified - they think they are going to be processed by the normal model.

First, the inputs hit the layer La.

Let's focus just on GPU0: x0 needs a0, a1, a2 params to do its forward path, but GPU0 has only a0 - it gets sent a1 from GPU1 and a2 from GPU2, bringing all pieces of the model together.

In parallel, GPU1 gets mini-batch x1 and it only has a1, but needs a0 and a2 params, so it gets those from GPU0 and GPU2.

Same happens to GPU2 that gets input x2. It gets a0 and a1 from GPU0 and GPU1, and with its a2 it reconstructs the full tensor.

All 3 GPUs get the full tensors reconstructed and a forward happens.

As soon as the calculation is done, the data that is no longer needed gets dropped - it's only used during the calculation. The reconstruction is done efficiently via a pre-fetch.

And the whole process is repeated for layer Lb, then Lc forward-wise, and then backward Lc -> Lb -> La.

To me this sounds like an efficient group backpacking weight distribution strategy:

1. person A carries the tent
2. person B carries the stove
3. person C carries the axe

Now each night they all share what they have with others and get from others what they don't have, and in the morning they pack up their allocated type of gear and continue on their way. This is Sharded DDP / Zero DP.

Compare this strategy to the simple one where each person has to carry their own tent, stove and axe, which would be far more inefficient. This is DataParallel (DP and DDP) in Pytorch.

While reading the literature on this topic you may encounter the following synonyms: Sharded, Partitioned.

If you pay close attention the way ZeRO partitions the model's weights - it looks very similar to tensor parallelism which will be discussed later. This is because it partitions/shards each layer's weights, unlike vertical model parallelism which is discussed next.

Implementations of ZeRO-DP stages 1+2+3:

- DeepSpeed

- PyTorch (originally it was implemented in FairScale and later it was upstreamed into the PyTorch core)

Deepspeed ZeRO Integration:

- HF Trainer integration
- Accelerate
- PyTorch Lightning
- Determined.AI

FSDP Integration:

- HF Trainer integration
- Accelerate
- PyTorch Lightning

Important papers:

Deepspeed:

- ZeRO: Memory Optimizations Toward Training Trillion Parameter Models
- ZeRO-Offload: Democratizing Billion-Scale Model Training
- ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning
- ZeRO++: Extremely Efficient Collective Communication for Giant Model Training
- DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models

PyTorch:

- PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel

Main DeepSpeed ZeRO Resources:

- Project's github
- Usage docs
- API docs
- Blog posts

ZeRO with multiple replicas

By default ZeRO uses all GPUs to create a single model replica - that's the model is spread out across all gpus. Which leads to various limitations such as:

1. the global batch size is inflexible - it's always a function of `total_gpus*micro_batch_size` - which on large clusters could lead to a huge global batch size which might be detrimental for efficient convergence. Granted one could use a tiny micro batch size to keep the global batch size in check, but this leads to smaller matrices on each GPU which results in less efficient compute
2. the much faster intra-node networking is not being benefited from since the slower inter-node network defines the overall speed of communications.

ZeRO++ solves the 2nd limitation by introducing Hierarchical Weight Partition for ZeRO (hpZ). In this approach instead of spreading whole model weights across all the gpus, each model replica is restricted to a single node. This increases the memory usage by the total number of nodes, but now the 2x `all_gather` calls to gather the sharded components are performed over a much faster intra-node connection. Only the `reduce_scatter` to aggregate and redistribute gradients is performed over the slower inter-node network.

The first limitation doesn't exactly get fixed since the overall global batch size remains the same, but since each replica is more efficient and because the additional memory pressure is likely to limit the possible micro batch size on each gpu, this overall should improve the throughput of the system.

PyTorch FSDP has this feature implemented in shardingStrategy.HYBRID_SHARD

Papers:

- [ZeRO++: Extremely Efficient Collective Communication for Giant Model Training]
- [PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel]

ZeRO variations

Published papers that propose modifications to the ZeRO protocol:

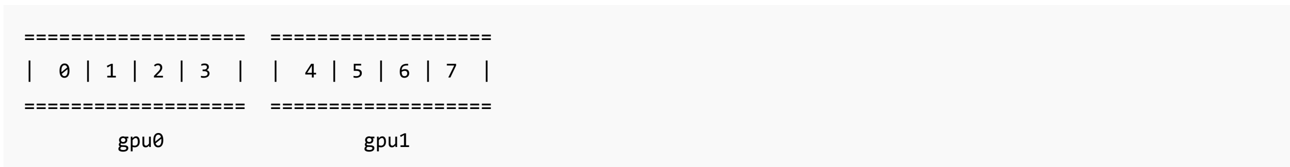
- [MiCS: Near-linear Scaling for Training Gigantic Model on Public Cloud] (2022)
- [AMSP: Super-Scaling LLM Training via Advanced Model States Partitioning] (2023)

Pipeline Parallelism methods

Naive Model Parallelism (Vertical)

Naive Model Parallelism (MP) is where one spreads groups of model layers across multiple GPUs. The mechanism is relatively simple - switch the desired layers `.to()` the desired devices and now whenever the data goes in and out those layers switch the data to the same device as the layer and leave the rest unmodified.

We refer to it as Vertical MP, because if you remember how most models are drawn, we slice the layers vertically. For example, if the following diagram shows an 8-layer model:



```
===================  ===================
|  0 | 1 | 2 | 3  |  |  4 | 5 | 6 | 7  |
===================  ===================
        gpu0                 gpu1
```

we just sliced it in 2 vertically, placing layers 0-3 onto GPU0 and 4-7 to GPU1.

Now while data travels from layer 0 to 1, 1 to 2 and 2 to 3 this is just the normal model. But when data needs to pass from layer 3 to layer 4 it needs to travel from GPU0 to GPU1 which introduces a communication overhead. If the participating GPUs are on the same compute node (e.g. same physical machine) this copying is pretty fast, but if the GPUs are located on different compute nodes (e.g. multiple machines) the communication overhead could be significantly larger.

Then layers 4 to 5 to 6 to 7 are as a normal model would have and when the 7th layer completes we often need to send the data back to layer 0 where the labels are (or alternatively send the labels to the last layer). Now the loss can be computed and the optimizer can do its work.

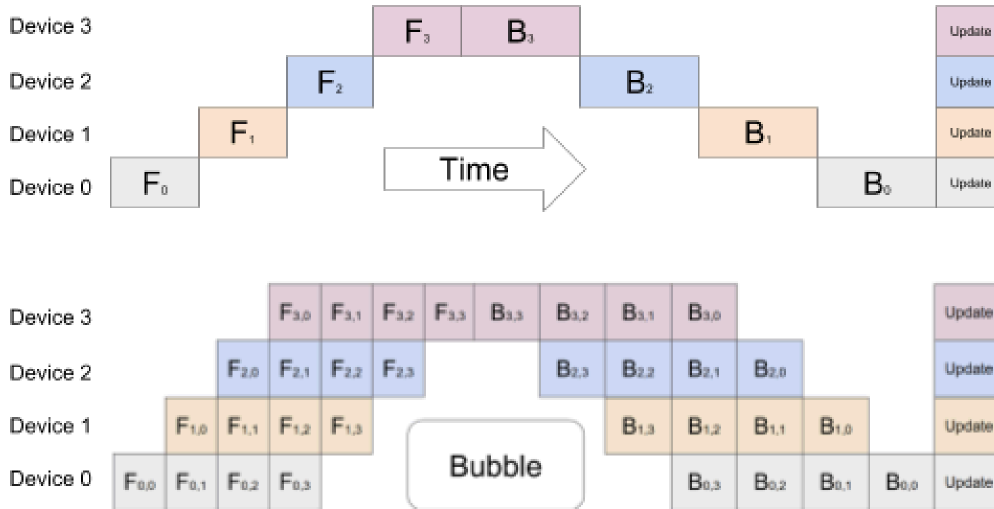
Problems:

- the main deficiency and why this one is called "naive" MP, is that all but one GPU is idle at any given moment. So if 4 GPUs are used, it's almost identical to quadrupling the amount of memory of a single GPU, and ignoring the rest of the hardware. Plus there is the overhead of copying the data between devices. So 4x 6GB cards will be able to accommodate the same size as 1x 24GB card using naive MP, except the latter will complete the training faster, since it doesn't have the data copying overhead. But, say, if you have 40GB cards and need to fit a 45GB model you can with 4x 40GB cards (but barely because of the gradient and optimizer states)
- shared embeddings may need to get copied back and forth between GPUs.

Pipeline Parallelism

Pipeline Parallelism (PP) is almost identical to a naive MP, but it solves the GPU idling problem, by chunking the incoming batch into micro-batches and artificially creating a pipeline, which allows different GPUs to concurrently participate in the computation process.

The following illustration from the [GPipe paper] shows the naive MP on the top, and PP on the bottom:



*Top: The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. **Bottom**: GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.*

It’s easy to see from the bottom diagram how PP has less dead zones, where GPUs are idle. The idle parts are referred to as the "bubble".

Both parts of the diagram show a parallelism that is of degree 4. That is 4 GPUs are participating in the pipeline. So there is the forward path of 4 pipe stages F0, F1, F2 and F3 and then the return reverse order backward path of B3, B2, B1 and B0.

PP introduces a new hyper-parameter to tune and it’s `chunks` which defines how many chunks of data are sent in a sequence through the same pipe stage. For example, in the bottom diagram you can see that `chunks=4`. GPU0 performs the same forward path on chunk 0, 1, 2 and 3 (F0,0, F0,1, F0,2, F0,3) and then it waits for other GPUs to do their work and only when their work is starting to be complete, GPU0 starts to work again doing the backward path for chunks 3, 2, 1 and 0 (B0,3, B0,2, B0,1, B0,0).

Note that conceptually this is the same concept as gradient accumulation steps (GAS). Pytorch uses `chunks`, whereas DeepSpeed refers to the same hyper-parameter as GAS.

Because of the chunks, PP introduces the concept of micro-batches (MBS). DP splits the global data batch size into mini-batches, so if you have a DP degree of 4, a global batch size of 1024 gets split up into 4 mini-batches of 256 each (1024/4). And if the number of `chunks` (or GAS) is 32 we end up with a micro-batch size of 8 (256/32). Each Pipeline stage works with a single micro-batch at a time.

To calculate the global batch size of the DP + PP setup we then do: `mbs*chunks*dp_degree` (`8*32*4=1024`).

Let’s go back to the diagram.

With `chunks=1` you end up with the naive MP, which is very inefficient. With a very large `chunks` value you end up with tiny micro-batch sizes which could be not every efficient either. So one has to experiment to find the value that leads to the highest efficient utilization of the gpus.

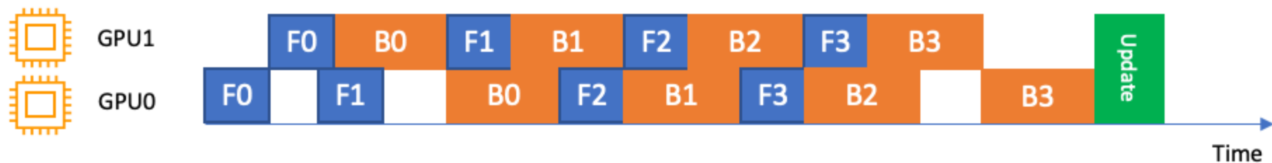
While the diagram shows that there is a bubble of "dead" time that can’t be parallelized because the last `forward` stage has to wait for `backward` to complete the pipeline, the purpose of finding the best value for `chunks` is to enable a high concurrent

GPU utilization across all participating GPUs which translates to minimizing the size of the bubble.

The choice of the schedule is critical to the efficient performance, with the most common schedules being in the order of invention:

- sequential Gpipe: Efficient training of giant neural networks using pipeline parallelism
- interleaved 1F1B Pipedream: Fast and efficient pipeline parallel dnn training
- looped, depth-first Efficient large-scale language model training on gpu clusters using Megatron-LM
- breadth-first Breadth-First Pipeline Parallelism

Here is for example an interleaved pipeline:



Here the bubble (idle time) is further minimized by prioritizing backward passes.

It's used by DeepSpeed, Varuna and SageMaker to name a few.

Varuna further tries to improve the schedule by using simulations to discover the most efficient scheduling.

There are 2 groups of PP solutions - the traditional Pipeline API and the more modern solutions that make things much easier for the end user by helping to partially or fully automate the process:

1. Traditional Pipeline API solutions:

- Megatron-LM
- DeepSpeed
- PyTorch

2. Modern solutions:

- PiPPy
- Varuna
- Sagemaker

Problems with traditional Pipeline API solutions:

- have to modify the model quite heavily, because Pipeline requires one to rewrite the normal flow of modules into a `nn.Sequential` sequence of the same, which may require changes to the design of the model.
- currently the Pipeline API is very restricted. If you had a bunch of python variables being passed in the very first stage of the Pipeline, you will have to find a way around it. Currently, the pipeline interface requires either a single Tensor or a tuple of Tensors as the only input and output. These tensors must have a batch size as the very first dimension, since pipeline is going to chunk the mini batch into micro-batches. Possible improvements are being discussed here https://github.com/pytorch/pytorch/pull/50693
- conditional control flow at the level of pipe stages is not possible - e.g., Encoder-Decoder models like T5 require special workarounds to handle a conditional encoder stage.
- have to arrange each layer so that the output of one model becomes an input to the other model.

I'm yet to try to experiment with Varuna and SageMaker but their papers report that they have overcome the list of problems mentioned above and that they require much smaller changes to the user's model.

Implementations:

- Pytorch (initial support in pytorch-1.8, and progressively getting improved in 1.9 and more so in 1.10). Some examples
- FairScale

- [DeepSpeed](#)
- [Megatron-LM](#) has an internal implementation - no API.
- [Varuna](#)
- [SageMaker](#) - this is a proprietary solution that can only be used on AWS.
- [OSLO](#) - this is implemented based on the Hugging Face Transformers.
- [PiPPy: Pipeline Parallelism for PyTorch](#) - automatic PP via `torch.fx`
- [nanotron](#)

Tensor Parallelism

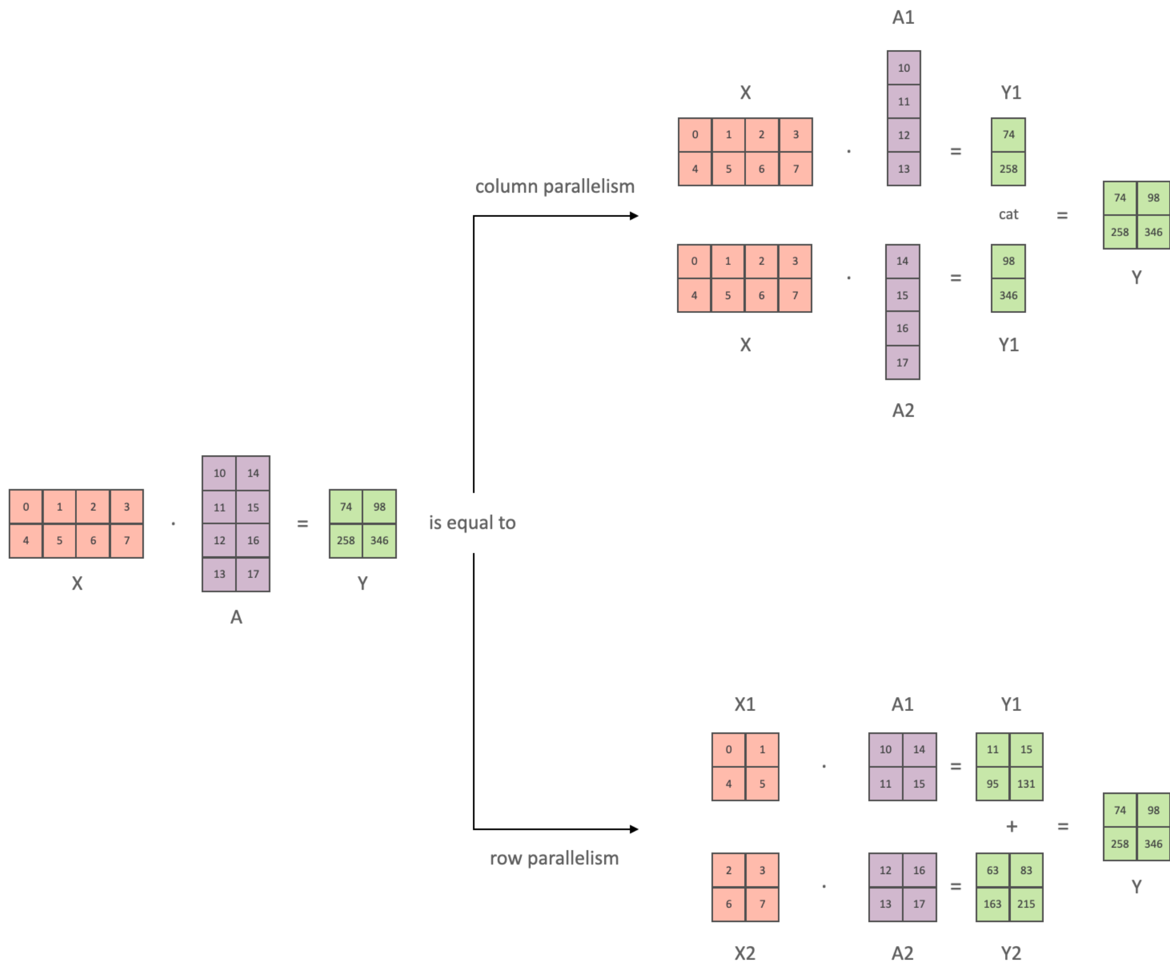
In Tensor Parallelism each GPU processes only a slice of a tensor and only aggregates the full tensor for operations that require the whole thing.

In this section we use concepts and diagrams from the [Megatron-LM](#) paper: [Efficient Large-Scale Language Model Training on GPU Clusters](#).

The main building block of any transformer is a fully connected `nn.Linear` followed by a nonlinear activation `GeLU`.

Following the Megatron's paper notation, we can write the dot-product part of it as `Y = GeLU(XA)`, where `X` and `Y` are the input and output vectors, and `A` is the weight matrix.

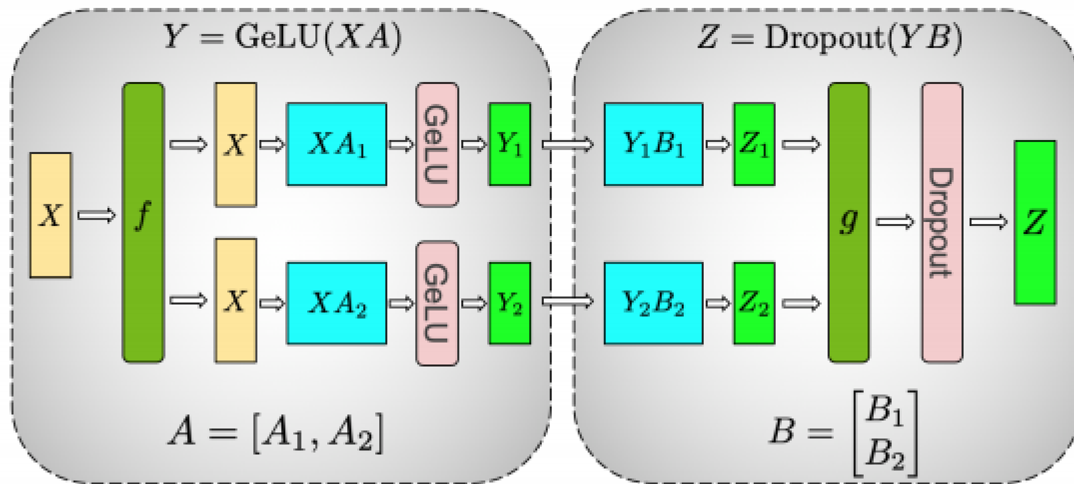
If we look at the computation in matrix form, it's easy to see how the matrix multiplication can be split between multiple GPUs:



If we split the weight matrix `A` column-wise across `N` GPUs and perform matrix multiplications `XA_1` through `XA_n` in parallel, then we will end up with `N` output vectors `Y_1, Y_2, ..., Y_n` which can be fed into `GeLU` independently:

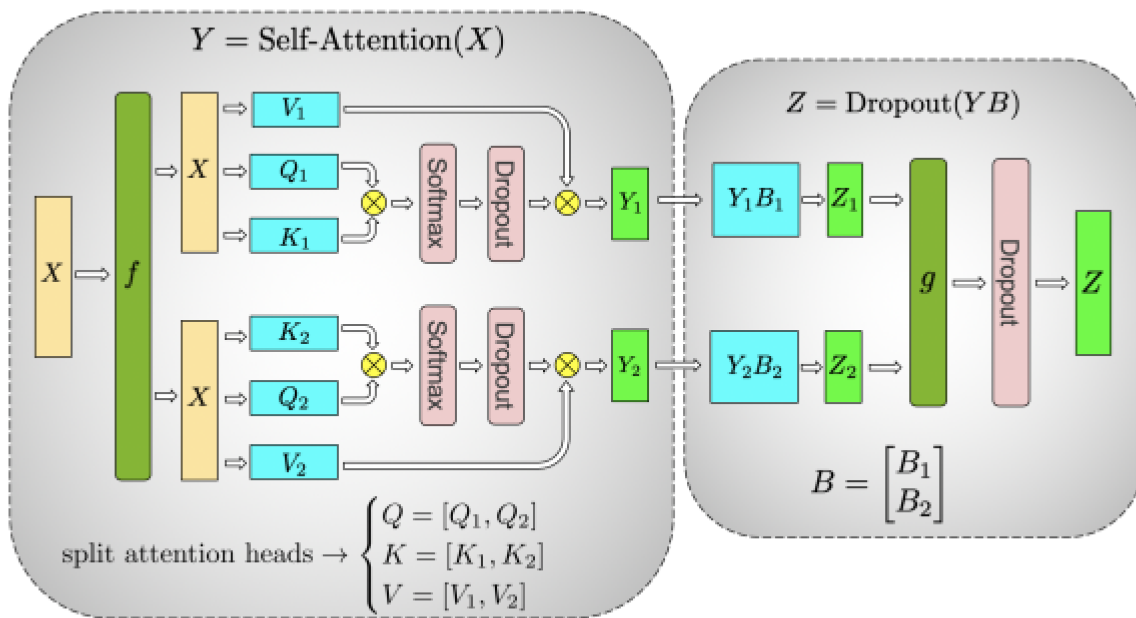
$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)]$$

Using this principle, we can update an MLP of arbitrary depth, without the need for any synchronization between GPUs until the very end, where we need to reconstruct the output vector from shards. The Megatron-LM paper authors provide a helpful illustration for that:



(a) MLP

Parallelizing the multi-headed attention layers is even simpler, since they are already inherently parallel, due to having multiple independent heads!



(b) Self-Attention

Important: TP requires very fast network, and therefore since typically intra-node networks are much faster than inter-node networks it's not advisable to do TP across nodes. Practically, if a node has 4 GPUs, the highest TP degree is therefore 4. If you need a TP degree of 8, you need to use nodes that have at least 8 GPUs.

Important: TP degree shouldn't span across nodes. For example if the node has 8 gpus, TP degree should be no more than 8.

TP can combined with other parallelization methods.

Alternative names:

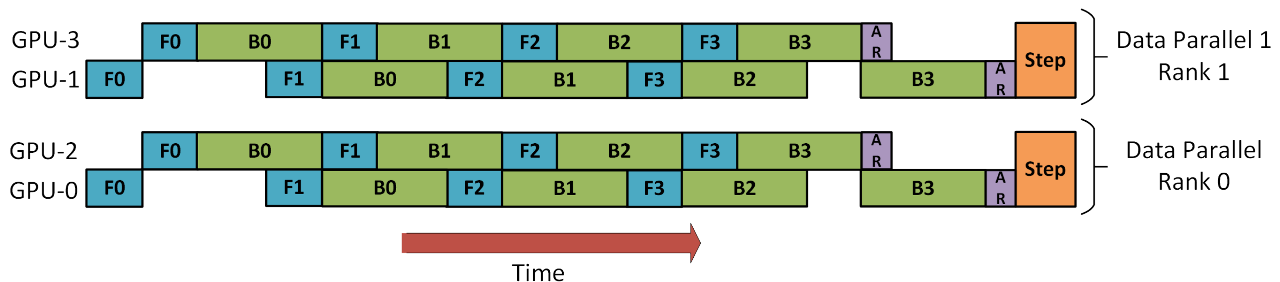
- DeepSpeed calls it tensor slicing

Implementations:

- Megatron-LM has an internal implementation, as it's very model-specific
- PyTorch
- SageMaker - this is a proprietary solution that can only be used on AWS.
- OSLO has the tensor parallelism implementation based on the Transformers.
- nanotron
- parallelformers (only inference at the moment)

DP+PP

The following diagram from the DeepSpeed pipeline tutorial demonstrates how one combines DP with PP.



Here it's important to see how DP rank 0 doesn't see GPU2 and DP rank 1 doesn't see GPU3. To DP there is just GPUs 0 and 1 where it feeds data as if there were just 2 GPUs. GPU0 "secretly" offloads some of its load to GPU2 using PP. And GPU1 does the same by enlisting GPU3 to its aid.

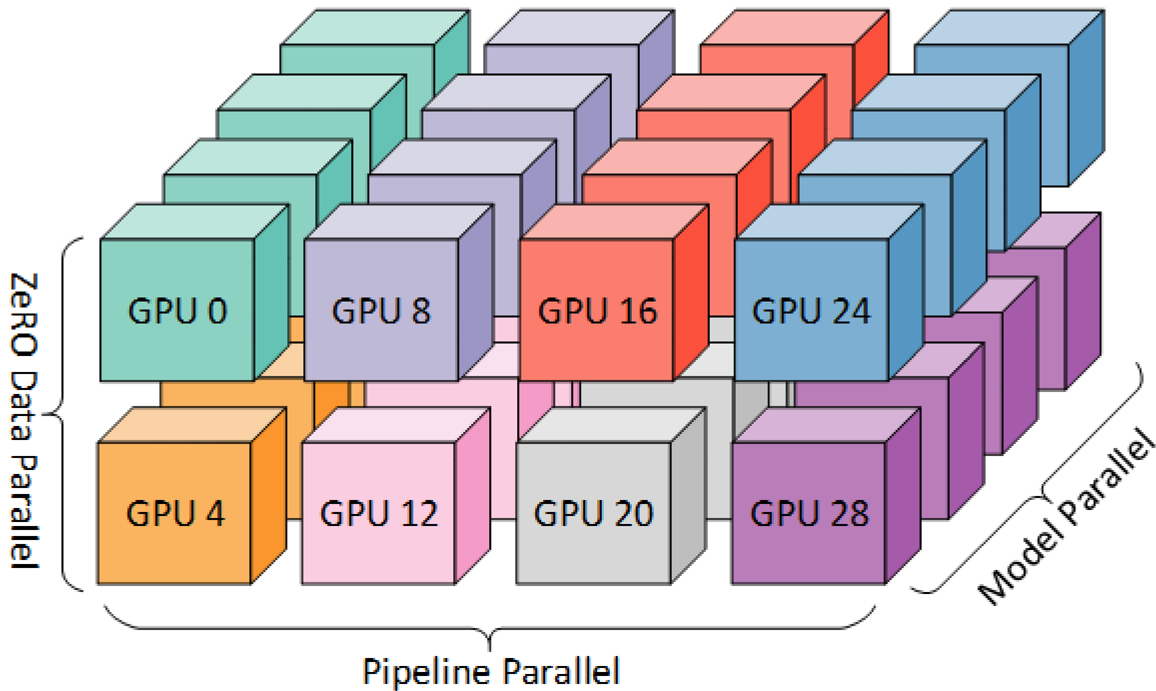
Since each dimension requires at least 2 GPUs, here you'd need at least 4 GPUs.

Implementations:

- DeepSpeed
- Megatron-LM
- Varuna
- SageMaker
- OSLO
- nanotron

DP+PP+TP

To get an even more efficient training a 3D parallelism is used where PP is combined with TP and DP. This can be seen in the following diagram.



This diagram is from a blog post 3D parallelism: Scaling to trillion-parameter models, which is a good read as well.

Since each dimension requires at least 2 GPUs, here you'd need at least 8 GPUs.

Implementations:

- DeepSpeed - DeepSpeed also includes an even more efficient DP, which they call ZeRO-DP.
- Megatron-LM
- Varuna
- SageMaker
- OSLO
- nanotron

ZeRO DP+PP+TP

One of the main features of DeepSpeed is ZeRO, which is a super-scalable extension of DP. It has already been discussed in ZeRO Data Parallelism. Normally it's a standalone feature that doesn't require PP or TP. But it can be combined with PP and TP.

When ZeRO-DP is combined with PP (and optionally TP) it typically enables only ZeRO stage 1 (optimizer sharding).

While it's theoretically possible to use ZeRO stage 2 (gradient sharding) with Pipeline Parallelism, it will have bad performance impacts. There would need to be an additional reduce-scatter collective for every micro-batch to aggregate the gradients before sharding, which adds a potentially significant communication overhead. By nature of Pipeline Parallelism, small micro-batches are used and instead the focus is on trying to balance arithmetic intensity (micro-batch size) with minimizing the Pipeline bubble (number of micro-batches). Therefore those communication costs are going to hurt.

In addition, there are already fewer layers than normal due to PP and so the memory savings won't be huge. PP already reduces gradient size by `1/PP`, and so gradient sharding savings on top of that are less significant than pure DP.

ZeRO stage 3 is not a good choice either for the same reason - more inter-node communications required.

And since we have ZeRO, the other benefit is ZeRO-Offload. Since this is stage 1 optimizer states can be offloaded to CPU.

Implementations:

- Megatron-DeepSpeed and Megatron-Deepspeed from BigScience, which is the fork of the former repo.
- OSLO

Important papers:

- Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model

Sequence Parallelism

ML tasks, such as DNA sequencing, may require training with very long sequence lengths (e.g. 256K), and even normal LLMs could be trained on sequences of 10k and longer.

Self-Attention, which is the key component of Transformers, suffers from quadratic memory requirements with respect to the sequence length, therefore when sequence length gets to a certain length, even a batch size of 1 might not be able to fit onto a single GPU and require additional partitioning along the sequence dimension. And once this is done, the sequence can be of any length.

As this type of parallelism is orthogonal to the other parallelization types described in this document, it can be combined with any of them, leading to 4D, ZeRO-DP+SP and other combinations.

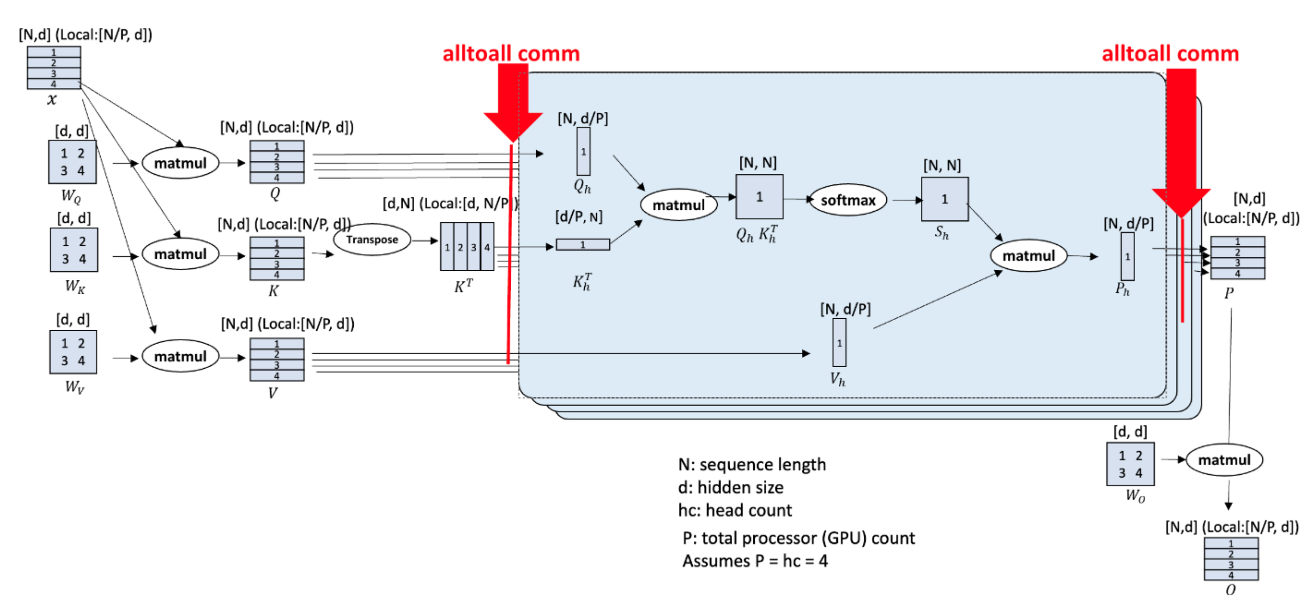
Deepspeed-Ulysses SP

Paper: DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models

In this implementation 2 elements are sharded:

1. The multiple-head attention weights are split across the participating GPUs so that each GPU has a few sub-heads only. This is done when the model is created/loaded. This is somewhat similar to Tensor Parallelism.
2. During training each input sequence is partitioned into chunks and each chunk is sent to one of the GPUs, which reminds us of ZeRO-3 sharding, except instead of weights the inputs are sharded.

During compute each sequence chunk is projected onto QKV and then gathered to the full sequence QKV on each device, computed on each device only for the subheads it owns and then gathered again into the full attention output for the MLP block.



source

On the diagram:

1. Input sequences N are partitioned across P available devices.
2. Each local N/P partition of the input sequence is projected into queries (Q), keys (K) and values (V) embeddings.
3. Next, local QKV embeddings are gathered into global QKV through highly optimized all-to-all collectives between participating compute devices.
4. Then the attention computation per head is performed:

$$Output\ context = Softmax\ (\frac{QK^T}{\sqrt{d}})V$$

5. At the end another all-to-all collective transforms output context tensor of attention computation to sequence (N/P) parallel for subsequent operators (MLP MatMul, layer norm, etc.) in the remaining modules of transformer layer block.

Example: Let's consider seqlen=8K, num_heads=128 and a single node of num_gpus=8

1. each GPU gets a 1K-long chunk of the original sequence (`8K/8`)
2. each GPU gets assigned 16 sub-heads (`128/8`)
3. a. on gpu0 before `forward` the original sequence is gathered back into 8K tokens b. the attention computation is done on the first 16 sub-heads the same logic is performed on the remaining 7 GPUs, each computing 8k attention over its 16 sub-heads

You can read the specifics of the very efficient comms here.

DeepSpeed-Ulysses keeps communication volume consistent by increasing GPUs proportional to message size or sequence length.

Colossal-AI's SP

Paper: Sequence parallelism: Long sequence training from system perspective

Colossal-AI's SP implementation uses ring self-attention, a ring-like communication collective in which query projections are local whereas key and values projections are transmitted in a ring-style to compute global attention, resulting in communication complexity linear in message size, M.

Megatron-LM's SP

Paper: Reducing Activation Recomputation in Large Transformer Models

Megatron-LM's SP is tightly integrated with its TP. Megatron-LM partitions sequence along sequence dimensions and applies allgather and reduce scatter collective to aggregate QKV projections for attention computation. Its communication volume increases linearly with message size (M) regardless of number of compute devices.

Implementations:

- Megatron-LM
- Deepspeed
- Colossal-AI

FlexFlow

FlexFlow also solves the parallelization problem in a slightly different approach.

Paper: "Beyond Data and Model Parallelism for Deep Neural Networks" by Zhihao Jia, Matei Zaharia, Alex Aiken

It performs a sort of 4D Parallelism over Sample-Operator-Attribute-Parameter.

1. Sample = Data Parallelism (sample-wise parallel)
2. Operator = Parallelize a single operation into several sub-operations
3. Attribute = Data Parallelism (length-wise parallel)
4. Parameter = Model Parallelism (regardless of dimension - horizontal or vertical)

Examples:

- Sample

Let's take 10 batches of sequence length 512. If we parallelize them by sample dimension into 2 devices, we get 10 x 512 which becomes be 5 x 2 x 512.

- Operator

If we perform layer normalization, we compute std first and mean second, and then we can normalize data. Operator parallelism allows computing std and mean in parallel. So if we parallelize them by operator dimension into 2 devices (cuda:0, cuda:1), first we copy input data into both devices, and cuda:0 computes std, cuda:1 computes mean at the same time.

- Attribute

We have 10 batches of 512 length. If we parallelize them by attribute dimension into 2 devices, 10 x 512 will be 10 x 2 x 256.

- Parameter

It is similar with tensor model parallelism or naive layer-wise model parallelism.

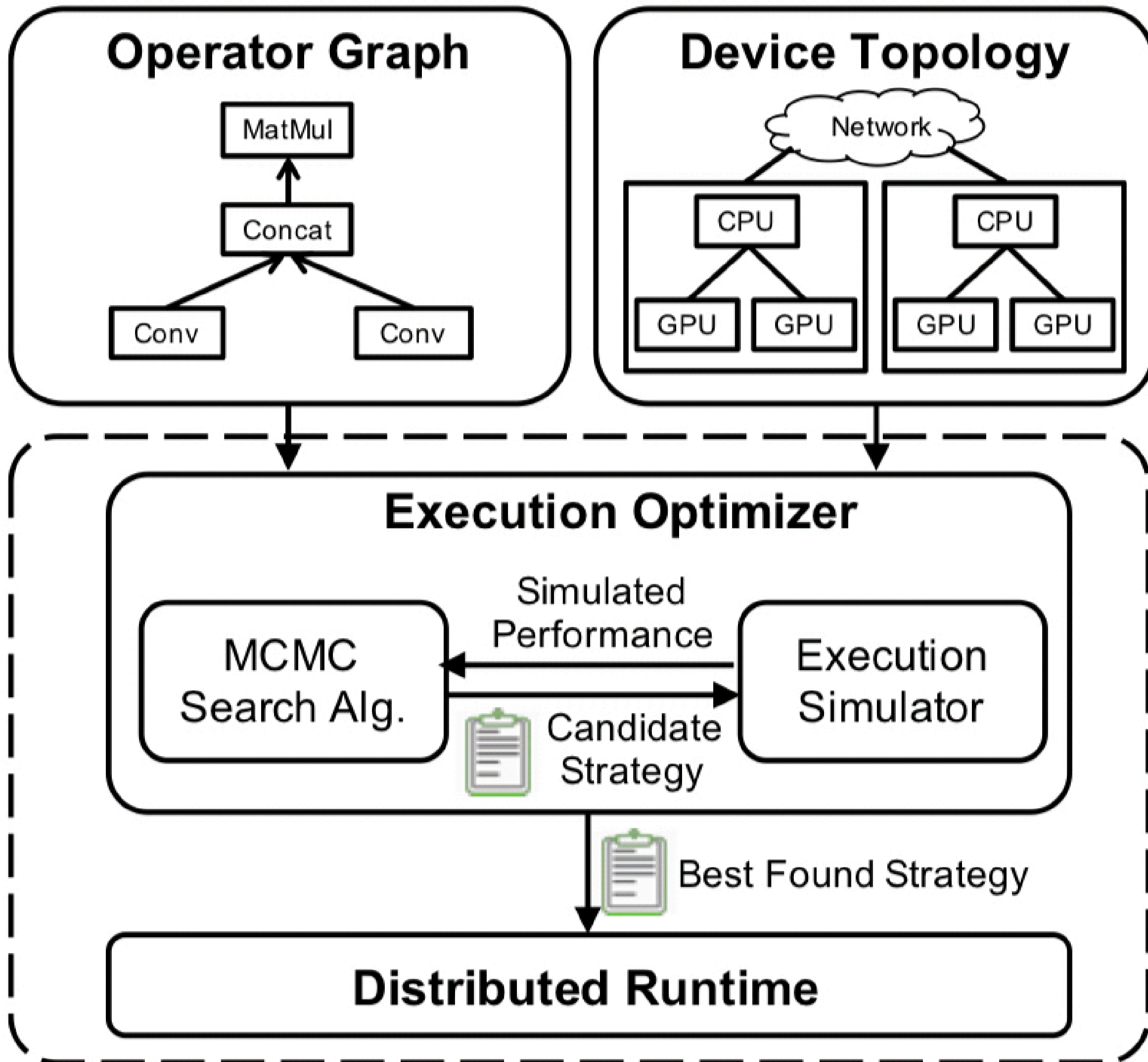


Figure 1. FlexFlow overview.

The significance of this framework is that it takes resources like (1) GPU/TPU/CPU vs. (2) RAM/DRAM vs. (3) fast-intra-connect/slow-inter-connect and it automatically optimizes all these algorithmically deciding which parallelisation to use where.

One very important aspect is that FlexFlow is designed for optimizing DNN parallelizations for models with static and fixed workloads, since models with dynamic behavior may prefer different parallelization strategies across iterations.

So the promise is very attractive - it runs a 30min simulation on the cluster of choice and it comes up with the best strategy to utilise this specific environment. If you add/remove/replace any parts it'll run and re-optimize the plan for that. And then you can train. A different setup will have its own custom optimization.

Inter-node speed requirements to use ZeRO

The ZeRO scalability protocol, be it Deepspeed ZeRO or PyTorch FSDP, requires a lot more inter-node traffic than TP+PP+DP solutions, and sometimes it can't take advantage of the faster intra-node connectivity, and therefore if your inter-node network is slow your expensive GPUs might be massively bottlenecked by the comms.

The ZeRO protocol partially overlaps comms with compute, so ideally you want to get close to `comms_time <= compute_time`. The overlap is not perfect, so there will be always some network bottleneck, but we want to make sure that `comms_time` is not much larger than `compute_time`.

In ZeRO-3, we have `all_gather` on weights in `forward`, then `all_gather` on weights in `backward`, last is `reduce_scatter` on gradients in backward. In total there are 3 global collective calls each sending a model size multiplied by how many bytes per parameter are used. e.g. a 10B param model in bf16 under ZeRO-3 will need to send 10*2*3=60GB of data.

In comparison DistributedDataParallel (DDP) uses a single `all_reduce` call, but which requires 2x data transmission, and so a 10B param model in bf16 under DDP will need to send 10*2*2=40GB of data.

ZeRO-1 which only shards the optimiser states, like DDP, will too need to transmit 40GB of data (one `all_gather` and one `reduce_scatter`.)

Here is how to calculate time in seconds for communication and compute:

- `comms_time = n_transmissions * n_bytes * model_size_in_B / inter-node-throughput_in_GBps`
- `compute_time = n_passes * n_bytes * model_size_in_B * seqlen * global_batch_size / (total_gpus * 1e3 * tflops_wo_comms)`

The compute time formula is a rough estimate which works for any Transformer-block based model. It ignores any small computations and includes only the massive `matmul`s.

As an experiment let's use the data points from IDEFICS-80B training.

When we trained IDEFICS-80B with a 340GBs EFA we were getting only 90TFLOPs w/ Deepspeed ZeRO-3 on A100s as compared to 150+TFLOPs one was getting with Megatron's TP+PP+DP. and moreover a big chunk of the model was frozen as were building a new models based on one language and one vision model. So our multiplier was less than 3. On the other hand we were using activation recomputation to save memory, so this is an additional transmission of all model weights and to top it all off since nccl wasn't supporting proper half-precision reduction we used fp32 for gradient reductions, so really our multiplier wasn't 3 but more like 4.5.

Values used for IDEFICS-80B training:

- `model_size_in_B` = `80`
- `n_bytes` = `2` in case of bf16 which is 2 bytes
- `n_transmissions` = `3` in the case of ZeRO-3/FSDP (1x reduce_scatter + 2x all_gather (fwd + bwd)) and `2` in case of ZeRO-1 (1x reduce_scatter + 1x all_gather),
- additionally, in the case of IDEFICS-80B we decided to reduce grads in fp32 to minimize NCCL accumulation loss, so we actually had `n_transmissions*n_bytes=3*2+2=4*2` for the additional `2` bytes but since half the model was frozen only about half of gradients were sent, so we still have the multiplier of `3`.
- `n_passes` = `4` with activation recomputation, or `3` w/o it. The model has to do only 1x compute per `forward` and 2x per `backward` (since the grads are calculated twice - once wrt inputs and once wrt weights). And with activation recomputation one more `forward` is done.
- `total_gpus` = `512`
- `global_batch_size` = `3584`
- `seqlen` = `1024`
- `inter-node-throughput_in_GBps` = 42.5 (340Gbps) (AWS EFA v1) -`tflops_wo_comms` is the tflops w/o the communication overhead. Not theoretical peak as that is unachievable, but perhaps 75% in the case of A100@BF16 - so `312*0.75=234` TFLOPS

We derived 340Gbps inter-node network throughput using all_reduce_bench.py which by default uses a payload of 4GB. In the case of IDEFICS-80B we had 80 layers, so approximately each layer was 1B params large. Which means that each layer was sending 2GB of data for bf16 tensors and 4GB of data with fp32 tensors, which matches the network benchmark. If you were to have a much smaller layer size, I'd recommend adapting the benchmark to that size. For example, if your layer size was only 100M param large, then your payload would be 0.2GB for bf16 tensors. As this is an order of magnitude smaller, the network is likely to give you a lower bandwidth, and you should use that in your calculations.

footnote: if parts of your model are frozen, then there will be less data sent in syncing the gradients. in IDEFICS we had more than half of the model frozen, so when grads were reduced we only had about half the traffic.

Which gives us:

- comms = `3 * 2 * 80 / 42.5` = 11 sec
- compute = `4 * 2 * 80 * 1024 * 3584 / (512 * 1e3 * 250)` = 18 sec

If we check against our IDEFICS-80B logs, which had each iteration at about 49 seconds.

So the good news is that the math checks out as comms + compute are in the ballpark of the measured time, except

We can do another sanity check by feeding the compute formulae 90 TFLOPS that we logged, in which case:

- compute = `4 * 2 * 80 * 1024 * 3584 / (512 * 1e3 * 90)` = 51 sec

and so 49 and 51 secs are pretty close. Except this tells us nothing since the logged TFLOPS were calculated using this formula, so, of course, it should match.

What I'd expect in the best case is where I have used close to theoretical peak TFLOPS in the formula and received the compute estimate to be about the same as the actual compute time measured on the system. Remember that since comms are interleaved with compute, when we measure `forward+backward` wallclock time it includes comms in it.

What's the conclusion? I'd say more investigation is needed as clearly there are additional hidden bottlenecks here. I no longer have access to this setup to investigate, so I will repeat this exercise afresh when I train another largish model and share the updated math with you. But this workout should give you a feeling for what's going on behind the scenes and how all these numbers work together.

Also this discussion didn't include into the math gradient accumulation steps (GAS). In the case of IDEFICS-80B it wasn't used. If GAS>1 the theoretical compute time doesn't change, but comms time instead of `3*2*M/GBps` would become `GAS*3*2*M/GBps`. The weights gathering via `all_gather` for `forward` and `backward` would transpire as many times as there are gradient accumulation steps. In theory for grads it'd need to happen only once, but since there is no place to store intermediary grads of the gathered weight on each GPU it'll have to be reduced GAS times as well. This is for ZeRO-2 and ZeRO-3. For ZeRO-1 GAS>1 requires no additional comms.

We also didn't discuss the `DataLoader` as a potential bottleneck here, but we tested that it was under 1 sec, i.e. a very small overhead.

Going back to comms math, we also didn't take into an account various hardware latencies, but when dealing with a large payloads they shouldn't add up a significant additional overhead.

And now you know how long it'll take to transmit that many GBs over the network of your system. For example, if the network were to be 5x slower than the one we used for IDEFICS-80B training, that is 8.5GBps (68Gbps) then:

- comms = `3 * 2 * 80 / 8.5` = 56 sec

which would definitely be a huge bottleneck compared to the faster compute.

If the network were to be 5x faster, that is 212GBs (1700Gbps) then:

- comms = `3 * 2 * 80 / 212` = 2 sec

which would be insignificant comparatively to the compute time, especially if some of it is successfully overlapped with the commute.

Also the Deepspeed team empirically [benchmarked a 176B model] on 384 V100 GPUs (24 DGX-2 nodes) and found that:

1. With 100 Gbps IB, we only have <20 TFLOPs per GPU (bad)
2. With 200-400 Gbps IB, we achieve reasonable TFLOPs around 30-40 per GPU (ok)
3. For 800 Gbps IB, we reach 40+ TFLOPs per GPU (excellent)

To remind the peak TFLOPS for NVIDIA V100 at fp16 is [125 TFLOPS].

But be careful here - this benchmark is for V100s! Which is about 2-3x slower than A100, and 4-8x slower than H100 for half-precision. So the comms have to be at least 4-8x faster for H100 nodes to match the above table at half precision. We need more benchmarks with more recent hardware.

footnote: the 2-3x range is because the official specs claim 3x TFLOPS increase for V100->A100, and A100->H100 each, but users benchmarking the difference report at most 2.5x improvements.

They also noticed that when training at scale, the communication overhead is more pronounced with small micro-batch size per GPU. And we may not be able to increase micro-batch size since global-batch size is often fixed to achieve good model convergence rate. This is solved by the recently introduced ZeRO++.

Finally, when doing the math above you need to know the actual bandwidth you get on your setup - which changes with payload size - the larger the payload the better the bandwidth. To get this information you need to look at your `reduce_bucket_size` and `prefetch_bucket_size` settings in the Deepspeed configuration file for reduction and prefetch correspondingly. The default is 0.5B params, which is 1GB in half-precision (0.5B x 2 bytes), or 2GB (0.5B x 4 bytes) if you use fp32 precision. So in order to measure the actual throughput you need to run an `all_reduce` benchmark with that payload and see what bandwidth gets reported. Then you can feed it to the calculations above.

Which Strategy To Use When

Here is a very rough outline at which parallelism strategy to use when. The first on each list is typically faster.

⇨ Single GPU

- Model fits onto a single GPU:
  1. Normal use
- Model doesn't fit onto a single GPU:
  1. ZeRO + Offload CPU and optionally NVMe
  2. as above plus Memory Centric Tiling (see below for details) if the largest layer can't fit into a single GPU
- Largest Layer not fitting into a single GPU:

1. ZeRO - Enable Memory Centric Tiling (MCT). It allows you to run arbitrarily large layers by automatically splitting them and executing them sequentially. MCT reduces the number of parameters that are live on a GPU, but it does not affect the activation memory. As this need is very rare as of this writing a manual override of `torch.nn.Linear` needs to be done by the user.

⇨ Single Node / Multi-GPU

- Model fits onto a single GPU:
  1. DDP - Distributed DP
  2. ZeRO - may or may not be faster depending on the situation and configuration used
- Model doesn't fit onto a single GPU:
  1. PP
  2. ZeRO
  3. TP

  With very fast intra-node connectivity of NVLINK or NVSwitch all three should be mostly on par, without these PP will be faster than TP or ZeRO. The degree of TP may also make a difference. Best to experiment to find the winner on your particular setup.

  TP is almost always used within a single node. That is TP size <= gpus per node.
- Largest Layer not fitting into a single GPU:

1. If not using ZeRO - must use TP, as PP alone won't be able to fit.
2. With ZeRO see the same entry for "Single GPU" above

⇨ **Multi-Node / Multi-GPU**

- If the model fits into a single node first try ZeRO with multiple replicas, because then you will be doing ZeRO over the faster intra-node connectivity, and DDP over slower inter-node
- When you have fast inter-node connectivity:
    1. ZeRO - as it requires close to no modifications to the model
    2. PP+TP+DP - less communications, but requires massive changes to the model
- when you have slow inter-node connectivity and still low on GPU memory:
    1. DP+PP+TP+ZeRO-1

Contributors

Samyam Rajbhandari, Horace He, Siddharth Singh, Olatunji Ruwase,

Working in SLURM Environment

Unless you're lucky and you have a dedicated cluster that is completely under your control chances are that you will have to use SLURM to timeshare the GPUs with others. But, often, if you train at HPC, and you're given a dedicated partition you still will have to use SLURM.

The SLURM abbreviation stands for: **Simple Linux Utility for Resource Management** - though now it's called The Slurm Workload Manager. It is a free and open-source job scheduler for Linux and Unix-like kernels, used by many of the world's supercomputers and computer clusters.

These chapters will not try to exhaustively teach you SLURM as there are many manuals out there, but will cover some specific nuances that are useful to help in the training process.

- SLURM For Users - everything you need to know to do your training in the SLURM environment.
- SLURM Administration - if you're unlucky to need to also manage the SLURM cluster besides using it, there is a growing list of recipes in this document to get things done faster for you.
- Performance - SLURM performance nuances.

SLURM Administration

Run a command on multiple nodes

1. to avoid being prompted with:

```
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

for every new node you haven't logged into yet, you can disable this check with:

```
echo "Host *" >> ~/.ssh/config
echo "  StrictHostKeyChecking no" >> ~/.ssh/config
```

Of course, check if that's secure enough for your needs. I'm making an assumption that you're already on the SLURM cluster and you're not ssh'ing outside of your cluster. You can choose not to set this and then you will have to manually approve each new node.

2. Install `pdsh`

You can now run the wanted command on multiple nodes.

For example, let's run `date`:

```
$ PDSH_RCMD_TYPE=ssh pdsh -w node-[21,23-26] date
node-25: Sat Oct 14 02:10:01 UTC 2023
node-21: Sat Oct 14 02:10:02 UTC 2023
node-23: Sat Oct 14 02:10:02 UTC 2023
node-24: Sat Oct 14 02:10:02 UTC 2023
node-26: Sat Oct 14 02:10:02 UTC 2023
```

Let's do something more useful and complex. Let's kill all GPU-tied processes that didn't exit when the SLURM job was cancelled:

First, this command will give us all process ids that tie up the GPUs:

```
nvidia-smi --query-compute-apps=pid --format=csv,noheader | sort | uniq
```

So we can now kill all those processes in one swoop:

```
 PDSH_RCMD_TYPE=ssh pdsh -w node-[21,23-26]  "nvidia-smi --query-compute-apps=pid --format=csv,noheader |
sort | uniq | xargs -n1 sudo kill -9"
```

Slurm settings

Show the slurm settings:

```
sudo scontrol show config
```

The config file is `/etc/slurm/slurm.conf` on the slurm controller node.

Once `slurm.conf` was updated to reload the config run:

```
sudo scontrol reconfigure
```

from the controller node.

Auto-reboot

If the nodes need to be rebooted safely (e.g. if the image has been updated), adapt the list of the node and run:

```
scontrol reboot ASAP node-[1-64]
```

For each of the non-idle nodes this command will wait till the current job ends, then reboot the node and bring it back up to `idle`.

Note that you need to have:

```
RebootProgram = "/sbin/reboot"
```

set in `/etc/slurm/slurm.conf` on the controller node for this to work (and reconfigure the SLURM daemon if you have just added this entry to the config file).

Changing the state of the node

The change is performed by `scontrol update`

Examples:

To undrain a node that is ready to be used:

```
scontrol update nodename=node-5 state=idle
```

To remove a node from the SLURM's pool:

```
scontrol update nodename=node-5 state=drain
```

Undrain nodes killed due to slow process exit

Sometimes processes are slow to exit when a job has been cancelled. If the SLURM was configured not to wait forever it'll automatically drain such nodes. But there is no reason for those nodes to not be available to the users.

So here is how to automate it.

The keys is to get the list of nodes that are drained due to `"Kill task failed"`, which is retrieved with:

```
sinfo -R | grep "Kill task failed"
```

now extract and expand the list of nodes, check that the nodes are indeed user-process free (or try to kill them first) and then undrain them.

Earlier you learned how to run a command on multiple nodes which we will use in this script.

Here is the script that does all that work for you: undrain-good-nodes.sh

Now you can just run this script and any nodes that are basically ready to serve but are currently drained will be switched to `idle` state and become available for the users to be used.

Modify a job's timelimit

To set a new timelimit on a job, e.g., 2 days:

```
scontrol update JobID=$SLURM_JOB_ID TimeLimit=2-00:00:00
```

To add additional time to the previous setting, e.g. 3 more hours.

```
scontrol update JobID=$SLURM_JOB_ID TimeLimit=+10:00:00
```

When something goes wrong with SLURM

Analyze the events log in the SLURM's log file:

```
sudo cat /var/log/slurm/slurmctld.log
```

This, for example, can help to understand why a certain node got its jobs cancelled before time or the node got removed completely.

SLURM for users

Quick start

Simply copy this example.slurm and adapt it to your needs.

SLURM partitions

In this doc we will use an example setup with these 2 cluster names:

- `dev`
- `prod`

To find out the hostname of the nodes and their availability, use:

```
sinfo -p dev
sinfo -p prod
```

Slurm configuration is at `/opt/slurm/etc/slurm.conf`.

Wait time for resource granting

```
squeue -u `whoami` --start
```

will show when any pending jobs are scheduled to start.

They may start sooner if others cancel their reservations before the end of the reservation.

Request allocation via dependency

To schedule a new job when one more of the currently scheduled job ends (regardless of whether it still running or not started yet), use the dependency mechanism, by telling `sbatch` to start the new job once the currently running job succeeds, using:

```
sbatch --dependency=CURRENTLY_RUNNING_JOB_ID tr1-13B-round1.slurm
```

Using `--dependency` may lead to shorter wait times that using `--begin`, since if the time passed to `--begin` allows even for a few minutes of delay since the stopping of the last job, the scheduler may already start some other jobs even if their priority is lower than our job. That's because the scheduler ignores any jobs with `--begin` until the specified time arrives.

Make allocations at a scheduled time

To postpone making the allocation for a given time, use:

```
salloc --begin HH:MM MM/DD/YY
```

Same for `sbatch`.

It will simply put the job into the queue at the requested time, as if you were to execute this command at this time. If resources are available at that time, the allocation will be given right away. Otherwise it'll be queued up.

Sometimes the relative begin time is useful. And other formats can be used. Examples:

```
--begin now+2hours
--begin=16:00
--begin=now+1hour
--begin=now+60  # seconds by default
--begin=2010-01-20T12:34:00
```

the time-units can be `seconds` (default), `minutes`, `hours`, `days`, or `weeks`:

Preallocated node without time 60min limit

This is very useful for running repetitive interactive experiments - so one doesn't need to wait for an allocation to progress. so the strategy is to allocate the resources once for an extended period of time and then running interactive `srun` jobs using this allocation.

set `--time` to the desired window (e.g. 6h):

```
salloc --partition=dev --nodes=1 --ntasks-per-node=1 --cpus-per-task=96 --gres=gpu:8 --time=6:00:00 bash
salloc: Pending job allocation 1732778
salloc: job 1732778 queued and waiting for resources
salloc: job 1732778 has been allocated resources
salloc: Granted job allocation 1732778
```

now use this reserved node to run a job multiple times, by passing the job id of `salloc`:

```
srun --jobid $SLURM_JOBID --pty bash
```

if run from inside `bash` started via `salloc`. But it can be started from another shell, but then explicitly set `--jobid`.

if this `srun` job timed out or manually exited, you can re-start it again in this same reserved node.

`srun` can, of course, call the real training command directly and not just `bash`.

Important: when allocating a single node, the allocated shell is not on the node (it never is). You have to find out the hostname of the node (reports when giving the allocation or via `squeue` and `ssh` to it.

When finished, to release the resources, either exit the shell started in `salloc` or `scancel JOBID`.

This reserved node will be counted towards hours usage the whole time it's allocated, so release as soon as done with it.

Actually, if this is just one node, then it's even easier to not use `salloc` but to use `srun` in the first place, which will both allocate and give you the shell to use:

```
srun --pty --partition=dev --nodes=1 --ntasks=1 --cpus-per-task=96 --gres=gpu:8 --time=60 bash
```

Hyper-Threads

By default, if the cpu has hyper-threads (HT), SLURM will use it. If you don't want to use HT you have to specify `--hint=nomultithread`.

footnote: HT is Intel-specific naming, the general concept is simultaneous multithreading (SMT)

For example for a cluster with with 2 cpus per node with 24 cores and 2 hyper-threads each, there is a total of 96 hyper-threads or 48 cpu-cores available. Therefore to utilize the node fully you'd need to configure either:

```
#SBATCH --cpus-per-task=96
```

or if you don't want HT:

```
#SBATCH --cpus-per-task=48
#SBATCH --hint=nomultithread
```

This last approach will allocate one thread per core and in this mode there are only 48 cpu cores to use.

Note that depending on your application there can be quite a performance difference between these 2 modes. Therefore try both and see which one gives you a better outcome.

On some setups like AWS the network's performance degrades dramatically when `--hint=nomultithread` is used!

Reuse allocation

e.g. when wanting to run various jobs on identical node allocation.

In one shell:

```
salloc --partition=prod --nodes=16 --ntasks=16 --cpus-per-task=96 --gres=gpu:8 --time=3:00:00 bash
echo $SLURM_JOBID
```

In another shell:

```
export SLURM_JOBID=<JOB ID FROM ABOVE>
srun --jobid=$SLURM_JOBID ...
```

You may need to set `--gres=gpu:0` to run some diagnostics job on the nodes. For example, let's check shared memory of all the hosts:

```
srun --jobid 631078 --gres=gpu:0 bash -c 'echo $(hostname) $(df -h | grep shm)'
```

Specific nodes selection

To exclude specific nodes (useful when you know some nodes are broken, but are still in IDLE state):

```
sbatch --exclude nodeA,nodeB
```

or via: `#SBATCH --exclude ...`

To use specific nodes:

```
sbatch --nodelist= nodeA,nodeB
```

can also use the short `-w` instead of `--nodelist`

The administrator could also define a `feature=example` in `slurm.conf` and then a user could ask for that subset of nodes via `--constraint=example`

Signal the running jobs to finish

Since each SLURM run has a limited time span, it can be configured to send a signal of choice to the program a desired amount of time before the end of the allocated time.

```
--signal=[[R][B]:]<sig_num>[@<sig_time>]
```

TODO: need to experiment with this to help training finish gracefully and not start a new cycle after saving the last checkpoint.

Detailed job info

While most useful information is preset in various `SLURM_*` env vars, sometimes the info is missing. In such cases use:

```
scontrol show -d job $SLURM_JOB_ID
```

and then parse out what's needed.

For a job that finished its run use:

```
sacct -j JOBID
```

e.g. with more details, depending on the partition:

```
sacct -u `whoami` --partition=dev  -ojobid,start,end,state,exitcode --format nodelist%300  -j JOBID
sacct -u `whoami` --partition=prod -ojobid,start,end,state,exitcode --format nodelist%300  -j JOBID
```

show jobs

Show only my jobs:

```
squeue -u `whoami`
```

Show jobs by job id:

```
squeue -j JOBID
```

Show jobs of a specific partition:

```
squeue --partition=dev
```

Aliases

Handy aliases

```
alias myjobs='squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"'
alias groupjobs='squeue -u foo,bar,tar -o "%.16i %u %9P %26j %.8T %.10M %.8l %.6D %.20S %R"'
alias myjobs-pending="squeue -u `whoami` --start"
alias idle-nodes="sinfo -p prod -o '%A'"
```

Zombies

If there are any zombies left behind across nodes, send one command to kill them all.

```
srun pkill python
```

Detailed Access to SLURM Accounting

`sacct` displays accounting data for all jobs and job steps in the Slurm job accounting log or Slurm database.

So this is a great tool for analysing past events.

For example, to see which nodes were used to run recent gpu jobs:

```
sacct -u `whoami` --partition=dev -ojobid,start,end,state,exitcode --format nodelist%300
```

`%300` here tells it to use a 300 char width for the output, so that it's not truncated.

See `man sacct` for more fields and info fields.

Queue

Cancel job

To cancel a job:

```
scancel [jobid]
```

To cancel all of your jobs:

```
scancel -u <userid>
```

To cancel all of your jobs on a specific partition:

```
scancel -u <userid> -p <partition>
```

Tips

- if you see that `salloc`'ed interactive job is scheduled to run much later than you need, try to cancel the job and ask for shorter period - often there might be a closer window for a shorter time allocation.

Logging

If we need to separate logs to different log files per node add `%N` (for short hostname) so that we have:

```
#SBATCH --output=%x-%j-%N.out
```

That way we can tell if a specific node misbehaves - e.g. has a corrupt GPU. This is because currently pytorch doesn't log which node / gpu rank triggered an exception.

Hoping it'll be a built-in feature of pytorch https://github.com/pytorch/pytorch/issues/63174 and then one won't need to make things complicated on the logging side.

Show the state of nodes

```
sinfo -p PARTITION
```

Very useful command is:

```
sinfo -s
```

and look for the main stat, e.g.:

```
NODES(A/I/O/T) "allocated/idle/other/total".
597/0/15/612
```

So here 597 out of 612 nodes are allocated. 0 idle and 15 are not available for whatever other reasons.

```
sinfo -p gpu_p1 -o "%A"
```

gives:

```
NODES(A/I)
236/24
```

so you can see if any nodes are available on the 4x v100-32g partition (`gpu_p1`)

To check a specific partition:

```
sinfo -p gpu_p1 -o "%A"
```

See the table at the top of this document for which partition is which.

sinfo states

- idle: no jobs running
- alloc: nodes are allocated to jobs that are currently executing
- mix: the nodes have some of the CPUs allocated, while others are idle
- drain: the node is unavailable due to an administrative reason
- drng: the node is running a job, but will after completion not be available due to an administrative reason

drained nodes

To see all drained nodes and the reason for drainage (edit `%50E` to make the reason field longer/shorter)

```
% sinfo -R -o "%50E %12U %19H %6t %N"
```

or just `-R` if you want it short:

```
% sinfo -R
```

Job arrays

To run a sequence of jobs, so that the next slurm job is scheduled as soon as the currently running one is over in 20h we use a job array.

Let's start with just 10 such jobs:

```
sbatch --array=1-10%1 array-test.slurm
```

`%1` limits the number of simultaneously running tasks from this job array to 1. Without it it will try to run all the jobs at once, which we may want sometimes (in which case remove %1), but when training we need one job at a time.

Alternatively, as always this param can be part of the script:

```
#SBATCH --array=1-10%1
```

Here is toy slurm script, which can be used to see how it works:

```
#!/bin/bash
#SBATCH --job-name=array-test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=1            # number of cores per tasks
#SBATCH --time 00:02:00              # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --error=%x-%j.out            # error file name (same to watch just one file)
#SBATCH --partition=dev

echo $SLURM_JOB_ID
echo "I am job ${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}"
date
sleep 10
date
```

Note `$SLURM_ARRAY_JOB_ID` is the same as `$SLURM_JOB_ID`, and `$SLURM_ARRAY_TASK_ID` is the index of the job.

To see the jobs running:

```
$ squeue -u `whoami` -o "%.10i %9P %26j %.8T %.10M %.6D %.20S %R"
     JOBID PARTITION                       NAME    STATE       TIME  NODES           START_TIME
NODELIST(REASON)
591970_[2-   dev             array-test  PENDING       0:00      1  2021-07-28T20:01:06
(JobArrayTaskLimit)
```

now job 2 is running.

To cancel the whole array, cancel the job id as normal (the number before `_`):

```
scancel 591970
```

To cancel a specific job:

```
scancel 591970_2
```

If it's important to have the log-file contain the array id, add `%A_%a`:

```
#SBATCH --output=%x-%j.%A_%a.log
```

More details https://slurm.schedmd.com/job_array.html

Job Array Trains and their Suspend and Release

In this recipe we accomplish 2 things:

1. Allow modification to the next job's slurm script
2. Allow suspending and resuming job arrays w/o losing the place in the queue when not being ready to continue running a job

SLURM is a very unforgiving environment where a small mistake can cost days of waiting time. But there are strategies to mitigate some of this harshness.

SLURM jobs have a concept of "age" in the queue which besides project priority governs when a job gets scheduled to run. If your have just scheduled a new job it has no "age" and will normally be put to run last compared to jobs that have entered the queue earlier. Unless of course this new job comes from a high priority project in which case it'll progress faster.

So here is how one can keep the "age" and not lose it when needing to fix something in the running script or for example to switch over to another script.

The idea is this:

1. `sbatch` a long job array, e.g., `-array=1-50%1`
2. inside the slurm script don't have any code other than `source another-script.slurm` - so now you can modify the target script or symlink to another script before the next job starts
3. if you need to stop the job array train - don't cancel it, but suspend it without losing your place in a queue
4. when ready to continue - unsuspend the job array - only the time while it was suspended is not counted towards its age, but all the previous age is retained.

The only limitation of this recipe is that you can't change the number of nodes, time and hardware and partition constraints once the job array was launched.

Here is an example:

Create a job script:

```
$ cat train-64n.slurm
#!/bin/bash
#SBATCH --job-name=tr8-104B
#SBATCH --nodes=64
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 20:00:00              # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
```

```
#SBATCH --partition=dev

source tr8-104B-64.slurm
```

Start it as:

```
sbatch --array=1-50%1 train-64.slurm
```

Now you can easily edit `tr8-104B-64.slurm` before the next job run and either let the current job finish if it's desired or if you need to abort it, just kill the currently running job, e.g. `1557903_5` (not job array `1557903`) and have the train pick up where it left, but with the edited script.

The nice thing is that this requires no changes to the original script (`tr8-104B-64.slurm` in this example), and the latter can still be started on its own.

Now, what if something is wrong and you need 10min or 10h to fix something. In this case we suspend the train using:

```
scontrol hold <jobid>
```

with being either a "normal" job, the id of a job array or the id for a job array step

and then when ready to continue release the job:

```
scontrol release <jobid>
```

Troubleshooting

Mismatching nodes number

If the pytorch launcher fails it often means that the number of SLURM nodes and the launcher nodes are mismatching, e.g.:

```
grep -ir nodes= tr123-test.slurm
#SBATCH --nodes=40
NNODES=64
```

This won't work. They have to match.

You can add a sanity check to your script:

```
#!/bin/bash
#SBATCH --job-name=test-mismatch
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
```

```
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

[...]

NNODES=2

# sanity check for having NNODES and `#SBATCH --nodes` match, assuming you use NNODES variable
if [ "$NNODES" != "$SLURM_NNODES" ]; then
    echo "Misconfigured script: NNODES=$NNODES != SLURM_NNODES=$SLURM_NNODES"
    exit 1
fi

[...]
```

or you could just do:

```
#SBATCH --nodes=2
[...]
NNODES=$SLURM_NNODES
```

and then it will always be correct

Find faulty nodes and exclude them

Sometimes a node is broken, which prevents one from training, especially since restarting the job often hits the same set of nodes. So one needs to be able to isolate the bad node(s) and exclude it from `sbatch`.

To find a faulty node, write a small script that reports back the status of the desired check.

For example to test if cuda is available on all nodes:

```
python -c ’import torch, socket; print(f"{socket.gethostname()}: {torch.cuda.is_available()}")’
```

and to only report the nodes that fail:

```
python -c ’import torch, socket; torch.cuda.is_available() or print(f"Broken node:
{socket.gethostname()}") ’
```

Of course, the issue could be different - e.g. gpu can’t allocate memory, so change the test script to do a small allocation on cuda. Here is one way:

```
python -c "import torch; torch.ones(1000,1000).cuda()"
```

But since we need to run the test script on all nodes and not just the first node, the slurm script needs to run it via `srun`. So our first diagnostics script can be written as:

```
srun --jobid $SLURM_JOBID bash -c 'python -c "import torch, socket; print(socket.gethostname(),
torch.cuda.is_available())"'
```

I slightly changed it, due to an issue with quotes.

You can always convert the one liner into a real script and then there is no issue with quotes.

```
$ cat << EOT >> test-nodes.py
#!/usr/bin/env python
import torch, socket
print(socket.gethostname(), torch.cuda.is_available())
EOT
$ chmod a+x ./test-nodes.py
```

Now let's create a driver slurm script. Use a few minutes time for this test so that SLURM yields it faster:

```
#!/bin/bash
#SBATCH --job-name=test-nodes
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

source $six_ALL_CCFRWORK/start-prod
srun --jobid $SLURM_JOBID ./test-nodes.py
```

Once it runs check the logs to see if any reported `False`, those are the nodes you want to exclude.

Now once the faulty node(s) is found, feed it to `sbatch`:

```
sbatch --exclude=hostname1,hostname2 ...
```

and `sbatch` will exclude the bad nodes from the allocation.

Additionally please report the faulty nodes to `#science-support` so that they get replaced

Here are a few more situations and how to find the bad nodes in those cases:

Broken NCCL

If you're testing something that requires distributed setup, it's a bit more complex. Here is a slurm script that tests that NCCL works. It sets up NCCL and checks that barrier works:

```
#!/bin/bash
#SBATCH --job-name=test-nodes-nccl
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

source $six_ALL_CCFRWORK/start-prod

NNODES=2

GPUS_PER_NODE=4
MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=6000

export LAUNCHER="python -u -m torch.distributed.launch \
    --nproc_per_node $GPUS_PER_NODE \
    --nnodes $NNODES \
    --master_addr $MASTER_ADDR \
    --master_port $MASTER_PORT \
    "

export SCRIPT=test-nodes-nccl.py

cat << EOT > $SCRIPT
#!/usr/bin/env python
import torch.distributed as dist
import torch
import socket
import os
import fcntl

def printflock(*msgs):
    """ print """
    with open(__file__, "r") as fh:
        fcntl.flock(fh, fcntl.LOCK_EX)
        try:
            print(*msgs)
        finally:
            fcntl.flock(fh, fcntl.LOCK_UN)

local_rank = int(os.environ["LOCAL_RANK"])
torch.cuda.set_device(local_rank)
dist.init_process_group("nccl")
header = f"{socket.gethostname()}-{local_rank}"
try:
    dist.barrier()
    printflock(f"{header}: NCCL {torch.cuda.nccl.version()} is OK")
```

```
except:
    printflock(f"{header}: NCCL {torch.cuda.nccl.version()} is broken")
    raise
EOT

echo $LAUNCHER --node_rank $SLURM_PROCID $SCRIPT

srun --jobid $SLURM_JOBID bash -c ’$LAUNCHER --node_rank $SLURM_PROCID $SCRIPT’
```

The script uses `printflock` to solve the interleaved print outputs issue.

GPU Memory Check

This tests if each GPU on the allocated nodes can successfully allocate 77Gb (e.g. to test 80GB A100s) (have to subtract a few GBs for cuda kernels).

```
import torch, os
import time
import socket
hostname = socket.gethostname()

local_rank = int(os.environ["LOCAL_RANK"]);

gbs = 77
try:
    torch.ones((gbs*2**28)).cuda(local_rank).contiguous() # alloc on cpu, then move to gpu
    print(f"{local_rank} {hostname} is OK")
except:
    print(f"{local_rank} {hostname} failed to allocate {gbs}GB DRAM")
    pass

time.sleep(5)
```

Broken Network

Yet another issue with a node is when its network is broken and other nodes fail to connect to it.

You’re likely to experience it with an error similar to:

```
work = default_pg.barrier(opts=opts)
RuntimeError: NCCL error in: /opt/conda/conda-bld/pytorch_1616554793803/work/torch/lib/c10d/
ProcessGroupNCCL.cpp:825, unhandled system error, NCCL version 2.7.8
ncclSystemError: System call (socket, malloc, munmap, etc) failed.
```

Here is how to debug this issue:

1. Add:

```
export NCCL_DEBUG=INFO
```

before the `srun` command and re-run your slurm script.

2. Now study the logs. If you find:

```
r11i6n2:486514:486651 [1] include/socket.h:403 NCCL WARN Connect to 10.148.3.247<56821> failed :
Connection refused
```

Let's see which node refuses to accept connections. We get the IP address from the error above and reverse resolve it to its name:

```
nslookup 10.148.3.247
247.3.148.10.in-addr.arpa       name = r10i6n5.ib0.xa.idris.fr.
```

Add `--exclude=r10i6n5` to your `sbatch` command and report it to JZ admins.

Run py-spy or any other monitor program across all nodes

When dealing with hanging, here is how to automatically log `py-spy` traces for each process.

Of course, this same process can be used to run some command for all nodes of a given job. i.e. it can be used to run something during the normal run - e.g. dump all the memory usage in each process via `nvidia-smi` or whatever other program is needed to be run.

```
cd ~/prod/code/tr8b-104B/bigscience/train/tr11-200B-ml/

salloc --partition=prod --nodes=40 --ntasks-per-node=1 --cpus-per-task=96 --gres=gpu:8 --time 20:00:00

bash 200B-n40-bf16-mono.slurm
```

In another shell get the JOBID for the above `salloc`:

```
squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"
```

adjust jobid per above and the nodes count (XXX: probably can remove `--nodes=40` altogether and rely on `salloc` config):

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c ’ps aux | grep
python | egrep -v "grep|srun" | grep `whoami` | awk "{print \$2}" | xargs -I {} py-spy dump --native --pid
{}’ || echo "failed"
```

now all `py-spy` traces go into the `trace-$nodename.out` files under `cwd`.

The key is to use `--gres=gpu:0` or otherwise the 2nd `srun` will block waiting for the first one to release the gpus.

Also the assumption is that some conda env that has `py-spy` installed got activated in `~/.bashrc`. If yours doesn't already do that, add the instruction to load the env to the above command, before the `py-spy` command - it'll fail to find it otherwise.

Don't forget to manually release the allocation when this process is done.

Convert SLURM_JOB_NODELIST into a hostfile

Some multi-node launchers require a `hostfile` - here is how to generate one:

```
# autogenerate the hostfile for deepspeed
# 1. deals with: SLURM_JOB_NODELIST in either of 2 formats:
# r10i1n8,r10i2n0
# r10i1n[7-8]
# 2. and relies on SLURM_STEP_GPUS=0,1,2... to get how many gpu slots per node
#
# usage:
# makehostfile > hostfile
function makehostfile() {
perl -le '$slots=split /,/, $ENV{"SLURM_STEP_GPUS"}; $_=$ENV{"SLURM_JOB_NODELIST"}; if
(/^(.*?)\[(\d+)-(\d+)\]/) { print map { "$1$_ slots=$slots\n" } $2..$3} elsif (/,/) { print map { "$1$_
slots=$slots\n" } split /,/ } '
}
```

Environment variables

You can always do:

```
export SOMEKEY=value
```

from the slurm script to get a desired environment variable passed to the program launched from it.

And you can also add to the top of the slurm script:

```
#SBATCH --export=ALL
```

The launched program will see all the environment variables visible in the shell where it was launched from.

Crontab Emulation

One of the most important Unix tools is the crontab, which is essential for being able to schedule various jobs. It however usually is absent from SLURM environment. Therefore one must emulate it. Here is how.

For this presentation we are going to use `$WORK/cron/` as the base directory. And that you have an exported environment variable `WORK` pointing to some location on your filesystem - if you use Bash you can set it up in your `~/.bash_profile` or if a different shell is used use whatever startup equivalent file is.

1. A self-perpetuating scheduler job

We will use `$WORK/cron/scheduler` dir for scheduler jobs, `$WORK/cron/cron.daily` for daily jobs and `$WORK/cron/cron.hourly` for hourly jobs:

```
$ mkdir -p $WORK/cron/scheduler
$ mkdir -p $WORK/cron/cron.daily
$ mkdir -p $WORK/cron/cron.hourly
```

Now copy these two slurm script in `$WORK/cron/scheduler`:

- cron-daily.slurm
- cron-hourly.slurm

after editing those to fit your specific environment's account and partition information.

Now you can launch the crontab scheduler jobs:

```
$ cd $WORK/cron/scheduler
$ sbatch cron-hourly.slurm
$ sbatch cron-daily.slurm
```

This is it, these jobs will now self-perpetuate and usually you don't need to think about it again unless there is an even that makes SLURM lose all its jobs.

2. Daily and Hourly Cronjobs

Now whenever you want some job to run once a day, you simply create a slurm job and put it into the `$WORK/cron/cron.daily` dir.

Here is an example job that runs daily to update the `mlocate` file index:

```
$ cat $WORK/cron/cron.daily/mlocate-update.slurm
#!/bin/bash
#SBATCH --job-name=mlocate-update    # job name
#SBATCH --ntasks=1                   # number of MP tasks
#SBATCH --nodes=1
#SBATCH --hint=nomultithread         # we get physical cores not logical
#SBATCH --time=1:00:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=PARTITION     # edit me
#SBATCH --account=GROUP@PARTITION # edit me

set -e
date
echo "updating mlocate db"
/usr/bin/updatedb -o $WORK/lib/mlocate/work.db -U $WORK --require-visibility 0
```

This builds an index of the files under `$WORK` which you can then quickly query with:

```
/usr/bin/locate -d $WORK/lib/mlocate/work.db pattern
```

To stop running this job, just move it out of the `$WORK/cron/cron.daily` dir.

The same principle applies to jobs placed into the `$WORK/cron/cron.hourly` dir. These are useful for running something every hour.

Please note that this crontab implementation is approximate timing-wise, due to various delays in SLURM scheduling they will run approximately every hour and every day. You can recode these to ask SLURM to start something at a more precise time if you have to, but most of the time the just presented method works fine.

Additionally, you can code your own variations to meet specific needs of your project, e.g., every-30min or every-12h jobs.

3. Cleanup

Finally, since every cron launcher job will leave behind a log file (which is useful if for some reason things don't work), you want to create a cronjob to clean up these logs. Otherwise you may run out of inodes - these logs files are tiny, but there could be tens of thousands of those.

You could use something like this in a daily job.

```
find $WORK/cron -name "*.out" -mtime +7 -exec rm -f {} +
```

Please note that it's set to only delete files that are older than 7 days, in case you need the latest logs for diagnostics.

Nuances

The scheduler runs with Unix permissions of the person who launched the SLRUM cron scheduler job and so all other SLURM scripts launched by that cron job.

Self-perpetuating SLURM jobs

The same approach used in [building a scheduler] can be used for creating stand-alone self-perpetuating jobs.

For example:

```
#!/bin/bash
#SBATCH --job-name=watchdog          # job name
#SBATCH --ntasks=1                   # number of MP tasks
#SBATCH --nodes=1
#SBATCH --time=0:30:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=PARTITION        # edit me

# ensure to restart self first 1h from now
RUN_FREQUENCY_IN_HOURS=1
sbatch --begin=now+${RUN_FREQUENCY_IN_HOURS}hour watchdog.slurm

... do the watchdog work here ...
```

and you launch it once with:

```
sbatch watchdog.slurm
```

This then will immediately schedule itself to be run 1 hour from the launch time and then the normal job work will be done. Regardless of whether the rest of the job will succeed or fail, this job will continue relaunching itself approximately once an hour. This is imprecise due to scheduler job starting overhead and node availability issues. But if there is a least one spare node available and the job itself is quick to finish the requirement to run at an approximate frequency should be sufficient.

As the majority of SLURM environment in addition to the expensive GPU nodes also provide much cheaper CPU-only nodes, you should choose a CPU-only SLURM partition for any jobs that don't require GPUs to run.

Getting information about the job

From within the slurm file one can access information about the current job's allocations.

Getting allocated hostnames and useful derivations based on that:

```
export HOSTNAMES=$(scontrol show hostnames "$SLURM_JOB_NODELIST")
export NUM_NODES=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | wc -l)
export MASTER_ADDR=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n 1)
```

Convert compact node list to expanded node list

Sometimes you get SLURM tools give you a string like: `node-[42,49-51]` which will require some coding to expand it into `node-42,node-49,node-50,node-51`, but there is a special tool to deal with that:

```
$ scontrol show hostnames node-[42,49-51]
node-42
node-49
node-50
node-51
```

Voila!

case study: this is for example useful if you want get a list of nodes that were drained because the job was too slow to exit, but really there is no real problem with the nodes. So this one-liner will give you the list of such nodes in an expanded format which you can then script to loop over this list to undrain these nodes after perhaps checking that the processes have died by this time:

```
sinfo -R | grep "Kill task failed" | perl -lne ’/(node-.*[\d\]]+)/ && print $1’ | xargs -n1 scontrol show
hostnames
```

Overcoming the lack of group SLURM job ownership

SLURM runs on Unix, but surprisingly its designers haven't adopted the concept of group ownership with regards to SLURM jobs. So if a member of your team started an array of 10 jobs 20h each, and went on vacation - unless you have `sudo` access you now can't do anything to stop those jobs if something is wrong.

I'm yet to find why this is so, but so far we have been using a kill switch workaround. You have to code it in your framework. For example, see how it was implemented in Megatron-Deepspeed (Meg-DS). The program polls for a pre-configured at start up path on the filesystem and if it finds a file there, it exits.

So if we start Meg-DS with `--kill-switch-path $WORK/tmp/training17-kill-switch` and then at any point we need to kill the SLURM job, we simply do:

```
touch $WORK/tmp/training17-kill-switch
```

and the next time the program gets to check for this file it'll detect the event and will exit voluntarily. If you have a job array, well, you will have to wait until each job starts, detects the kill switch and exits.

Of course, don't forget to remove it when you're done stopping the jobs.

```
rm $WORK/tmp/training17-kill-switch
```

Now, this doesn't always work. If the job is hanging, it'll never come to the point of checking for kill-switch and the only solution here is to contact the sysadmins to kill the job for you. Sometimes if the hanging is a simple case pytorch's distributed setup will typically auto-exit after 30min of preset timeout time, but it doesn't always work.

SLURM Performance

Here you will find discussions of SLURM-specific settings that impact performance.

srun's `--cpus-per-task` may need to be explicit

You need to make sure that the launched by `srun` program receives as many cpu-cores as intended. For example, in a typical case of a ML training program, each gpu needs at least one cpu-core for the process driving it plus a few more cores for the `DataLoader`. You need multiple cores so that each task can be performed in parallel. If you have 8 gpus and 2 `DataLoader` workers per gpu, you need at least `3*8=24` cpu-cores per node.

The number of cpus per task is defined by `--cpus-per-task`, which is passed to `sbatch` or `salloc` and originally `srun` would inherit this setting. However, recently this behavior has changed:

A quote from the `sbatch` manpage:

> NOTE: Beginning with 22.05, srun will not inherit the --cpus-per-task value requested by salloc or sbatch. It must be requested again with the call to srun or set with the SRUN_CPUS_PER_TASK environment variable if desired for the task(s).

Which means that if in the past your SLURM script could have been:

```
#SBATCH --cpus-per-task=48
[...]

srun myprogram
```

and the program launched by `srun` would have received 48 cpu-cores because `srun` used to inherit the `--cpus-per-task=48` settings from `sbatch` or `salloc` settings, according to the quoted documentation since SLURM 22.05 this behavior is no longer true.

footnote: I tested with SLURM@22.05.09 and the old behavior was still true, but this is definitely the case with 23.x series as reported here. So the change might have happened in the later 22.05 series.

So if you leave things as is, now the program will receive just 1 cpu-core (unless the `srun` default has been modified).

You can easily test if your SLURM setup is affected, using `os.sched_getaffinity(0))`, as it shows which cpu-cores are eligible to be used by the current process. So it should be easy to count those with `len(os.sched_getaffinity(0))`.

Here is how you can test if you're affected:

```
$ cat test.slurm
#!/bin/bash
#SBATCH --job-name=test-cpu-cores-per-task
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=48   # adapt to your env if you have less than 48 cpu cores
#SBATCH --time=0:10:00
```

```
#SBATCH --partition=x        # adapt to your env to the right partition name
#SBATCH --output=%x-%j.out

srun python -c 'import os; print(f"visible cpu cores: {len(os.sched_getaffinity(0))}")'
```

If you get

```
visible cpu cores: 48
```

then you don't need to do anything, if however you get:

```
visible cpu cores: 1
```

or another value smaller than 48 then you're affected.

To fix that you need to change your SLURM script to either:

```
#SBATCH --cpus-per-task=48
[...]

srun --cpus-per-task=48 myprogram
```

or:

```
#SBATCH --cpus-per-task=48
[...]

SRUN_CPUS_PER_TASK=48
srun myprogram
```

To enable Hyper-Threads or not

As explained in the Hyper-Threads section you should be able to double the number of available cpu-cores if your CPUs support hyper-threading and for some workloads this may lead to an overall faster performance.

However, you should test the performance w/ and w/o HT, compare the results and choose the setting that gives the best outcome.

case study: on AWS p4 nodes I discovered that enabling HT made the network throughput 4x slower. Since then we were careful to have HT disabled on that particular setup.

Selecting Training Hyper-Parameters And Model Initializations

The easiest way to find a good hparam and model init starter set is to steal it from a similar training that you know has succeeded. Here is a collection of public training LLM/VLM logbooks to get you started. The other common source is papers if they disclose that information. You can also try to reach out to the authors and ask them for these details if they didn't publish it.

Glossary

Training jargon uses a multitude of abbreviations and terms, so here are some important for this chapter.

- BS: Batch Size - here we mean batch size per gpu, often it is also referred to as MBS (micro-batch-size)
- GBS: Global Batch Size - total batch size per iteration - may include gradient accumulation
- GAS: Gradient Accumulation Steps - how many forward/backward cycles to perform before one full iteration is complete
- TFLOPs: Trillion FLOPs per second - FLOPS
- PP: Pipeline Parallelism

Global Batch Size Ramp Up

If you intend to train with a very large GBS, with say 1024, or 2048 samples and even higher, when you just start training, it's very wasteful to feed such large batch sizes to the model. At this point it's totally random and can't benefit from having too refined data. Therefore to save data and resources, one often ramps up the global batch size over some period of time.

It's also important to not start with GBS that is too small, since otherwise the progress won't be efficient. When there is too little data the compute (TFLOPS) is inefficient and will slow everything down. This is especially so when Pipeline Parallelism (PP) is used, since the most important thing about PP tuneup is a small GPU idleness bubble, and the smaller the GBS the larger the bubble is.

For example, for BLOOM-176B, where we did use PP, after doing throughput benchmarking we found that starting with GBS=16 was incredibly slow (8 TFLOPs), so we eventually started with GBS=192 (73 TFLOPs) and then we ramped up to GBS=2048 (150 TFLOPs) - we increased GBS by 16 every 9_765_625 samples.

STD Init

This hyper parameter is super-important and it requires math to get it right. For details see STD Init.

Avoiding, Recovering From and Understanding Instabilities

Sub-sections:

- Understanding Training Loss Patterns - types of spikes, divergences, grokking moments, resumes, etc.

Learning from Training Logbooks

The best learning is to read Publicly available training LLM/VLM logbooks because there you can see exactly what happened and how the problem has been overcome.

STD Init

Correctly initializing the initial distribution of the tensors can have a tremendous impact on training's stability. The `std` value isn't fixed and depends on the hidden dimension size.

This proved to be a very crucial setting in our pre-BLOOM 104B experiments and we couldn't break past the first few thousands iterations until we figured out that the 0.02 default `--init-method-std` in Megatron-LM was a way too big for our model.

We referred to these two sources:

1. "Transformers without Tears" paper https://arxiv.org/abs/1910.05895 prescribes: `sqrt(2/(NHIDDEN*5))`
2. The 530B training paper https://arxiv.org/abs/2201.11990 they used an even smaller init formula: `sqrt(1/(NHIDDEN*3))`

and decided to go with the 530B one as it leads to an even smaller init value.

To make it easier to compare the two formulas, they can be rewritten as:

1. `sqrt(0.4000/NHIDDEN)`
2. `sqrt(0.3333/NHIDDEN)`

Thus for `NHIDDEN=14336` the math was `sqrt(1/(14336*3)) = 0.00482` and that's what we used. It surely wasn't the only reason why we had no stability issues during BLOOM-176B training, but I think it was one of the crucial ones.

Numerical instabilities

Certain mathematical operations could be unstable when dealing with low precision numbers.

For example, please see this very interesting PyTorch guide on numerical stability.

Now let's look at a specific example of this concept in action.

During 104B training experiments where fp16 mixed precision was used - the following improvement was proposed by Corby Rosset to make self-attention more stable.

Specifically this line shows that the `norm_factor` may be multiplied after the Query * Key matrix multiplication. If the dim of Q and K are very large, the output may blow up and the `norm_factor` won't be able to save it.

Proposal: move the `norm_factor` inward, so Q and K are scaled down before matrix multiply:

```
        matmul_result = torch.baddbmm(
            matmul_result,
            1.0/math.sqrt(self.norm_factor) * query_layer.transpose(0, 1),   # [b * np, sq, hn]
            1.0/math.sqrt(self.norm_factor) * key_layer.transpose(0, 1).transpose(1, 2),  # [b * np, hn,
sk]
            beta=0.0 if alibi is None else 1.0, alpha=1.0)

        # change view to [b, np, sq, sk]
        attention_scores = matmul_result.view(*output_size)
```

To make the operation mathematically equivalent, moving the norm factor inward requires taking sqrt again if n is a scalar, A and B matrices:

```
n * (A dot B) === (sqrt(n) * A) dot (sqrt(n) * B)
```

Now A and B dimensions can be significantly larger.

For CUDA kernel writers CuBlas's `GemmStridedBatchedEx` at the time of this writing has a similar issue. It is defined as:

```
C+i*strideC=αop(A+i*strideA)op(B+i*strideB)+β(C+i*strideC), for i ∈[0,batchCount−1]
```

The issue is that `alpha` is multiplied after the matrix-matrix multiplication is done so it can cause instability.

"Bad" combination of data batch and model parameter state

PaLM team observed dozens of loss spikes at "highly irregular intervals" when training larger models. While they were not able to track down the root cause, they mitigated the issue by restarting from an earlier checkpoint and skipping potentially problematic data batches. Section 5.1 Training instability

Understanding Training Loss Patterns

Training loss plot is similar to the heart beat pattern - there is the good, the bad and you-should-worry one. After studying many training loss trajectories one develops an intuition to explain various loss behaviors during one's training and how to act on those.

I warn you that the "Understanding" in the title of this section is overloaded since very often we don't really understand why certain types of spikes happen. Here "understanding" refers to recognizing various patterns. We then usually have techniques to overcome the bad patterns and bring the training successfully to the finish line.

Thus you will find here a gallery of training loss patterns sometimes with real explanations, but more often than not educated guesses to what might be happening.

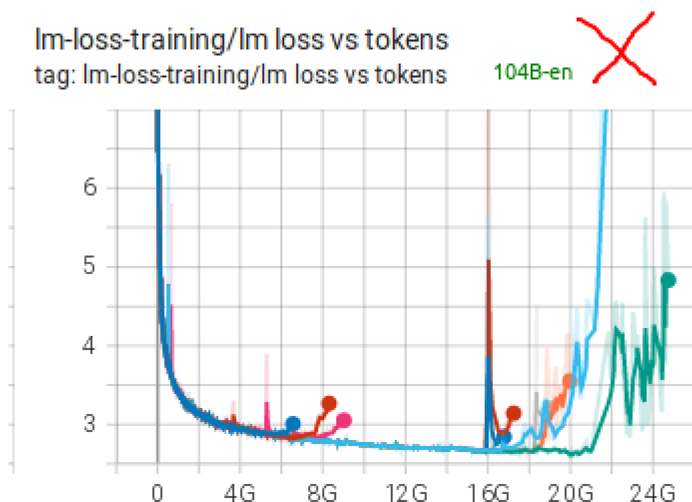
Please excuse the plot snapshots looking wildly different from each other as they have come from many sources over multiple years.

The good, the bad and the unexpected

Let's look at some good, bad and unusual patterns.

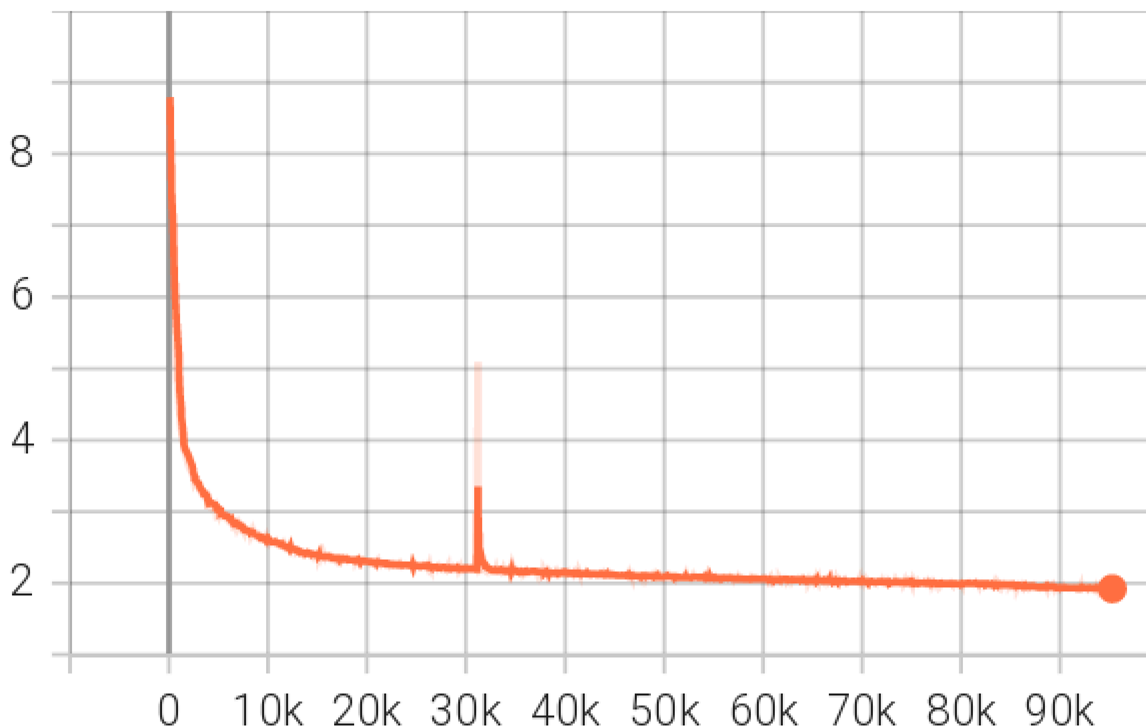
A very failed training

Prior to starting BLOOM-176B training we did multiple experiments with the 104B model. We failed to figure out how to not diverge very early on.



As you can see many attempts were made, many techniques were applied (see chronicles. We think the 2 main obstacles were using fp16 and data that had a lot of garbage in it. For BLOOM-176B we switched to bf16, used much cleaner data and also added an embedding layer-norm and that made all the difference.

An almost perfect training



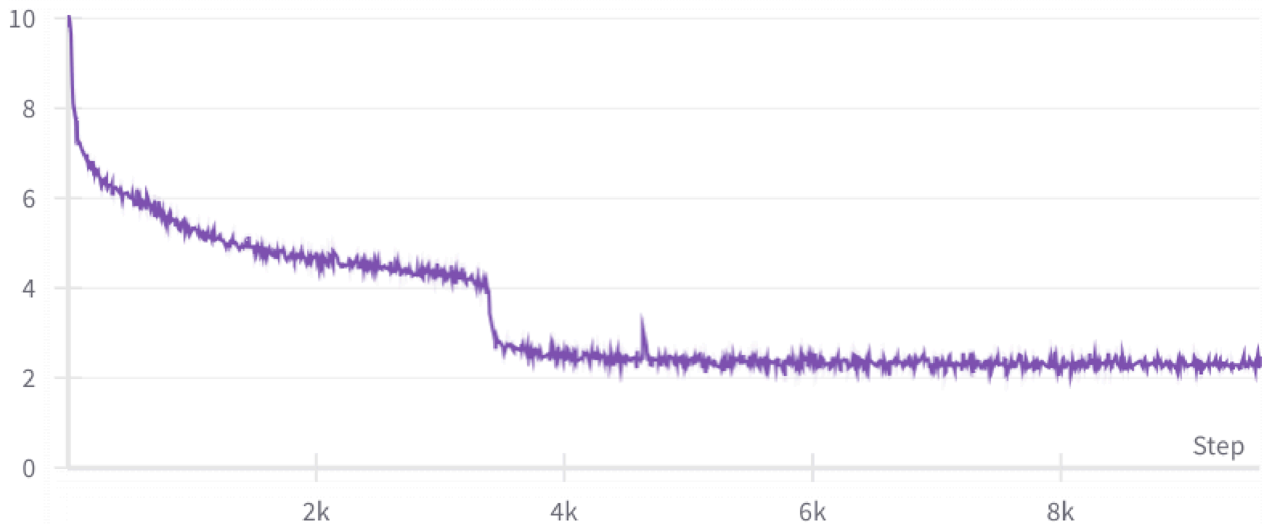
The BLOOM-176B training had a close to perfect training loss trajectory, with a single spike that has recovered in 200 steps.

You can inspect the TB to zoom in and check other plots.

This was the almost perfect training indeed. Lots of hard work was put into achieving this.

The grokking moment

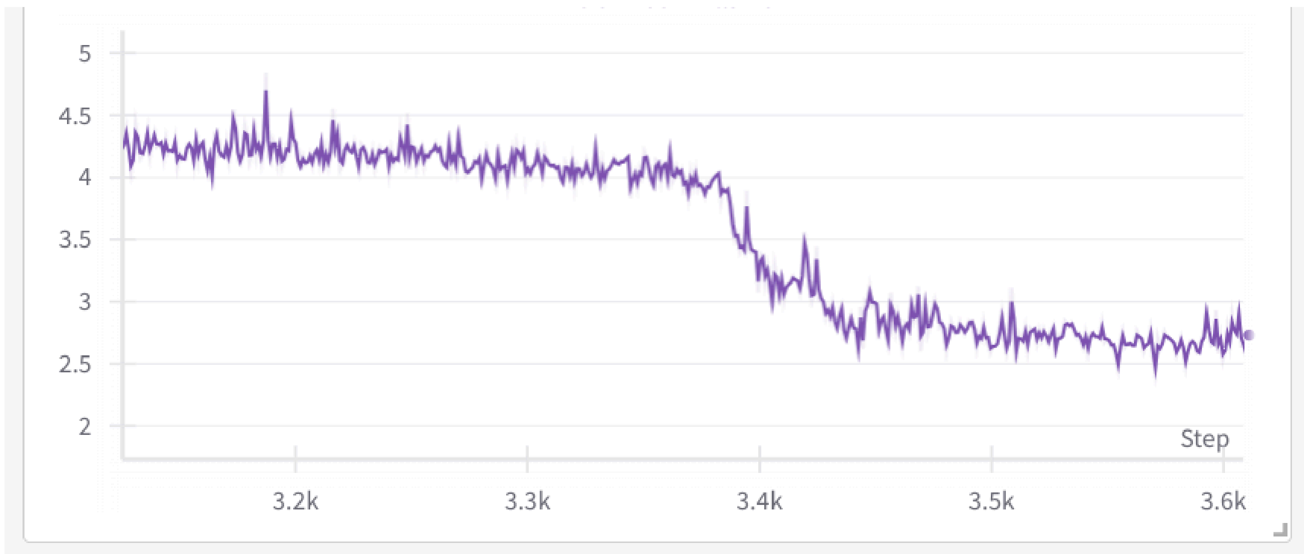
Recently I was doing some performance testing and run a tiny global batch size of 8 on 8x A100 nodes on llama-2-7b trained from scratch. (w/ Deepspeed ZeRO-3 DP using HF Transformers Llama implementation)



Here one can observe a rapid loss improvement from 4 to 2.5 in just 480 samples after a very steady much slower improvements. My colleague [Gautam Mittal] called it the [grokking] moment. In just a handful of steps the model suddenly generalized to much better predict the masked tokens.

Normally one doesn't see such a dramatic improvement when using a much larger batch size.

If we zoom in it took about 60 8-sample per iteration steps:



Main types of loss spikes

In general there are 3 types of loss spikes:

1. Fast recovering spikes
2. Slow recovering spikes
3. Not fully recovering spikes

The spikes usually happen because of a bad data pocket, either due to badly shuffled data or because it hasn't been cleaned from some garbage scraped from the websites.

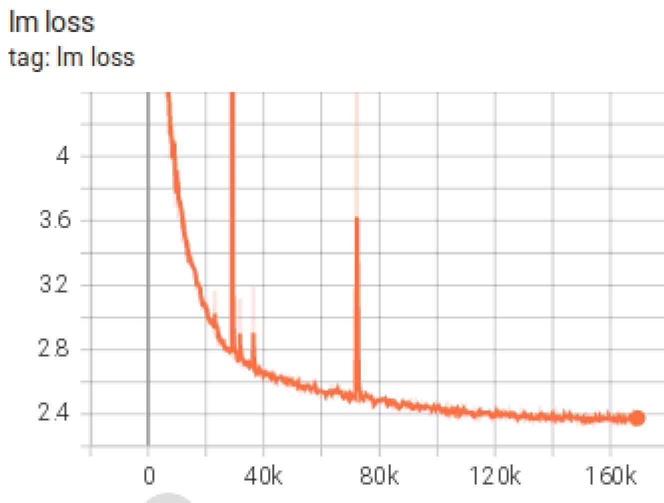
While one would suspect that the batch before the spike was the trigger, but if you were to study that batch's contents

you are likely to find nothing unusual - quite often the problem starts developing many steps before and then most of the sudden it happens. But also it might not be easy to study the batch, since it could amount to a size of a book when the global batch size and the sequence lengths are huge.

Fast recovering spikes

Loss spikes can happen often and as long as they quickly bounce back to where they left off the training usually continues as if nothing happened:

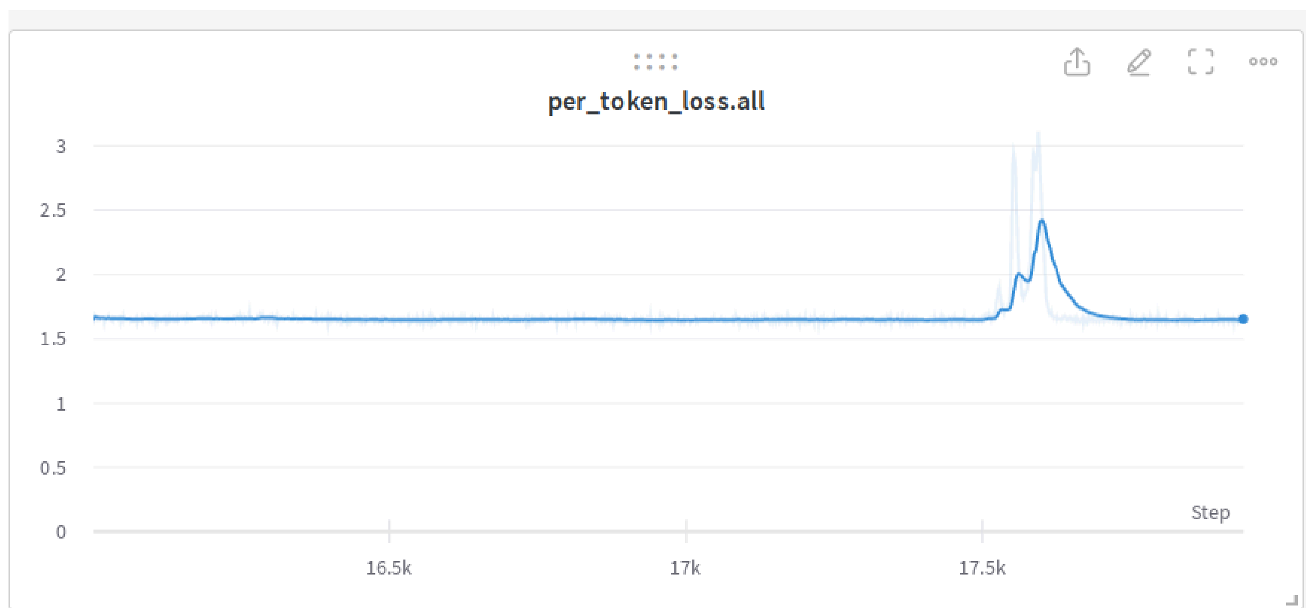
Here is an example of the 13B pre-BLOOM training experiment:



As you can see there are many spikes, some of a huge magnitude but they have all quickly recovered.

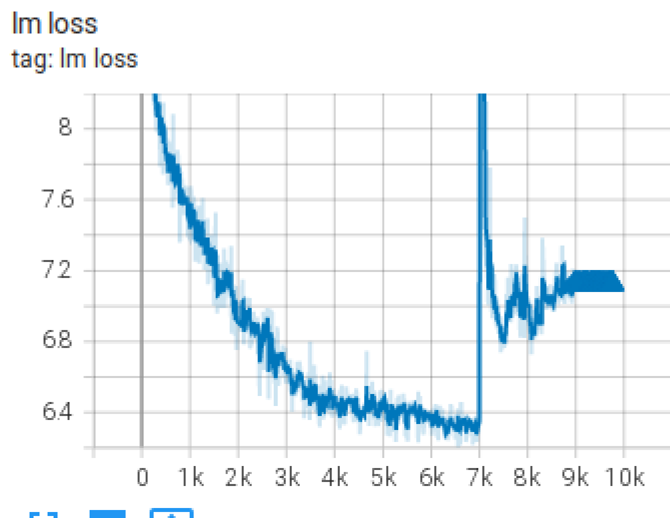
Slow recovering spikes

Here is a slow recovering spike from the IDEFICS-80B training:

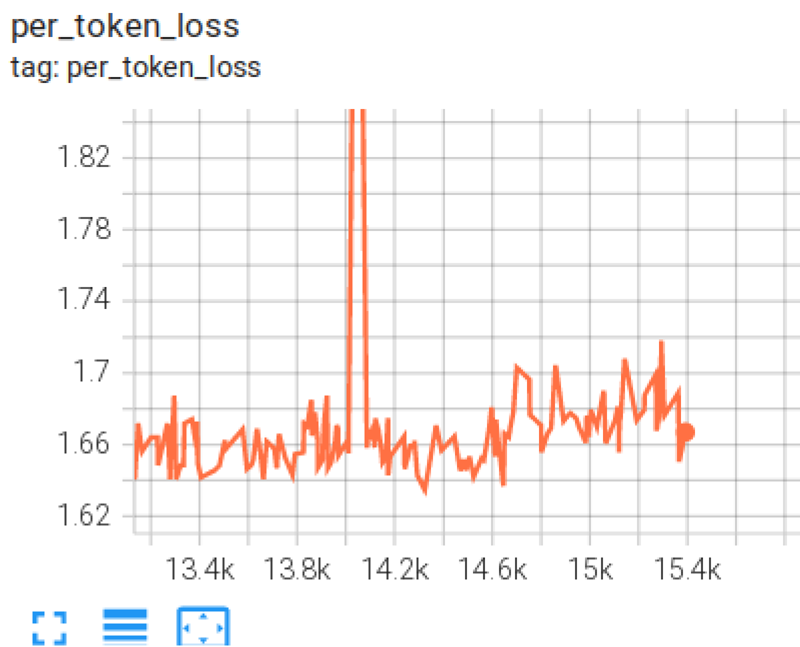


Not fully recovering spikes

This 104B model attempt spiked, started recovering but decided to not recover fully and instead started diverging

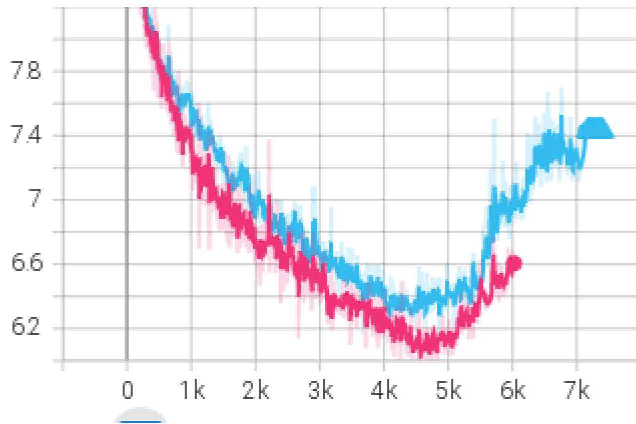


Here is another example from the IDEFICS-80B training:

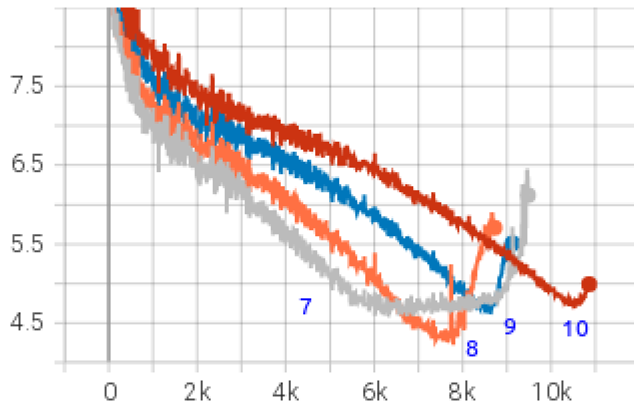


Non-spike diverging

Here are a few examples of diverging that didn't go through a spike



and here are a few more:

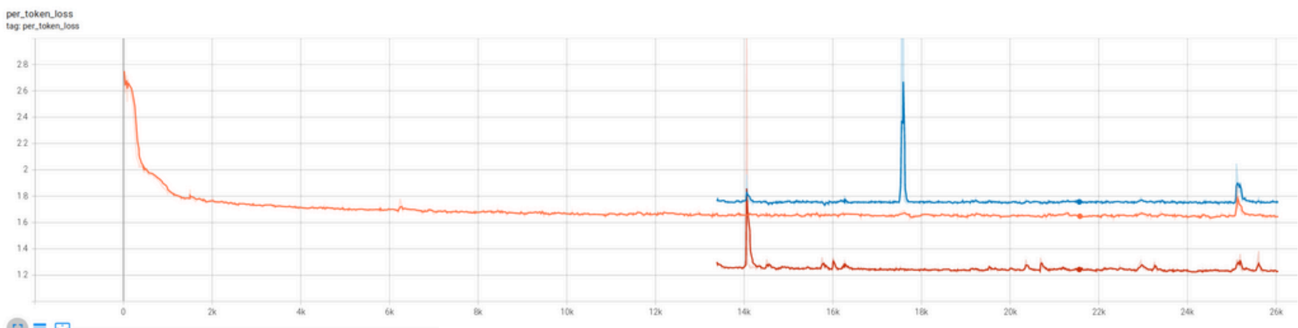


as you can see each restart makes a bit of progress and then the model diverges.

All these are from the 104B model attempts.

Multiple datasets spikes

During the IDEFICS-80B training we were using 2 different dataset types mixed together:



Legend: cm4 (high), average (mid) and pmd (low)

You can see that the loss spikes were sometimes happening simultaneously on both datasets and at other times only one

of the datasets loss would spike.

Here the model was learning two different data distributions and as you can see it was not reporting the same loss and the spike behaviors on both data distributions. The pmd datasets loss was much easier for the model than the cm4 one.

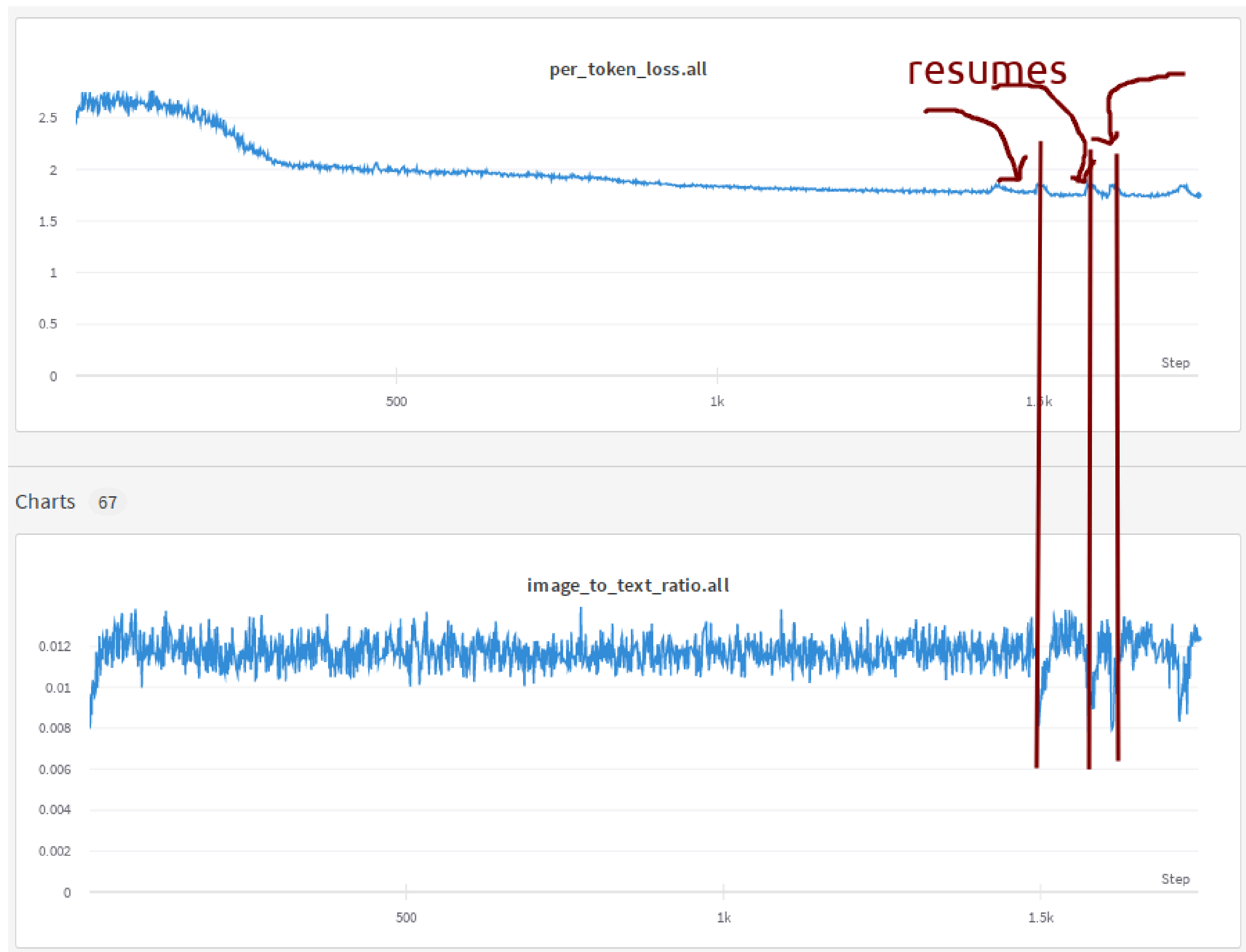
Resume-related spikes

Training resume due to a hardware crash or because a need to rollback to an earlier checkpoint due to encountering a divergence is pretty much guaranteed to happen. If your training software doesn't resume perfectly so that the model doesn't notice there was a resume various problems could be encountered.

The most complicated challenge of resume is restoring various RNGs, getting to the DataLoader index where the previous training was restored, and dealing with various other requirements if you use complex DataLoaders that are specific to your setup.

DataSampler related issues

During IDEFICS-80B training we had a very complicated DataLoader which was suffering from image to text ratio fluctuations when the DataLoader was getting restored on resume, so we ended up having a small spike on each resume which would then recover:

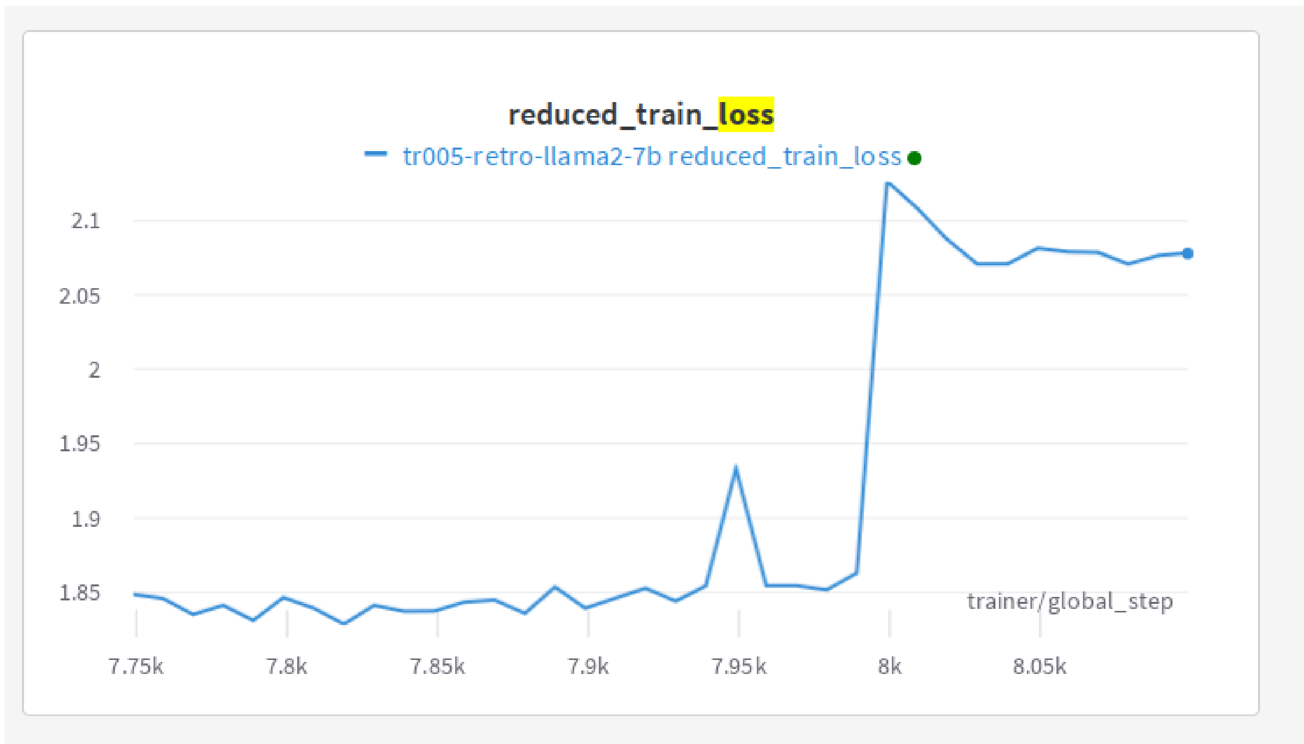


You can see the loss and ratio plots correlation here. As we had to resume about a dozen times we saw a lot of those spikes.

Impacts of repeat data

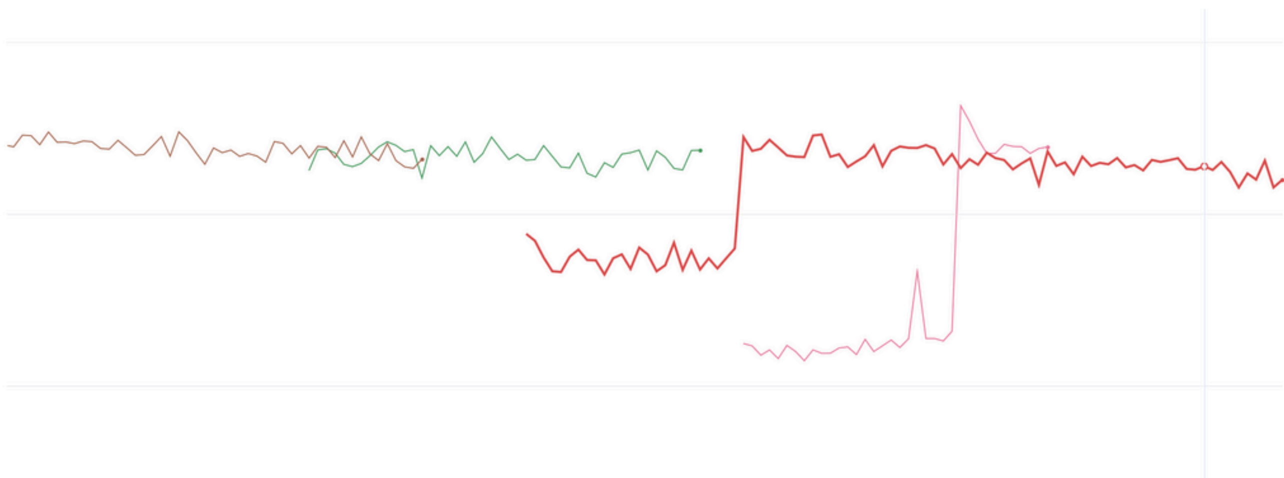
I was training a variation of Llama2 and saw this super unusual spike that didn't diverge or recover but which switched to a new higher loss level:

I rolled back to just before the weird behavior occurred and restarted. The loss training progressed at the same loss level for a bit and then again spiked and shifted to a higher loss.



I have never seen this type of divergence before. I was scratching my head for a while and then decided to look at the bigger picture.

As of this writing [Wandb](#) doesn't handle resume data plotting correctly if a rollback was performed, that is it ignores all new data after the rollback until the steps of the old data have been overcome. This forces us to start a new wandb plot for every resume with a rollback so that new data is shown. And if you needs to see the whole plot you have to stitch them and which includes dead data points that are no longer true. So I did the stitching and saw this puzzle:



There was no real spike in the two earlier runs. The loss never went up in the first place. In both resumes it was under-reporting loss due to an exactly repeated data and then it reached data it hasn't seen before and started reporting correctly. In other words it was overfitting and reporting a false loss.

The cause of the problem is data repetition, and since it clearly memorised some of it it was reporting a better loss.

The problem comes from [pytorch-lightning](#) not handling resumes correctly wrt DataSampler automatically - basically every time you resume you start your data stream from scratch. This, of course, requires a user to somehow fix the situation. You could change the seed to somewhat ameliorate the situation and avoid the exact data sequence, but it

still leaves you with repeat data, which isn't what you want for any serious training (or ablation experiments, since your observation will be invalid, if they assume IID data distribution.

footnote: I discussed this issue with the PTL developers and they said that they tried hard to come up with a generic solution but it wasn't meant to be. So the user needs to figure it out.

Make sure to check your training framework documentation whether it handles the DataSampler resuming correctly. Make sure you didn't discover this problem after the training has finished and you ended up training 6x times the same 50B of tokens from the planned 300B tokens seen only once each.

Doing a couple of resumes early on before embarking on the real training should also expose if there is a problem. Albeit, if the data gets reshuffled on each resume you are unlikely to see it. It'll only be seen if the seed is the same.

Checkpoints

- torch-checkpoint-convert-to-bf16 - converts an existing fp32 torch checkpoint to bf16. If safetensors are found those are converted as well. Should be easily adaptable to other similar use cases.
- torch-checkpoint-shrink.py - this script fixes checkpoints which for some reason stored tensors with storage larger than their view at the moment of saving. It clones the current view and re-saves them with just the storage of the current view.

Debugging and Troubleshooting

Guides

- Debugging PyTorch programs
- Diagnosing Hangings and Deadlocks in Multi-Node Multi-GPU Python Programs
- Troubleshooting NVIDIA GPUs
- Underflow and Overflow Detection
- NCCL Debug and Performance - notes for debugging NCCL-based software and tuning it up for the peak performance

Tools

- Debug Tools
- torch-distributed-gpu-test.py - this a `torch.distributed` diagnostics script that checks that all GPUs in the cluster (one or many nodes) can talk to each other and allocate gpu memory.
- NicerTrace - this is an improved `trace` python module with multiple additional flags added to the constructor and more useful output.

NCCL: Debug and Performance

Notes for debugging NCCL-based software and tuning it up for the peak performance

NCCL Environment Variables

The full list can be found [here](). That list is long but many of those variables are no longer in use.

Debug Environment Variables

The following env vars are most useful during debugging NCCL-related issues such as hanging and crashing.

`NCCL_DEBUG`

This is the most commonly used env var to debug networking issues.

Values:

- `VERSION` - Prints the NCCL version at the start of the program.
- `WARN` - Prints an explicit error message whenever any NCCL call errors out.
- `INFO` - Prints debug information
- `TRACE` - Prints replayable trace information on every call.

For example:

```
NCCL_DEBUG=INFO python -m torch.distributed.run --nproc_per_node 2 --nnodes 1
torch-distributed-gpu-test.py
```

This will dump a lot of NCCL-related debug information, which you can then search online if you find that some problems are reported.

And `NCCL_DEBUG_FILE` should be very useful when using `NCCL_DEBUG` as the information is copious especially if using many nodes.

`NCCL_DEBUG_FILE`

When using `NCCL_DEBUG` env var, redirect all NCCL debug logging output to a file.

The default is `stdout`. When using many GPUs it can be very useful to save each process' debug info into its own log file, which can be done like so:

```
NCCL_DEBUG_FILE=/path/to/nccl-log.%h.%p.txt
```

- `%h` is replaced with the hostname
- `%p` is replaced with the process PID.

If you then need to analyse hundreds of these at once, here are some useful shortcuts:

- grep for a specific match and also print the file and line number where it was found:

```
grep -n "Init COMPLETE" nccl-log*
```

- show `tail -1` of all nccl log files followed by the name of each file

```
find . -name "nccl*" -exec sh -c 'echo "$(tail -1 "$1") ($1)"' _ {} \;
```

`NCCL_DEBUG_SUBSYS`

`NCCL_DEBUG_SUBSYS` used in combination with `NCCL_DEBUG` tells the latter which subsystems to show. Normally you don't have to specify this variable, but sometimes the developers helping you may ask to limit the output to only some sub-systems, for example:

```
NCCL_DEBUG_SUBSYS=INIT,GRAPH,ENV,TUNING
```

`NCCL_P2P_DISABLE`

Disables P2P comms - e.g. NVLink won't be used if there is one and the performance will be much slower as a result of that.

`NCCL_SOCKET_IFNAME`

This one is very useful if you have multiple network interfaces and you want to choose a specific one to be used.

By default NCCL will try to use the fastest type of an interface, which is typically `ib` (InfiniBand).

But say you want to use an Ethernet interface instead then you can override with:

```
NCCL_SOCKET_IFNAME=eth
```

This env var can be used at times to debug connectivity issues, if say one of the interfaces is firewalled, and perhaps the others aren't and can be tried instead. Or if you are not sure whether some problem is related to the network interface or something else, so it helps to test other interfaces to invalidate that the issue comes from network.

Performance-Oriented Environment Variables

The following env vars are used primarily to tune up performance.

`NCCL_ALGO`

This one defines which algorithms NCCL will use. Typically it's one of tree, ring, collnetdirect and collnetchain.

I was asking questions about how a user can do the optimization and was told at [this NCCL Issue]() that basically the user shouldn't try to optimize anything as NCCL has a ton of smart algorithms inside that will try to automatically switch from one algorithm to another depending on a concrete situation.

Sylvain Jeaugey shared:

> There used to be a static threshold, but it's been replaced by a more complex tuning system. The new system builds a model of the latency and bandwidth of each algorithm/protocol combination (that's many, many combinations) and decides which one should perform best depending on the size. So there is no longer an env var and a static value, which is good because the performance of each algorithm depends on the number of nodes

> and number of GPUs per node and therefore we need to navigate a 2D space of algo/protocols which isn't easy. You can always force one algorithm with `NCCL_ALGO=TREE` and `NCCL_ALGO=RING` and see what performance you get and whether NCCL switches at the right point. I know it's hard to understand, but it's also the best solution we found to have the best performance across all platforms and users without users having to manually tune the switch points. Downside is, if you want to manually tune things, you can't.

If you use `NCCL_ALGO` you need to list the algorithms to consider, but otherwise you have no control over it. So, really, this is only useful if you want to make sure that one of the algorithms isn't used.

When asking about which algorithm is better, I received:

> Roughly speaking, ring is superior in terms of peak bandwidth (except on 2 nodes), tree is superior in terms of base latency (especially as we scale). `Bandwidth = Size / Time`, so whether you look at the time or the bandwidth for a given size, it will be a combination of both the peak bandwidth and the base latency. For a fixed size, as you scale, the base latency of ring will become prevalent and tree will be better.

`NCCL_CROSS_NIC`

The `NCCL_CROSS_NIC` variable controls whether NCCL should allow rings/trees to use different NICs, causing inter-node communication to use different NICs on different nodes.

To maximize inter-node communication performance when using multiple NICs, NCCL tries to communicate between same NICs between nodes, to allow for network design where each NIC from each node connects to a different network switch (network rail), and avoid any risk of traffic flow interference. The NCCL_CROSS_NIC setting is therefore dependent on the network topology, and in particular depending on whether the network fabric is rail-optimized or not.

This has no effect on systems with only one NIC.

Values accepted:

- 0: Always use the same NIC for the same ring/tree, to avoid crossing network rails. Suited for networks with per NIC switches (rails), with a slow inter-rail connection. Note there are corner cases for which NCCL may still cause cross-rail communication, so rails still need to be connected at the top.
- 1: Do not attempt to use the same NIC for the same ring/tree. This is suited for networks where all NICs from a node are connected to the same switch, hence trying to communicate across the same NICs does not help avoiding flow collisions.
- 2: (Default) Try to use the same NIC for the same ring/tree, but still allow for it if it would result in better performance.

Extrapolating benchmarks from several nodes to many

As it's often not easy to benchmark hundreds of nodes, often we try to benchmark interconnect performance using, say, 4 nodes. I wasn't sure whether this would give the correct indication for when 40 or 400 nodes will be used so I asked about it here and the answer was:

> Extrapolating at scale is not that hard for ring and tree (we have a function in `tuning.cc` predicting it, based on the ring linear latency and the tree log latency with reduced BW). Now as you scale, there are many factors which may cause your real performance to be very far off the prediction, like routing. Also note on an IB network you'll be able to use SHARP; that way your latency stays mostly constant as you scale, your bandwidth doesn't degrade much either, and you're always better than both ring and tree.

Counting NCCL calls

Enable NCCL debug logging for subsystems - collectives:

```
export NCCL_DEBUG=INFO
export NCCL_DEBUG_SUBSYS=COLL
```

if you're working in a slurm environment with many nodes you probably want to perform this only on rank `0`, like so:

```
if [[ $SLURM_PROCID == "0" ]]; then
  export NCCL_DEBUG=INFO
  export NCCL_DEBUG_SUBSYS=COLL
fi
```

Assuming your logs were all sent to `main_log.txt`, you can then count how many of each collective call were performed with:

```
grep -a "NCCL INFO Broadcast" main_log.txt     | wc -l
2590
grep -a "NCCL INFO AllReduce" main_log.txt     | wc -l
5207
grep -a "NCCL INFO AllGather" main_log.txt     | wc -l
1849749
grep -a "NCCL INFO ReduceScatter" main_log.txt | wc -l
82850
```

It might be a good idea to first isolate a specific stage of the training, as loading and saving will have a very different pattern from training iterations.

So I typically first slice out one iteration. e.g. if each iteration log starts with: `iteration: ...` then I'd first do:

```
csplit main_log.txt '/iteration: /' "{*}"
```

and then analyse one of the resulting files that correspond to the iterations. By default it will be named something like `xx02`.

Debugging PyTorch programs

Getting nodes to talk to each other

Once you need to use more than one node to scale your training, e.g., if you want to use DDP to train faster, you have to get the nodes to talk to each other, so that communication collectives could send data to each other. This is typically done via a comms library like NCCL. And in our DDP example, at the end of training step all GPUs have to perform an `all_reduce` call to synchronize the gradients across all ranks.

In this section we will discuss a very simple case of just 2 nodes (with 8 GPUs each) talking to each other and which can then be easily extended to as many nodes as needed. Let's say that these nodes have the IP addresses 10.0.0.1 and 10.0.0.2.

Once we have the IP addresses we then need to choose a port for communications.

In Unix there are 64k ports. The first 1k are reserved for common services so that any computer on the Internet could connect to any other computer knowing ahead of time which port to connect to. For example, port 22 is reserved for SSH. So that whenever you do `ssh example.com` in fact the program open a connection to `example.com:22`.

As there are thousands of services out there, the reserved 1k ports is not enough, and so various services could use pretty much any port. But fear not, when you get your Linux box on the cloud or an HPC, you're unlikely to have many preinstalled services that could use a high number port, so most ports should be available.

Therefore let's choose port 6000.

Now we have: `10.0.0.1:6000` and `10.0.0.2:6000` that we want to be able to communicate with each other.

The first thing to do is to open port `6000` for incoming and outgoing connections on both nodes. It might be open already or you might have to read up the instructions of your particular setup on how to open a given port.

Here are multiple ways that you could use to test whether port 6000 is already open.

```
telnet localhost:6000
nmap -p 6000 localhost
nc -zv localhost 6000
curl -v telnet://localhost:6000
```

Most of these should be available via `apt install` or whatever your package manager uses.

Let's use `nmap` in this example. If I run:

```
$ nmap -p 22 localhost
[...]
PORT   STATE SERVICE
22/tcp open  ssh
```

We can see the port is open and it tells us which protocol and service is allocated as a bonus.

Now let's run:

```
$ nmap -p 6000 localhost
```

```
[...]

PORT     STATE  SERVICE
6000/tcp closed X11
```

Here you can see port 6000 is closed.

Now that you understand how to test, you can proceed to test the `10.0.0.1:6000` and `10.0.0.2:6000`.

First ssh to the first node in terminal A and test if port 6000 is opened on the second node:

```
ssh 10.0.0.1
nmap -p 6000 10.0.0.2
```

if all is good, then in terminal B ssh to the second node and do the same check in reverse:

```
ssh 10.0.0.2
nmap -p 6000 10.0.0.1
```

If both ports are open you can now use this port. If either or both are closed you have to open these ports. Since most clouds use a proprietary solution, simply search the Internet for "open port" and the name of your cloud provider.

The next important thing to understand is that compute nodes will typically have multiple network interface cards (NICs). You discover those interfaces by running:

```
$ sudo ifconfig
```

One interface is typically used by users to connecting to nodes via ssh or for various other non-compute related services - e.g., sending an email or download some data. Often this interface is called `eth0`, with `eth` standing for Ethernet, but it can be called by other names.

Then there is the inter-node interface which can be Infiniband, EFA, OPA, HPE Slingshot, etc. ([more information](#)). There could be one or dozens of those interfaces.

Here are some examples of `ifconfig`'s output:

```
$ sudo ifconfig
enp5s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.23  netmask 255.255.255.0  broadcast 10.0.0.255
        [...]
```

I removed most of the output showing only some of the info. Here the key information is the IP address that is listed after `inet`. In the example above it's `10.0.0.23`. This is the IP address of interface `enp5s0`.

If there is another node, it'll probably be `10.0.0.24` or `10.0.0.21` or something of sorts - the last segment will be the one with a different number.

Let's look at another example:

```
$ sudo ifconfig
ib0     Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        inet addr:172.0.0.50  Bcast: 172.0.0.255  Mask:255.255.255.0
        [...]
```

Here `ib` typically tells us it's an InfiniBand card, but really it can be any other vendor. I have seen OmniPath using `ib` for example. Again `inet` tells us the IP of this interface is `172.0.0.50`.

If you lost me, we want the IP addresses so that we could test if ip:port is open on each node in question.

Finally, going back to our pair of `10.0.0.1:6000` and `10.0.0.2:6000` let's do an `all_reduce` test using 2 terminals, where we choose `10.0.0.1` as the master host which will coordinate other nodes. For testing we will use this helper debug program torch-distributed-gpu-test.py.

In terminal A:

```
$ ssh 10.0.0.1
$ python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node 8 \
 --master_addr 10.0.0.1 --master_port 6000 torch-distributed-gpu-test.py
```

In terminal B:

```
$ ssh 10.0.0.2
$ python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node 8 \
 --master_addr 10.0.0.1 --master_port 6000 torch-distributed-gpu-test.py
```

Note that I'm using the same `--master_addr 10.0.0.1 --master_port 6000` in both cases because we checked port 6000 is open and we use `10.0.0.1` as the coordinating host.

This approach of running things manually from each node is painful and so there are tools that automatically launch the same command on multiple nodes

pdsh

`pdsh` is one such solution - which is like `ssh` but will automatically run the same command on multiple nodes:

```
PDSH_RCMD_TYPE=ssh pdsh -w 10.0.0.1,10.0.0.2 \
"python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 2 --nproc_per_node 8 \
 --master_addr 10.0.0.1 --master_port 6000 torch-distributed-gpu-test.py"
```

You can see how I folded the 2 sets of commands into 1. If you have more nodes, just add more nodes as `-w` argument.

SLURM

If you use SLURM, it's almost certain that whoever set things up already have all the ports opened for you, so it should just work. But if it doesn't the information in this section should help debug things.

Here is how you'd use this with SLURM.

```
#!/bin/bash
```

```
#SBATCH --job-name=test-nodes        # name
#SBATCH --nodes=2                    # nodes
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=10           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#
export GPUS_PER_NODE=8
export MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
export MASTER_PORT=6000
#
srun --jobid $SLURM_JOBID bash -c 'python -m torch.distributed.run \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank $SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
torch-distributed-gpu-test.py'
```

If you have more than 2 nodes you just need to change the number of nodes and the above script will automatically work for any number of them.

MPI:

Another popular way is to use Message Passing Interface (MPI). There are a few open source implementations of it available.

To use this tool you first create a `hostfile` that contains your target nodes and the number of processes that should be run on each host. In the example of this section, with 2 nodes and 8 gpus each it'd be:

```
$ cat hostfile
10.0.0.1:8
10.0.0.2:8
```

and to run, it's just:

```
$ mpirun --hostfile  -np 16 -map-by ppr:8:node python my-program.py
```

Note that I used `my-program.py` here because torch-distributed-gpu-test.py was written to work with `torch.distributed.run` (also known as `torchrun`). With `mpirun` you will have to check your specific implementation to see which environment variable it uses to pass the rank of the program and replace `LOCAL_RANK` with it, the rest should be mostly the same.

Nuances:

- You might have to explicitly tell it which interface to use by adding `--mca btl_tcp_if_include 10.0.0.0/24` to match our example. If you have many network interfaces it might use one that isn't open or just the wrong interface.
- You can also do the reverse and exclude some interfaces. e.g. say you have `docker0` and `lo` interfaces - to exclude those add `--mca btl_tcp_if_exclude docker0,lo`.

`mpirun` has a gazillion of flags and I will recommend reading its manpage for more information. My intention was only to show you how you could use it. Also different `mpirun` implementations may use different CLI options.

Solving the Infiniband connection between multiple nodes

In one situation on Azure I got 2 nodes on a shared subnet and when I tried to run the 2 node NCCL test:

```
NCCL_DEBUG=INFO python -u -m torch.distributed.run --nproc_per_node=1 --nnodes 2 --rdzv_endpoint
10.2.0.4:6000  --rdzv_backend c10d torch-distributed-gpu-test.py
```

I saw in the debug messages that Infiniband interfaces got detected:

```
node-2:5776:5898 [0] NCCL INFO NET/IB : Using [0]ibP111p0s0:1/IB [1]rdmaP1111p0s2:1/RoCE [RO]; OOB
eth0:10.2.0.4<0>
```

But the connection would then time out with the message:

```
node-2:5776:5902 [0] transport/net_ib.cc:1296 NCCL WARN NET/IB : Got completion from peer 10.2.0.5<33092>
with error 12, opcode 0, len
0, vendor err 129 (Recv)
node-2:5776:5902 [0] NCCL INFO transport/net.cc:1134 -> 6
node-2:5776:5902 [0] NCCL INFO proxy.cc:679 -> 6
node-2:5776:5902 [0] NCCL INFO proxy.cc:858 -> 6 [Proxy Thread]
```

and nothing works. So here the Ethernet connectivity between 2 nodes works but not the IB interface.

There could be a variety of reason for this failing, but of the most likely one is when you're on the cloud and the 2 nodes weren't provisioned so that their IB is connected. So your Ethernet inter-node connectivity works, but it's too slow. Chances are that you need to re-provision the nodes so that they are allocated together. For example, on Azure this means you have to allocate nodes within a special availability set

Going back to our case study, once the nodes were deleted and recreated within an availability set the test worked out of the box.

The individual nodes are often not meant for inter-node communication and often the clouds have the concept of clusters, which are designed for allocating multiple nodes as a group and are already preconfigured to work together.

Prefixing logs with `node:rank`, interleaved asserts

In this section we will use `torchrun` (`torch.distributed.run`) during the demonstration and at the end of this section similar solutions for other launchers will be listed.

When you have warnings and tracebacks (or debug prints), it helps a lot to prefix each log line with its `hostname:rank` prefix, which is done by adding `--role $(hostname -s): --tee 3` to `torchrun`:

```
python -m torch.distributed.run --role $(hostname -s): --tee 3 --nnodes 1 --nproc_per_node 2 \
torch-distributed-gpu-test.py
```

Now each log line will be prefixed with `[hostname:rank]`

Note that the colon is important.

If you're in a SLURM environment the above command line becomes:

```
srun --jobid $SLURM_JOBID bash -c 'python -m torch.distributed.run \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank $SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
--role $(hostname -s): --tee 3 \
torch-distributed-gpu-test.py'
```

Of course adjust your environment variables to match, this was just an example.

Important! Note, that I'm using a single quoted string of commands passed to `bash -c`. This way `hostname -s` command is delayed until it's run on each of the nodes. If you'd use double quotes above, `hostname -s` will get executed on the starting node and then all nodes will get the same hostname as the prefix, which defeats the purpose of using these flags. So if you use double quotes you need to rewrite the above like so:

```
srun --jobid $SLURM_JOBID bash -c "python -m torch.distributed.run \
--nproc_per_node $GPUS_PER_NODE --nnodes $SLURM_NNODES --node_rank \$SLURM_PROCID \
--master_addr $MASTER_ADDR --master_port $MASTER_PORT \
--role \$(hostname -s): --tee 3 \
torch-distributed-gpu-test.py"
```

`$SLURM_PROCID` is escaped too as it needs to be specific to each node and it's unknown during the launch of the slurm job on the main node. So there are 2 `\$` escapes in this version of the command.

This prefixing functionality is also super-helpful when one gets the distributed program fail and which often results in interleaved tracebacks that are very difficult to interpret. So by `grep`ing for one `node:rank` string of choice, it's now possible to reconstruct the real error message.

For example, if you get a traceback that looks like:

```
  File "/path/to/training/dataset.py", line 785, in __init__
  File "/path/to/training/dataset.py", line 785, in __init__
    if self.dataset_proba.sum() != 1:
AttributeError: 'list' object has no attribute 'sum'
    if self.dataset_proba.sum() != 1:
  File "/path/to/training/dataset.py", line 785, in __init__
  File "/path/to/training/dataset.py", line 785, in __init__
    if self.dataset_proba.sum() != 1:
    if self.dataset_proba.sum() != 1:
AttributeError: 'list' object has no attribute 'sum'
AttributeError: 'list' object has no attribute 'sum'
AttributeError: 'list' object has no attribute 'sum'
```

and when it's dozens of frames over 8 nodes it can't be made sense of, but the above `-tee` + `--role` addition will generate:

```
[host1:0]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:1]  File "/path/to/training/dataset.py", line 785, in __init__
```

```
[host1:0]    if self.dataset_proba.sum() != 1:
[host1:0]AttributeError: 'list' object has no attribute 'sum'
[host1:1]    if self.dataset_proba.sum() != 1:
[host1:2]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:3]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:3]    if self.dataset_proba.sum() != 1:
[host1:2]    if self.dataset_proba.sum() != 1:
[host1:1]AttributeError: 'list' object has no attribute 'sum'
[host1:2]AttributeError: 'list' object has no attribute 'sum'
[host1:3]AttributeError: 'list' object has no attribute 'sum'
```

and you can `grep` this output for just one `host:rank` prefix, which gives us:

```
$ grep "[host1:0]" log.txt
[host1:0]  File "/path/to/training/dataset.py", line 785, in __init__
[host1:0]    if self.dataset_proba.sum() != 1:
[host1:0]AttributeError: 'list' object has no attribute 'sum'
```

and voila, you can now tell what really happened. And as I mentioned earlier there can be easily a hundred to thousands of interleaved traceback lines there.

Also, if you have just one node, you can just pass `-tee 3` and there is no need to pass `--role`.

If `hostname -s` is too long, but you have each host with its own sequence number like:

```
[really-really-really-long-hostname-5:0]
[really-really-really-long-hostname-5:1]
[really-really-really-long-hostname-5:2]
```

you can of course make it shorter by replacing `hostname -s` with `hostname -s | tr -dc '0-9'`, which would lead to much shorter prefixes:

```
[5:0]
[5:1]
[5:2]
```

And, of course, if you're doing debug prints, then to solve this exact issue you can use [`printflock`](printflock).

Here is how you accomplish the same feat with other launchers:

- `srun` in SLURM: add `--label`
- `openmpi`: add `--tag-output`
- `accelerate`: you can just pass the same `-tee` + `--role` flags as in `torchrun`

Dealing with Async CUDA bugs

When using CUDA, failing pytorch programs very often produce a python traceback that makes no sense or can't be acted upon. This is because due to CUDA's async nature - when a CUDA kernel is executed, the program has already moved on

and when the error happened the context of the program isn't there. The async functionality is there to make things faster, so that while the GPU is churning some `matmul` the program on CPU could already start doing something else.

At other times some parts of the system will actually tell you that they couldn't generate the correct traceback, as in this error:

```
[E ProcessGroupNCCL.cpp:414] Some NCCL operations have failed or timed out. Due to the
asynchronous nature of CUDA kernels, subsequent GPU operations might run on corrupted/
incomplete data. To avoid this inconsistency, we are taking the entire process down.
```

There are a few solutions.

If the failure is instant and can be reproduced on CPU (not all programs work on CPU), simply re-rerun it after hiding your GPUs. This is how you do it:

```
CUDA_VISIBLE_DEVICES="" python my-pytorch-program.py
```

The env var `CUDA_VISIBLE_DEVICES` is used to manually limit the visibility of GPUs to the executed program. So for example if you have 8 gpus and you want to run program1.py with first 4 gpus and program2.py with the remaining 2 gpus you can do:

```
CUDA_VISIBLE_DEVICES="0,1,2,3" python my-pytorch-program1.py
CUDA_VISIBLE_DEVICES="4,5,6,7" python my-pytorch-program2.py
```

and the second program won't be the wiser that it's not using GPUs 0-3.

But in the case of debug we are hiding all GPUs, by setting `CUDA_VISIBLE_DEVICES=""`.

Now the program runs on CPU and you will get a really nice traceback and will fix the problem in no time.

But, of course, if you your program requires multiple GPUs this won't work. And so here is another solution.

Rerun your program after setting this environment variable:

```
CUDA_LAUNCH_BLOCKING=1 python my-pytorch-program.py
```

This variable tells pytorch (or any other CUDA-based program) to turn its async nature off everywhere and now all operations will be synchronous. So when the program crashes you should now get a perfect traceback and you will know exactly what ails your program.

In theory enabling this variable should make everything run really slow, but in reality it really depends on your software. We did the whole of BLOOM-176B training using `CUDA_LAUNCH_BLOCKING=1` with `Megatron-Deepspeed`](https://github.com/bigscience-workshop/Megatron-DeepSpeed) and had zero slowdown - we had to use it as pytorch was hanging without it and we had no time to figure the hanging out.

So, yes, when you switch from async to sync nature, often it can hide some subtle race conditions, so there are times that a hanging disappears as in the example I shared above. So measure your throughput with and without this flag and sometimes it might actual not only help with getting an in-context traceback but actually solve your problem altogether.

Note: NCCL==2.14.3 coming with `pytorch==1.13` hangs when `CUDA_LAUNCH_BLOCKING=1` is used. So don't use it with that

version of pytorch. The issue has been fixed in `nccl>=2.17` which should be included in `pytorch==2.0`.

segfaults and getting a backtrace from a core file

It's not uncommon for a complex pytorch program to segfault and drop a core file. Especially if you're using complex extensions like NCCL.

The corefile is what the program generates when it crashes on a low-level - e.g. when using a python extension - such as a CUDA kernel or really any library that is coded directly in some variant of C or another language and made accessible in python through some binding API. The most common cause of a segfault is when such software accesses memory it has not allocated. For example, a program may try to free memory it hasn't allocated. But there could be many other reasons.

When a segfault event happens Python can't do anything, as the proverbial carpet is pulled out from under its feet, so it can't generate an exception or even write anything to the output.

In these situation one must go and analyse the libC-level calls that lead to the segfault, which is luckily saved in the core file.

If your program crashed, you will often find a file that will look something like: `core-python-3097667-6`

Before we continue make sure you have `gdb` installed:

```
sudo apt-get install gdb
```

Now make sure you know the path to the python executable that was used to run the program that crashed. If you have multiple python environment you have to activate the right environment first. If you don't `gdb` may fail to unpack the core file.

So typically I'd go:

```
conda activate my-env
gdb python core-python-3097667-6
```

- adjust `my-env` to whatever env you use, or instead of conda use whatever way you use to activate your python environment - and perhaps you're using the system-wise python and then you don't need to activate anything.
- adjust the name of the core file to the file you have gotten - it's possible that there are many - pick the latest then.

Now `gdb` will churn for a bit and will give you a prompt where you type: `bt`. We will use an actual core file here:

```
(gdb) bt
#0  0x0000147539887a9f in raise () from /lib64/libc.so.6
#1  0x000014753985ae05 in abort () from /lib64/libc.so.6
#2  0x000014751b85a09b in __gnu_cxx::__verbose_terminate_handler() [clone .cold.1] () from /lib64/
libstdc++.so.6
#3  0x000014751b86053c in __cxxabiv1::__terminate(void (*)()) () from /lib64/libstdc++.so.6
#4  0x000014751b860597 in std::terminate() () from /lib64/libstdc++.so.6
#5  0x000014751b86052e in std::rethrow_exception(std::__exception_ptr::exception_ptr) () from /lib64/
libstdc++.so.6
#6  0x000014750bb007ef in c10d::ProcessGroupNCCL::WorkNCCL::handleNCCLGuard() ()
   from .../python3.8/site-packages/torch/lib/libtorch_cuda_cpp.so
#7  0x000014750bb04c69 in c10d::ProcessGroupNCCL::workCleanupLoop() ()
```

```
   from.../python3.8/site-packages/torch/lib/libtorch_cuda_cpp.so
#8  0x000014751b88cba3 in execute_native_thread_routine () from /lib64/libstdc++.so.6
#9  0x000014753a3901cf in start_thread () from /lib64/libpthread.so.0
#10 0x0000147539872dd3 in clone () from /lib64/libc.so.6
```

and there you go. How do you make sense of it?

Well, you go from the bottom of the stack to the top. You can tell that a `clone` call was made in `libc` which then called `start_thread` in `libpthread` and then if you keep going there are a bunch of calls in the torch libraries and finally we can see that the program terminated itself, completing with `raise` from `libc` which told the Linux kernel to kill the program and create the core file.

This wasn't an easy to understand backtrace.

footnote: Yes, python calls it a *traceback* and elsewhere it's called a *backtrace* - it's confusing, but it's more or less the same thing.

Actually I had to ask pytorch devs for help and received:

- PyTorch `ProcessGroup` watchdog thread caught an asynchronous error from NCCL
- This error is an "`unhandled system error`" which in this particular case turned out to be an IB-OPA error
- The `ProcessGroup`'s `WorkCleanUp` thread rethrew the error so that the main process would crash and the user would get notified (otherwise this async error would not surface)

Trust me there are times when even if you're inexperienced the backtrace can give you enough of a hint to where you should look for troubleshooting.

But fear not - most of the time you won't need to understand the traceback. Ideally you'd just attach the core file to your filed Issue. But it can easily be 5GB large. So the developers that will be trying to help you will ask you to generate a `gdb` backtrace and now you know how to do that.

I didn't promise it'll be easy, I just showed you where to start.

Now another useful details is that many programs these days run multiple threads. And `bt` only shows the main thread of the process. But, often, it can be helpful to see where other threads in the process were when segfault has happened. For that you simply type 2 commands at the `(gdb)` prompt:

```
(gdb) thread apply all bt
(gdb) bt
```

and this time around you typically will get a massive report, one backtrace per thread.

py-spy

This is a super-useful tool for analysing hanging programs. For example, when a you have a resource deadlock or there is an issue with a network connection.

You will find an exhaustive coverage of this tool here.

strace

Similar to py-spy, `strace` is a super-useful tool which traces any running application at the low-level system calls - e.g. `libC` and alike.

For example, run:

```
strace python -c "print('strace')"
```

and you will see everything that is done at the system call level as the above program runs.

But usually it's more useful when you have a stuck program that spins all CPU cores at 100% but nothing happens and you want to see what's it doing. In this situation you simply attached to the running program like so:

```
strace --pid PID
```

where you get the PID for example from the output of `top` or `ps`. Typically I just copy-n-paste the PID of the program that consumes the most CPU - `top` usually shows it at the very top of its listing.

Same as `py-spy` you may need `sudo` perms to attached to an already running process - it all depends on your system setup. But you can always start a program with `strace` as I have shown in the original example.

Let's look at a small sub-snippet of the output of `strace python -c "print('strace')"`

```
write(1, "strace\n", 7strace
)                 = 7
```

Here we can see that a write call was executed on filedescriptor `1`, which almost always is `stdout` (`stdin` being 0, and `stderr` being 2).

If you're not sure what a filedescriptor is pointing to, normally you can tell from `strace`'s output itself. But you can also do:

```
ls -l /proc/PID/fd
```

where PID is the pid of the currently running program you're trying to investigate.

For example, when I run the above while running a pytest test with gpus, I got (partial output):

```
l-wx------ 1 stas stas 64 Mar  1 17:22 5 -> /dev/null
lr-x------ 1 stas stas 64 Mar  1 17:22 6 -> /dev/urandom
lrwx------ 1 stas stas 64 Mar  1 17:22 7 -> /dev/nvidiactl
lrwx------ 1 stas stas 64 Mar  1 17:22 8 -> /dev/nvidia0
lr-x------ 1 stas stas 64 Mar  1 17:22 9 -> /dev/nvidia-caps/nvidia-cap2
```

so you can see that a device `/dev/null` is open as FD (file descriptor) 5, `/dev/urandom` as FD 6, etc.

Now let's go look at another snippet from our `strace` run.

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

Here it tried to see if file `/etc/ld.so.preload` exists, but as we can see it doesn't - this can be useful if some shared library is missing - you can see where it's trying to load it from.

Let's try another one:

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=21448, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8028807000
mmap(0x7f8028808000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) =
0x7f8028808000
mmap(0x7f8028809000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f8028809000
mmap(0x7f802880a000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =
0x7f802880a000
close(3)
```

here we can see that it opens `/lib/x86_64-linux-gnu/libpthread.so.0` and assigns it FD 3, it then reads 832 chars from FD 3, (we can also see that the first chars are ELF - which stands for a shared library format), then memory maps it and closes that file.

In this following example, we see a python cached file is opened, its filepointer is moved to 0, and then it's read and closed.

```
openat(AT_FDCWD, "/home/stas/anaconda3/envs/py38-pt113/lib/python3.8/__pycache__/abc.cpython-38.pyc",
O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=5329, ...}) = 0
lseek(3, 0, SEEK_CUR)                   = 0
lseek(3, 0, SEEK_CUR)                   = 0
fstat(3, {st_mode=S_IFREG|0664, st_size=5329, ...}) = 0
brk(0x23bf000)                          = 0x23bf000
read(3, "U\r\r\n\0\0\0\0\24\216\177c\211\21\0\0\343\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 5330) = 5329
read(3, "", 1)                          = 0
close(3)
```

It's important to notice that file descriptors are re-used, so we have seen the same FD 3 twice, but each time it was open to a different file.

If your program is for example trying to reach to the Internet, you can also tell these calls from `strace` as the program would be reading from a socket file descriptor.

So let's run an example on a program that downloads files from the HF hub:

```
strace python -c 'import sys; from transformers import AutoConfig;
AutoConfig.from_pretrained(sys.argv[1])' t5-small
```

here is some relevant to this discussion snippet:

```
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC, IPPROTO_TCP) = 3
setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
```

```
ioctl(3, FIONBIO, [1])                  = 0
connect(3, {sa_family=AF_INET6, sin6_port=htons(443), sin6_flowinfo=htonl(0), inet_pton(AF_INET6,
"2600:1f18:147f:e850:e203:c458:10cd:fc3c
", &sin6_addr), sin6_scope_id=0}, 28) = -1 EINPROGRESS (Operation now in progress)
poll([{fd=3, events=POLLOUT|POLLERR}], 1, 10000) = 1 ([{fd=3, revents=POLLOUT}])
getsockopt(3, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
[...]
write(3, "\26\3\3\0F\20\0\0BA\4\373m\244\16\354/\334\205\361j\225\356\202m*\305\332\275\251\17J"..., 126)
= 126
read(3, 0x2f05c13, 5)                   = -1 EAGAIN (Resource temporarily unavailable)
poll([{fd=3, events=POLLIN}], 1, 9903)  = 1 ([{fd=3, revents=POLLIN}])
read(3, "\24\3\3\0\1", 5)               = 5
read(3, "\1", 1)                        = 1
read(3, "\26\3\3\0(", 5)                = 5
read(3, "\0\0\0\0\0\0\0\344\v\273\225`\4\24m\234~\371\332%l\364\254\34\3472<\0356s\313"..., 40) = 40
ioctl(3, FIONBIO, [1])                  = 0
poll([{fd=3, events=POLLOUT}], 1, 10000) = 1 ([{fd=3, revents=POLLOUT}])
write(3, "\27\3\3\1.\0\374$\361\217\337\377\264g\215\364\345\256\260\211$\326pkR\345\276,\321\221`-"...,
307) = 307
ioctl(3, FIONBIO, [1])                  = 0
read(3, 0x2ef7283, 5)                   = -1 EAGAIN (Resource temporarily unavailable)
poll([{fd=3, events=POLLIN}], 1, 10000) = 1 ([{fd=3, revents=POLLIN}])
```

You can see where that again it uses FD 3 but this time it opens a INET6 socket instead of a file. You can see that it then connects to that socket, polls, reads and writes from it.

There are many other super useful understandings one can derive from using this tool.

BTW, if you don't want to scroll up-down, you can also save the output to a file:

```
strace -o strace.txt python -c "print('strace')"
```

Now, since you're might want to strace the program from the very beginning, for example to sort out some race condition on a distributed filesystem, you will want to tell it to follow any forked processes. This what the `-f` flag is for:

```
strace -o log.txt -f python -m torch.distributed.run --nproc_per_node=4 --nnodes=1 --tee 3 test.py
```

So here we launch 4 processes and will end up running `strace` on at least 5 of them - the launcher plus 4 processes (each of which may spawn further child processes).

It will conveniently prefix each line with the pid of the program so it should be easy to tell which system was made by which process.

But if you want separate logs per process, then use `-ff` instead of `-f`.

The `strace` manpage has a ton of other useful options.

Invoke pdb on a specific rank in multi-node training

Once pytorch 2.2 is released you will have a new handy debug feature:

```
import torch.distributed as dist
[...]

def mycode(...):

   dist.breakpoint(0)
```

This is the same as `ForkedPdb` (below) but will automatically break for you on the rank of your choice - rank0 in the example above. Just make sure to call `up;;n` right away when the breakpoint hits to get into your normal code.

Here is what it does underneath:

```
import sys
import pdb

class ForkedPdb(pdb.Pdb):
    """
    PDB Subclass for debugging multi-processed code
    Suggested in: https://stackoverflow.com/questions/4716533/
how-to-attach-debugger-to-a-python-subproccess
    """
    def interaction(self, *args, **kwargs):
        _stdin = sys.stdin
        try:
            sys.stdin = open('/dev/stdin')
            pdb.Pdb.interaction(self, *args, **kwargs)
        finally:
            sys.stdin = _stdin


def mycode():

    if dist.get_rank() == 0:
        ForkedPdb().set_trace()
    dist.barrier()
```

so you can code it yourself as well.

And you can use that `ForkedPdb` code for normal forked applications, minus the `dist` calls.

Floating point math discrepancies on different devices

It's important to understand that depending on which device the floating point math is performed on the outcomes can be different. For example doing the same floating point operation on a CPU and a GPU may lead to different outcomes, similarly when using 2 different GPU architectures, and even more so if these are 2 different types of accelerators (e.g. NVIDIA vs. AMD GPUs).

Here is an example of discrepancies I was able to get doing the same simple floating point math on an 11 Gen Intel i7 CPU

and an NVIDIA A100 80GB (PCIe) GPU:

```
import torch

def do_math(device):
    inv_freq = (10 ** (torch.arange(0, 10, device=device) / 10))
    print(f"{inv_freq[9]:.20f}")
    return inv_freq.cpu()

a = do_math(torch.device("cpu"))
b = do_math(torch.device("cuda"))

torch.testing.assert_close(a, b, rtol=0.0, atol=0.0)
```

when we run it we get 2 out of 10 elements mismatch:

```
7.94328212738037109375
7.94328308105468750000
[...]
AssertionError: Tensor-likes are not equal!

Mismatched elements: 2 / 10 (20.0%)
Greatest absolute difference: 9.5367431640625e-07 at index (9,)
Greatest relative difference: 1.200604771156577e-07 at index (9,)
```

This was a simple low-dimensional example, but in reality the tensors are much bigger and will typically end up having more mismatches.

Now you might say that the `1e-6` discrepancy can be safely ignored. And it's often so as long as this is a final result. If this tensor from the example above is now fed through a 100 layers of `matmuls`, this tiny discrepancy is going to compound and spread out to impact many other elements with the final outcome being quite different from the same action performed on another type of device.

For example, see this discussion - the users reported that when doing Llama-2-7b inference they were getting quite different logits depending on how the model was initialized. To clarify the initial discussion was about Deepspeed potentially being the problem, but in later comments you can see that it was reduced to just which device the model's buffers were initialized on. The trained weights aren't an issue they are loaded from the checkpoint, but the buffers are recreated from scratch when the model is loaded, so that's where the problem emerges.

It's uncommon that small variations make much of a difference, but sometimes the difference can be clearly seen, as in this example where the same image is produced on a CPU and an MPS device.

To better illustrate the problem that motivated me to post this issue, I can present some output from CURL. This first image is from the CPU:

This is from the mps device:

This snapshot and the commentary come from this PyTorch Issue thread.

If you're curious where I pulled this code from - this is a simplified reduction of this original code in modeling_llama.py:

```
class LlamaRotaryEmbedding(nn.Module):
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()

        self.dim = dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2).float().to(device) / self.dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)
```

Debug Tools

git-related tools

Useful aliases

Show a diff of all files modified in the current branch against HEAD:

```
alias brdiff="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@');
git diff origin/\$def_branch..."
```

Same, but ignore white-space differences, adding `--ignore-space-at-eol` or `-w`:

```
alias brdiff-nows="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/
origin/@@'); git diff -w origin/\$def_branch..."
```

List all the files that were added or modified in the current branch compared to HEAD:

```
alias brfiles="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@');
git diff --name-only origin/\$def_branch..."
```

Once we have the list, we can now automatically open an editor to load just added and modified files:

```
alias bremacs="def_branch=\$(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@');
emacs \$(git diff --name-only origin/\$def_branch...) &"
```

git-bisect

(note to self: this is a sync from `the-art-of-debugging/methodology.md` which is the true source)

The discussed next approach should work for any revision control system that supports bisecting. We will use `git bisect` in this discussion.

`git bisect` helps to quickly find the commit that caused a certain problem.

Use case: Say, you were using `transformers==4.33.0` and then you needed a more recent feature so you upgraded to the bleed-edge `transformers@main` and your code broke. There could have been hundreds of commits between the two versions and it'd be very difficult to find the right commit that lead to the breakage by going through all the commits. Here is how you can quickly find out which commit was the cause.

footnote: HuggingFace Transformers is actually pretty good at not breaking often, but given its complexity and enormous size it happens nevertheless and the problems are fixed very quickly once reported. Since it's a very popular Machine Learning library it makes for a good debugging use case.

Solution: Bisecting all the commits between the known good and bad commits to find the one commit that's to blame.

We are going to use 2 shell terminals: A and B. Terminal A will be used for `git bisect` and terminal B for testing your software. There is no technical reason why you couldn't get away with a single terminal but it's easier with 2.

1. In terminal A fetch the git repo and install it in devel mode (`pip install -e .`) into your Python environment.

```
git clone https://github.com/huggingface/transformers
cd transformers
pip install -e .
```

Now the code of this clone will be used automatically when you run your application, instead of the version you previously installed from PyPi or Conda or elsewhere.

Also for simplicity we assume that all the dependencies have already been installed.

2. next we launch the bisecting - In terminal A, run:

```
git bisect start
```

3. Discover the last known good and the first known bad commits

`git bisect` needs just 2 data points to do its work. It needs to know one earlier commit that is known to work (`good`) and one later commit that is know to break (`bad`). So if you look at the sequence of commits on a given branch it'd have 2 known points and many commits around these that are of an unknown quality:

```
...... orig_good ..... .... .... .... ..... orig_bad ....
------------->--------------->----------------> time
```

So for example if you know that `transformers==4.33.0` was good and `transformers@main` (`HEAD`) is bad, find which commit is corresponding to the tag `4.33.0` by visiting the releases page and searching for `4.33.0`. We find that it was commit with SHA `5a4f340d`.

footnote: typically the first 8 hex characters are enough to have a unique identifier for a given repo, but you can use the full 40 character string.

So now we specify which is the first known good commit:

```
git bisect good 5a4f340d
```

and as we said we will use `HEAD` (latest commit) as the bad one, in which case we can use `HEAD` instead finding out the corresponding SHA string:

```
git bisect bad HEAD
```

If however you know it broke in `4.34.0` you can find its latest commit as explained above and use that instead of `HEAD`.

We are now all set at finding out the commit that broke things for you.

And after you told `git bisect` the good and the bad commits it has already switched to a commit somewhere in the middle:

```
...... orig_good ..... .... current .... .... ..... orig_bad ........
------------->--------------->---------------->----------------> time
```

You can run `git log` to see which commit it has switched to.

And to remind, we installed this repo as `pip install -e .` so the Python environment is instantly updated to the current commit's code version.

4. Good or bad

The next stage is telling `git bisect` if the current commit is `good` or `bad`:

To do so in terminal B run your program once.

Then in terminal A run:

```
git bisect bad
```

If it fails, or:

```
git bisect good
```

if it succeeds.

If, for example, if the result was bad, `git bisect` will internally flag the last commit as new bad and will half the commits again, switching to a new current commit:

```
...... orig_good ..... current .... new_bad .... ..... orig_bad ....
------------->--------------->---------------->----------------> time
```

And, vice versa, if the result was good, then you will have:

```
...... orig_good ..... .... new_good .... current ..... orig_bad ....
------------->--------------->---------------->----------------> time
```

5. Repeat until no more commits left

Keep repeating step 4 step until the problematic commit is found.

Once you finished bisecting, `git bisect` will tell you which commit was responsible for breaking things.

```
...... orig_good ..... .... last_good first_bad .... .. orig_bad ....
------------->--------------->---------------->----------------> time
```

If you followed the little commit diagrams, it'd correspond for the`first_bad` commit.

You can then go to `https://github.com/huggingface/transformers/commit/` and append the commit SHA to that url which will take you to the commit, (e.g. `https://github.com/huggingface/transformers/commit/`

`57f44dc4288a3521bd700405ad41e90a4687abc0` and which will then link to the PR from which it originated. And then you can ask for help by following up in that PR.

If your program doesn't take too long to run even if there are thousands of commits to search, you are facing `n` bisecting steps from `2**n` so 1024 commits can be searched in 10 steps.

If your program is very slow, try to reduce it to something small - ideally a small reproduction program that shows the problem really fast. Often, commenting out huge chunks of code that you deem irrelevant to the problem at hand, can be all it takes.

If you want to see the progress, you can ask it to show the current range of remaining commits to check with:

```
git bisect visualize --oneline
```

6. Clean up

So now restore the git repo clone to the same state you started from (most likely `HEAD) with:

```
git bisect reset
```

and possible reinstall the good version of the library while you report the issue to the maintainers.

Sometimes, the issue emerges from intentional backward compatibility breaking API changes, and you might just need to read the project's documentation to see what has changed. For example, if you switched from `transformers==2.0.0` to `transformers==3.0.0` it's almost guaranteed that your code will break, as major numbers difference are typically used to introduce major API changes.

7. Possible problems and their solutions:

a. skipping

If for some reason the current commit cannot be tested - it can be skipped with:

```
git bisect skip
```

and it `git bisect` will continue bisecting the remaining commits.

This is often helpful if some API has changed in the middle of the commit range and your program starts to fail for a totally different reason.

You might also try to make a variation of the program that adapts to the new API, and use it instead, but it's not always easy to do.

b. reversing the order

Normally git expects `bad` to be after `good`.

```
...... orig_good ..... .... .... .... ..... orig_bad ....
------------->--------------->---------------->----------------> time
```

Now, if `bad` happens before `good` revision order-wise and you want to find the first revision that fixed a previously existing problem - you can reverse the definitions of `good` and `bad` - it'd be confusing to work with overloaded logic states, so it's

recommended to use a new set of states instead - for example, `fixed` and `broken` - here is how you do that.

```
git bisect start --term-new=fixed --term-old=broken
git bisect fixed
git bisect broken 6c94774
```

and then use:

```
git fixed / git broken
```

instead of:

```
git good / git bad
```

c. complications

There are sometimes other complications, like when different revisions' dependencies aren't the same and for example one revision may require `numpy=1.25` and the other `numpy=1.26`. If the dependency package versions are backward compatible installing the newer version should do the trick. But that's not always the case. So sometimes one has to reinstall the right dependencies before re-testing the program.

Sometimes, it helps when there is a range of commits that are actually broken in a different way, you can either find a range of `good...bad` commits that isn't including the other bad range, or you can try to `git bisect skip` the other bad commits as explained earlier.

Diagnosing Hangings and Deadlocks in Multi-Node Multi-GPU Python Programs

While the methodologies found in this article were developed while working with multi-node multi-gpu pytorch-based training, they, of course, can help with any multi-process multi-node Python programs.

Helper tools

Try to use the following script torch-distributed-gpu-test.py to diagnose the situation.

This will help primarily with discovering network-related issues. And also to quickly understand how multi-gpu communications work.

For code-related issues read the rest of this document.

Approaches to diagnosing multi-gpu hanging / deadlocks

py-spy

First do `pip install py-spy`.

Now you can attach to each process with:

```
py-spy dump -n -p PID
```

and it will tell you where the process hangs (very often it's a nccl collective function or a `barrier`).

- `PID` is the process id of the hanging python process.
- `-n` is useful if you want to see stack traces from python extensions written in C, C++, etc., as the program may hang in one of the extensions
- you may need to add `sudo` before the command - for more details see this note.

If you have no `sudo` access your sysadmin might be able to perform this for you:

```
sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

which will allow you running `py-spy` (and `strace`) without needing `sudo`. Beware of the possible security implications - but typically if your compute node is inaccessible from the Internet it's less likely to be a risk.

To make this change permanent edit `/etc/sysctl.d/10-ptrace.conf` and set:

```
kernel.yama.ptrace_scope = 0
```

Here is an example of `py-spy dump` python stack trace:

```
Thread 835995 (active): "MainThread"
    broadcast (torch/distributed/distributed_c10d.py:1191)
    _aggregate_total_loss (deepspeed/runtime/pipe/engine.py:540)
    train_batch (deepspeed/runtime/pipe/engine.py:330)
    train_step (megatron/training.py:436)
    train (megatron/training.py:851)
    pretrain (megatron/training.py:187)
    <module> (pretrain_gpt.py:239)
```

The very first line is where the program is stuck.

If the hanging happens inside a CPP extension, add `--native` `py-spy` and it'll show the non-python code if any.

multi-process py-spy

Now, how do you do it for multiple processes. Doing it one-by-one is too slow. So let's do it at once.

If the launch command was `python`, what you do is:

```
pgrep -P $(pgrep -o python) | xargs -I {} py-spy dump --pid {}
```

if `deepspeed`:

```
pgrep -P $(pgrep -o deepspeed) | xargs -I {} py-spy dump --pid {}
```

for `accelerate`:

```
pgrep -P $(pgrep -o accelerate) | xargs -I {} py-spy dump --pid {}
```

you get the idea.

This particular approach will only analyse the main processes and not various other sub-processes/threads spawned by these processes. So if you have 8 gpus and 8 processes, the above will generate 8 stack traces.

If you want all processes and their subprocesses, then you'd just run:

```
pgrep -f python | xargs -I {} py-spy dump --pid {}
```

(and as before replace `python` with the name of the launcher program if it's not `python`)

multi-node py-spy via srun

What if you have multiple nodes?

You can of course `ssh` to each node interactively and dump the stack traces.

If you're using the SLURM environment you can use `srun` to do it on all nodes for you.

Now in another console get the `SLURM_JOBID` (or get it from `salloc` log):

```
squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"
```

Now use the following `srun` command after adjusting jobid with `SLURM_JOBID` from the outcome of the command above this sentence:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c 'ps aux | grep
python | egrep -v "grep|srun" | grep `whoami` | awk "{print \$2}" | xargs -I {} py-spy dump --native --pid
{}' || echo "failed"
```

Notes:

- One must use `--gres=gpu:0` for the monitor `srun` or otherwise it will block until the main `srun` (the one running the training) exits.
- Each node will generate its unique log file named `trace-nodename.out` - so this would help to identify which node(s) are problematic. You can remove `--output=trace-%N.out` if you want it all being dumped to stdout
- In some SLURM versions you may also need to add `--overlap`
- In some SLURM versions the jobid might not match that of reported in `squeue`, so you have to get the correct `SLURM_JOB_ID` from the logs of the job you're trying to "attach" to - i.e. your `srun` job that allocated the GPUs.
- Sometimes `bash` doesn't work, but `sh` does. I think it has to do with what dot files get `sourced`
- You might need to also activate a custom python environment, which you can do like so:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c 'conda
activate myenvname; ps auxc | ... ' || echo "failed"
```

or you can do it inside `~/.bashrc` or whatever shell's rc file you decide to use.

As mentioned before if you want just the main processes you'd use this instead:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c 'pgrep -P
$(pgrep -o python) | xargs -I {} py-spy dump --pid {}' || echo "failed"
```

Adjust `python` if need be as explained in the multi-gpu section above.

The previous longer command will deliver traces for all python processes.

If you're not getting anything, start with the basic debug like:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c 'date'
```

once you know you're talking to all the nodes, then you can progressively unravel the depth of calls, as in:

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'date'
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'pgrep -o python'
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'pgrep -P $(pgrep -o python) '
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 sh -c 'pgrep -P $(pgrep -o python) | xargs
-I {} py-spy dump --pid {}'
```

and at each stage check that the output makes sense - e.g. the 2nd and 3rd call you should be getting the PIDs of the processes.

multi-node py-spy via pdsh

`pdsh` seems to be a good easy tool to use to accomplish remote work on multiple nodes. Say, you're running on 2 nodes with hostnames `nodename-5` and `nodename-8`, then you can quickly test that remote execution is working by getting the `date` on all of these hosts with just:

```
$ PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] "date"
nodename-5: Wed Oct 25 04:32:43 UTC 2023
nodename-8: Wed Oct 25 04:32:45 UTC 2023
```

footnote: `pdsh` should be available via a normal OS package installer

Once you tested that `date` works it's time to move to `py-spy`.

To do `py-spy` on all python processes that are sub-processes, it'd be:

```
PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] 'pgrep -P $(pgrep -o python) | xargs -I {} py-spy dump --pid {}'
```

but as you're likely to need to have the `~/.bashrc` run, you will need to clone it into `~/.pdshrc`, reduce that clone to what is needed to be run (e.g. modify `PATH`, `activate conda`) and then `source` it, like:

```
PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] 'source ~/.pdshrc; pgrep -P $(pgrep -o python) | xargs -I {}
py-spy dump --pid {}"'
```

The reason you need a startup script is because usually `~/.bashrc` starts with:

```
# If not running interactively, don't do anything
case $- in
    *i*) ;;
      *) return;;
esac
```

so when you run such non-interactive workflows Bash won't process its `~/.bashrc` normally (exit early) and thus anything relying on this startup script won't work. So you can either remove the non-interactive exiting code above or fork `~/.bashrc` into a startup file that only contains what's needed for the remote command to succeed.

footnote: there is nothing special about `~/.pdshrc` - any other name would do, since you're manually `source`ing it.

And if your system isn't setup to run `py-spy` w/o `sudo` as explained a few sections up, you'd need something like this:

```
PDSH_RCMD_TYPE=ssh pdsh -w nodename-[5,8] 'sudo bash -c "source ~/.pdshrc; pgrep -P $(pgrep -o python) |
xargs -I {} py-spy dump --pid {}"'
```

Of course, you may need to edit the `pgrep` section to narrow down which processes you want to watch.

Additionally, to avoid being prompted with:

```
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

for every new node you haven't logged into yet, you can disable this check with:

```
echo "Host *" >> ~/.ssh/config
echo "  StrictHostKeyChecking no" >> ~/.ssh/config
```

Here I assume you're on an isolated cluster so you don't need to worry about security issues and thus bypassing such check is most likely OK.

multi-node py-spy via ds_ssh

This is yet another way, but please make sure to read the `pdsh` section above first.

The following notes require `pip install deepspeed`.

In one SLURM environment I also attempted using `pdsh` via `ds_ssh`, but somehow I wasn't able to run `py-spy` remotely - the main issue was that remote `ssh` command wasn't giving the same env as when I was logged in interactively via `ssh`. But if you have `sudo` access on the compute nodes then you could do:

First prepare `hostfile`:

```
function makehostfile() {
perl -e '$slots=split /,/, $ENV{"SLURM_STEP_GPUS"};
$slots=8 if $slots==0; # workaround 8 gpu machines
@nodes = split /\n/, qx[scontrol show hostnames $ENV{"SLURM_JOB_NODELIST"}];
print map { "$_ slots=$slots\n" } @nodes'
}
makehostfile > hostfile
```

Adapt `$slots` to the number of gpus per node. You may have to adapt this script if your `scontrol` produces a different output.

Now run the `py-spy` extraction command over all participating nodes:

```
ds_ssh -f hostfile "source ~/.pdshrc; ps aux | grep python | grep -v grep | grep `whoami` | awk '{print
\$2}' | xargs -I {} sudo py-spy dump --pid {} "
```

Notes:

- Put inside `~/.pdshrc` whatever init code that you may need to run. If you don't need any you can remove `source ~/.pdshrc;` from the command line.
- If you don't have it already `ds_ssh` is installed when you do `pip install deepspeed`.
- you might need to `export PDSH_RCMD_TYPE=ssh` if you get `rcmd: socket: Permission denied` error

Network-level hanging

The hanging could be happening at the network level. `NCCL_DEBUG=INFO` can help here.

Run the script with `NCCL_DEBUG=INFO` env var and try to study the outcome for obvious errors. It will tell you which device it's using, e.g.:

```
DeepWhite:21288:21288 [0] NCCL INFO NET/Socket : Using [0]enp67s0:192.168.50.21<0>
```

So it's using interface `enp67s0` over `192.168.50.21`

Is your `192.168.50.21` firewalled? or is it somehow a misconfigured network device?

Does it work if you use a loopback device `127.0.0.1`?

```
NCCL_DEBUG=INFO NCCL_SOCKET_IFNAME=lo python -m torch.distributed.run --nproc_per_node 4 --nnodes 1
torch-distributed-gpu-test.py
```

if not, see what other local network devices you have via `ifconfig` - try that instead of `lo` if any.

It's currently using `enp67s0` in the above example.

Isolate problematic GPUs

You can also try to see if only some GPUs fail

For example, does it work if you use the first 2 or the last 2 gpus:

```
CUDA_VISIBLE_DEVICES=0,1 python -m torch.distributed.run --nproc_per_node 2 --nnodes 1
torch-distributed-gpu-test.py
```

then the 2nd pair:

```
CUDA_VISIBLE_DEVICES=2,3 python -m torch.distributed.run --nproc_per_node 2 --nnodes 1
torch-distributed-gpu-test.py
```

python `trace`

Now what happens when the training doesn't just hang, but the hanging process stops responding? e.g. this happens when there is a serious hardware issue. But what if it is recurrent and `py-spy` won't help here, since it won't be able to attach to a process that is not responding.

So next came the idea of tracing all calls like one does with `strace(1)`, I researched python calls tracing facilities and have discovered that python has a `trace` sub-system.

The following code will trace all python calls and log them to the console and into a dedicated per process log file, via a custom `Tee` module I added.

This then can help to understand where some processes stopped responding, since we will have the log of the last call and all the previous calls before it went unresponsive.

```
$ cat train.py
[...]

def main():
    # [...]
    train()

import re
class Tee:
    """
    A helper class to tee print’s output into a file.
    Usage:
    sys.stdout = Tee(filename)
    """

    def __init__(self, filename):
        self.stdout = sys.stdout
        self.file = open(filename, "a")

    def __getattr__(self, attr):
        return getattr(self.stdout, attr)

    def write(self, msg):
        self.stdout.write(msg)
        self.file.write(msg)
        self.file.flush()

    def flush(self):
        self.stdout.flush()
        self.file.flush()

if __name__ == "__main__":

    import sys
    import trace
    import socket
    import os

    # enable the trace
    if 0:
        cwd = os.path.realpath(’.’)
        pid = os.getpid()
        hostname = socket.gethostname()
        local_rank = int(os.environ["LOCAL_RANK"])
        trace_output_file = f"{cwd}/trace-{hostname}-{local_rank}-{pid}.txt"

        # create a Trace object, telling it what to ignore, and whether to
        # do tracing or line-counting or both.
        tracer = trace.Trace(
            ignoredirs=[sys.prefix, sys.exec_prefix],
            trace=1,
```

```
            count=1,
            timing=True,
        )

        # run the new command using the given tracer
        sys.stdout = Tee(trace_output_file)
        tracer.run('main()')
    else:
        main()
```

This code doesn't require any special handing other than enabling the trace by changing `if 0` to `if 1`.

If you don't set `ignoredirs`, this will now dump all python calls. Which means expect a lot of GBs of data logged, especially if you have hundreds of GPUs.

Of course, you don't have to start tracing from `main` - if you suspect a specific are you can start tracing there instead and it'll be much faster and less data to save.

I wish I could tell `trace` which packages to follow, but alas it only supports dirs to ignore, which is much more difficult to set, and thus you end up with a lot more data than needrf. But still this is a super useful tool for debugging hanging processes.

Also, your code will now run much much slower and the more packages you trace the slower it will become.

NicerTrace

As `Trace` proved to provide very limited usability when debugging a complex multi-node multi-hour run crash, I have started on working on a better version of the `trace` python module.

You can find it here: [NicerTrace]()

I added multiple additional flags to the constructor and made the output much more useful. You fill find a full working example in that same file, just run:

```
python trace/NicerTrace.py
```

and you should see:

```
        trace/NicerTrace.py:1 <module>
0:00:00 <string>:     1:         trace/NicerTrace.py:185 main
0:00:00 NicerTrace.py:   186:     img = Image.new("RGB", (4, 4))
        PIL.Image:2896 new
0:00:00 Image.py:  2912:     _check_size(size)
        PIL.Image:2875 _check_size
0:00:00 Image.py:  2883:     if not isinstance(size, (list, tuple)):
0:00:00 Image.py:  2886:     if len(size) != 2:
0:00:00 Image.py:  2889:     if size[0] < 0 or size[1] < 0:
```

as you will see in the example I set:

```
            packages_to_include=["PIL"],
```

so it'll trace `PIL` plus anything that is not under `site-packages`. If you need to trace another package, just add it to that list.

This is a very fresh work-in-progress package, so it's evolving as we are trying to make it help us resolve a very complex crashing situation.

Working with generated trace files

When the per-node-rank trace files has been generated the following might be helpful to quickly analyse the situation:

- grep for a specific match and also print the file and line number where it was found:

```
grep -n "backward" trace*
```

- show `tail -1` of all trace files followed by the name of each file:

```
find . -name "trace*" -exec sh -c 'echo "$1: $(tail -3 "$1")"' _ {} \;
```

- or similar to the above, but print 5 last lines with the leading filename and some vertical white space for an easier reading:

```
find . -name "trace*" -exec sh -c 'echo; echo $1; echo "$(tail -5 "$1")"' _ {} \;
```

- count how many times grep matched a given pattern in each ifle and print the matched file (in this example matching the pattern `backward`):

```
find . -name "trace*" -exec sh -c 'echo "$1: $(grep "backward" $1 | wc -l)"' _ {} \;
```

good old `print`

Now once you discovered where the hanging happens to further understand why this is happening, a debugger would ideally be used, but more often than not debugging multi-process (multi-node) issues can be very difficult.

In such situations a good old `print` works. You just need to add some debug prints before the calls where things hang, things that would help understand what lead to the deadlock. For example, some `barrier` was missing and one or a few processes skipped some code and while the rest of processes are still blocking waiting for everybody to send some data (for example in NCCL collective functions like `gather` or `reduce`).

You of course, want to prefix each print with the rank of the process so that you could tell which is which. For example:

```
import torch.distributed as dist
print(f"{dist.get_rank()}: passed stage 0")
```

What you will quickly discover is that if you have multiple GPUs these prints will be badly interleaved and you will have a hard time making sense of the debug data. So let's fix this. We are going to override `print` with a custom version of the same, but which uses `flock` to ensure that only one process can write to stdout at the same time.

The helper module `printflock.py` is included here. To activate it just run this at the top of the module you're debugging:

```
from printflock import printflock as print
```

and now all your `print` calls in that module will magically be non-iterleaved. You can of course, just use `printflock` directly:

```
from printflock import printflock
import torch.distributed as dist
printflock(f"{dist.get_rank()}: passed stage 0")
```

core files

If the hanging happens inside non-python code, and `py-spy --native` isn't enough for some reason you can make the hanging program dump a core file, which is done with one of these approaches:

```
gcore <pid>
kill -ABRT <pid>
```

and then you can introspect the core file as explained here.

If you don't get the core file dumped you need to configure your system to allow so and also specify where the core files should be saved to.

To ensure the file is dumped in bash run (other shells may use a different command):

```
ulimit -c unlimited
```

To make this persistent run:

```
echo '* soft core unlimited' >> /etc/security/limits.conf
```

On some systems like Ubuntu the core files are hijacked by `apport`, check the contents of `/proc/sys/kernel/core_pattern` to see where they are sent. You can override where they are sent with:

```
sudo sysctl -w kernel.core_pattern=/tmp/core-%e.%p.%h.%t
```

Change the directory if you want to, but make sure that the user the program is running under can write to that directory. To make this change permanent edit `/etc/sysctl.conf` and add `kernel.core_pattern=/tmp/core-%e.%p.%h.%t` (or modify if it's already there).

footnote: see `man core` for all the different templates available

If on Ubuntu by default it sends core files to `apport`, which may save the core to `/var/lib/apport/coredump` or `/var/crash`. But you can change it explained above.

A quick way to test if your setup can generate a core file is:

```
sleep 10 &
killall -SIGSEGV sleep
```

Normally `SIGSEGV` isn't recommended for a real situation of diagnosing a hanging program, because `SIGSEGV` is likely to launch a sighandler, but for this test it's good enough.

Code loops

Code loops can be tricky to debug in hanging scenarios. If you have code like the following:

```
for i, d in enumerate(data):
    some_hanging_call(d)
```

it's possible that one process hangs in the first iteration, and another process in the second iteration, which makes things very confusing. But the stack trace won't give such indication, as the line numbers would be the same, even though the processes aren't in the same place code progression-wise.

In such situations unroll the loop to be:

```
d_iter = iter(data)
some_hanging_call(next(d_iter)
some_hanging_call(next(d_iter)
```

and now when you run `py-spy` the line numbers will be correct. The processes hanging in the first iteration will report the first `some_hanging_call` and those in the second iteration in the second call - as each now has its own line.

Hardware-specific issues

AMD/ROCm hangs or slow with IOMMU enabled

AMD Instinct users may need to either Disable IOMMU or set it to:

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=soft"
```

in `/etc/default/grub` (the grub config file could be elsewhere depending on the OS).

Disabling is `GRUB_CMDLINE_LINUX="amd_iommu=off"`

Underflow and Overflow Detection

For this section we are going to use the underflow_overflow library.

If you start getting `loss=NaN` or the model inhibits some other abnormal behavior due to `inf` or `nan` in activations or weights one needs to discover where the first underflow or overflow happens and what led to it. Luckily you can accomplish that easily by activating a special module that will do the detection automatically.

Let's use a `t5-large` model for this demonstration.

```
from .underflow_overflow import DebugUnderflowOverflow
from transformers import AutoModel

model = AutoModel.from_pretrained("t5-large")
debug_overflow = DebugUnderflowOverflow(model)
```

[`underflow_overflow.DebugUnderflowOverflow`] inserts hooks into the model that immediately after each forward call will test input and output variables and also the corresponding module's weights. As soon as `inf` or `nan` is detected in at least one element of the activations or weights, the program will assert and print a report like this (this was caught with `google/mt5-small` under fp16 mixed precision):

```
Detected inf/nan during batch_number=0
Last 21 forward frames:
abs min  abs max  metadata
                  encoder.block.1.layer.1.DenseReluDense.dropout Dropout
0.00e+00 2.57e+02 input[0]
0.00e+00 2.85e+02 output
[...]
                  encoder.block.2.layer.0 T5LayerSelfAttention
6.78e-04 3.15e+03 input[0]
2.65e-04 3.42e+03 output[0]
             None output[1]
2.25e-01 1.00e+04 output[2]
                  encoder.block.2.layer.1.layer_norm T5LayerNorm
8.69e-02 4.18e-01 weight
2.65e-04 3.42e+03 input[0]
1.79e-06 4.65e+00 output
                  encoder.block.2.layer.1.DenseReluDense.wi_0 Linear
2.17e-07 4.50e+00 weight
1.79e-06 4.65e+00 input[0]
2.68e-06 3.70e+01 output
                  encoder.block.2.layer.1.DenseReluDense.wi_1 Linear
8.08e-07 2.66e+01 weight
1.79e-06 4.65e+00 input[0]
1.27e-04 2.37e+02 output
                  encoder.block.2.layer.1.DenseReluDense.dropout Dropout
0.00e+00 8.76e+03 input[0]
```

```
0.00e+00 9.74e+03 output
                  encoder.block.2.layer.1.DenseReluDense.wo Linear
1.01e-06 6.44e+00 weight
0.00e+00 9.74e+03 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.DenseReluDense T5DenseGatedGeluDense
1.79e-06 4.65e+00 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.dropout Dropout
3.18e-04 6.27e+04 input[0]
0.00e+00      inf output
```

The example output has been trimmed in the middle for brevity.

The second column shows the value of the absolute largest element, so if you have a closer look at the last few frames, the inputs and outputs were in the range of `1e4`. So when this training was done under fp16 mixed precision the very last step overflowed (since under `fp16` the largest number before `inf` is `64e3`). To avoid overflows under `fp16` the activations must remain way below `1e4`, because `1e4 * 1e4 = 1e8` so any matrix multiplication with large activations is going to lead to a numerical overflow condition.

At the very start of the trace you can discover at which batch number the problem occurred (here `Detected inf/nan during batch_number=0` means the problem occurred on the first batch).

Each reported frame starts by declaring the fully qualified entry for the corresponding module this frame is reporting for. If we look just at this frame:

```
                  encoder.block.2.layer.1.layer_norm T5LayerNorm
8.69e-02 4.18e-01 weight
2.65e-04 3.42e+03 input[0]
1.79e-06 4.65e+00 output
```

Here, `encoder.block.2.layer.1.layer_norm` indicates that it was a layer norm for the first layer, of the second block of the encoder. And the specific calls of the `forward` is `T5LayerNorm`.

Let's look at the last few frames of that report:

```
Detected inf/nan during batch_number=0
Last 21 forward frames:
abs min  abs max  metadata
[...]
                  encoder.block.2.layer.1.DenseReluDense.wi_0 Linear
2.17e-07 4.50e+00 weight
1.79e-06 4.65e+00 input[0]
2.68e-06 3.70e+01 output
                  encoder.block.2.layer.1.DenseReluDense.wi_1 Linear
8.08e-07 2.66e+01 weight
1.79e-06 4.65e+00 input[0]
1.27e-04 2.37e+02 output
                  encoder.block.2.layer.1.DenseReluDense.wo Linear
1.01e-06 6.44e+00 weight
```

```
0.00e+00 9.74e+03 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.DenseReluDense T5DenseGatedGeluDense
1.79e-06 4.65e+00 input[0]
3.18e-04 6.27e+04 output
                  encoder.block.2.layer.1.dropout Dropout
3.18e-04 6.27e+04 input[0]
0.00e+00      inf output
```

The last frame reports for `Dropout.forward` function with the first entry for the only input and the second for the only output. You can see that it was called from an attribute `dropout` inside `DenseReluDense` class. We can see that it happened during the first layer, of the 2nd block, during the very first batch. Finally, the absolute largest input elements was `6.27e+04` and same for the output was `inf`.

You can see here, that `T5DenseGatedGeluDense.forward` resulted in output activations, whose absolute max value was around 62.7K, which is very close to fp16's top limit of 64K. In the next frame we have `Dropout` which renormalizes the weights, after it zeroed some of the elements, which pushes the absolute max value to more than 64K, and we get an overflow (`inf`).

As you can see it's the previous frames that we need to look into when the numbers start going into very large for fp16 numbers.

Let's match the report to the code from [`models/t5/modeling_t5.py`]():

```
class T5DenseGatedGeluDense(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.wi_0 = nn.Linear(config.d_model, config.d_ff, bias=False)
        self.wi_1 = nn.Linear(config.d_model, config.d_ff, bias=False)
        self.wo = nn.Linear(config.d_ff, config.d_model, bias=False)
        self.dropout = nn.Dropout(config.dropout_rate)
        self.gelu_act = ACT2FN["gelu_new"]

    def forward(self, hidden_states):
        hidden_gelu = self.gelu_act(self.wi_0(hidden_states))
        hidden_linear = self.wi_1(hidden_states)
        hidden_states = hidden_gelu * hidden_linear
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.wo(hidden_states)
        return hidden_states
```

Now it's easy to see the `dropout` call, and all the previous calls as well.

Since the detection is happening in a forward hook, these reports are printed immediately after each `forward` returns.

Going back to the full report, to act on it and to fix the problem, we need to go a few frames up where the numbers started to go up and most likely switch to the `fp32` mode here, so that the numbers don't overflow when multiplied or summed up. Of course, there might be other solutions. For example, we could turn off `amp` temporarily if it's enabled, after moving the original `forward` into a helper wrapper, like so:

```
import torch
```

```
def _forward(self, hidden_states):
    hidden_gelu = self.gelu_act(self.wi_0(hidden_states))
    hidden_linear = self.wi_1(hidden_states)
    hidden_states = hidden_gelu * hidden_linear
    hidden_states = self.dropout(hidden_states)
    hidden_states = self.wo(hidden_states)
    return hidden_states

def forward(self, hidden_states):
    if torch.is_autocast_enabled():
        with torch.cuda.amp.autocast(enabled=False):
            return self._forward(hidden_states)
    else:
        return self._forward(hidden_states)
```

Since the automatic detector only reports on inputs and outputs of full frames, once you know where to look, you may want to analyse the intermediary stages of any specific `forward` function as well. In such a case you can use the `detect_overflow` helper function to inject the detector where you want it, for example:

```
from underflow_overflow import detect_overflow


class T5LayerFF(nn.Module):
    [...]

    def forward(self, hidden_states):
        forwarded_states = self.layer_norm(hidden_states)
        detect_overflow(forwarded_states, "after layer_norm")
        forwarded_states = self.DenseReluDense(forwarded_states)
        detect_overflow(forwarded_states, "after DenseReluDense")
        return hidden_states + self.dropout(forwarded_states)
```

You can see that we added 2 of these and now we track if `inf` or `nan` for `forwarded_states` was detected somewhere in between.

Actually, the detector already reports these because each of the calls in the example above is a `nn.Module`, but let's say if you had some local direct calculations this is how you'd do that.

Additionally, if you're instantiating the debugger in your own code, you can adjust the number of frames printed from its default, e.g.:

```
from .underflow_overflow import DebugUnderflowOverflow

debug_overflow = DebugUnderflowOverflow(model, max_frames_to_save=100)
```

Specific batch absolute mix and max value tracing

The same debugging class can be used for per-batch tracing with the underflow/overflow detection feature turned off.

Let's say you want to watch the absolute min and max values for all the ingredients of each `forward` call of a given batch, and only do that for batches 1 and 3. Then you instantiate this class as:

```
debug_overflow = DebugUnderflowOverflow(model, trace_batch_nums=[1, 3])
```

And now full batches 1 and 3 will be traced using the same format as the underflow/overflow detector does.

Batches are 0-indexed.

This is helpful if you know that the program starts misbehaving after a certain batch number, so you can fast-forward right to that area. Here is a sample truncated output for such configuration:

```
                  *** Starting batch number=1 ***
abs min  abs max  metadata
                  shared Embedding
1.01e-06 7.92e+02 weight
0.00e+00 2.47e+04 input[0]
5.36e-05 7.92e+02 output
[...]
                  decoder.dropout Dropout
1.60e-07 2.27e+01 input[0]
0.00e+00 2.52e+01 output
                  decoder T5Stack
     not a tensor output
                  lm_head Linear
1.01e-06 7.92e+02 weight
0.00e+00 1.11e+00 input[0]
6.06e-02 8.39e+01 output
                   T5ForConditionalGeneration
     not a tensor output

                  *** Starting batch number=3 ***
abs min  abs max  metadata
                  shared Embedding
1.01e-06 7.92e+02 weight
0.00e+00 2.78e+04 input[0]
5.36e-05 7.92e+02 output
[...]
```

Here you will get a huge number of frames dumped - as many as there were forward calls in your model, so it may or may not what you want, but sometimes it can be easier to use for debugging purposes than a normal debugger. For example, if a problem starts happening at batch number 150. So you can dump traces for batches 149 and 150 and compare where numbers started to diverge.

You can also specify the batch number after which to stop the training, with:

```
debug_overflow = DebugUnderflowOverflow(model, trace_batch_nums=[1, 3], abort_after_batch_num=3)
```

Tensor precision / Data types

Quarter, Half and Mixed Precision

fp16

bf16

mixed fp16

mixed bf16

fp8

General OPs

`LayerNorm`-like operations must not do their work in half-precision, or they may lose a lot of data. Therefore when these operations are implemented correctly they do efficient internal work in fp32 and then their outputs are downcast to half-precision. Very often it's just the accumulation that is done in fp32, since adding up half-precision numbers is very lossy.

example:

Reduction collectives

fp16: ok to do in fp16 if loss scaling is in place

bf16: only ok in fp32

Gradient accumulation

best done in fp32 for both, but definitely for bf16

Optimizer step / Vanishing gradients

when adding a tiny gradient to a large number, that addition is often nullified

fp32 master weights and fp32 optim states

bf16 master weights and optim states can be done when using Kahan Summation and/or Stochastic rounding

Using fp16-pretrained model in bf16 regime

usually fails

Using bf16-pretrained model in fp16 regime

will lose some performance on conversion, but should work - best to finetune a bit

FP8

Main paper: [FP8 Formats for Deep Learning]

Reproducibility

Achieve determinism in randomness based software

When debugging always set a fixed seed for all the used Random Number Generators (RNG) so that you get the same data / code path on each re-run.

Though with so many different systems it can be tricky to cover them all. Here is an attempt to cover a few:

```
import random, torch, numpy as np
def enforce_reproducibility(use_seed=None):
    seed = use_seed if use_seed is not None else random.randint(1, 1000000)
    print(f"Using seed: {seed}")

    random.seed(seed)    # python RNG
    np.random.seed(seed) # numpy RNG

    # pytorch RNGs
    torch.manual_seed(seed)          # cpu + cuda
    torch.cuda.manual_seed_all(seed) # multi-gpu - can be called without gpus
    if use_seed: # slower speed! https://pytorch.org/docs/stable/notes/
randomness.html#cuda-convolution-benchmarking
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark     = False

    return seed
```

a few possible others if you use those subsystems/frameworks instead:

```
    torch.npu.manual_seed_all(seed)
    torch.xpu.manual_seed_all(seed)
    tf.random.set_seed(seed)
```

When you rerun the same code again and again to solve some problem set a specific seed at the beginning of your code with:

```
enforce_reproducibility(42)
```

But as it mentions above this is for debug only since it activates various torch flags that help with determinism but can slow things down so you don't want this in production.

However, you can call this instead to use in production:

```
enforce_reproducibility()
```

i.e. w/o the explicit seed. And then it'll pick a random seed and log it! So if something happens in production you can now reproduce the same RNGs the issue was observed in. And no performance penalty this time, as the `torch.backends.cudnn` flags are only set if you provided the seed explicitly. Say it logged:

```
Using seed: 1234
```

you then just need to change the code to:

```
enforce_reproducibility(1234)
```

and you will get the same RNGs setup.

As mentioned in the first paragraphs there could be many other RNGs involved in a system, for example, if you want the data to be fed in the same order for a `DataLoader` you need to have its seed set as well.

Additional resources:

- Reproducibility in pytorch

Reproduce the software and system environment

This methodology is useful when discovering some discrepancy in outcomes - quality or a throughput for example.

The idea is to log the key components of the environment used to launch a training (or inference) so that if at a later stage it needs to be reproduced exactly as it was it can be done.

Since there is a huge variety of systems and components being used it's impossible to prescribe a way that will always work. So let's discuss one possible recipe and you can then adapt it to your particular environment.

This is added to your slurm launcher script (or whatever other way you use to launch the training) - this is Bash script:

```
SAVE_DIR=/tmp # edit to a real path
export REPRO_DIR=$SAVE_DIR/repro/$SLURM_JOB_ID
mkdir -p $REPRO_DIR
# 1. modules (writes to stderr)
module list 2> $REPRO_DIR/modules.txt
# 2. env
/usr/bin/printenv | sort > $REPRO_DIR/env.txt
# 3. pip (this includes devel installs SHA)
pip freeze > $REPRO_DIR/requirements.txt
# 4. uncommitted diff in git clones installed into conda
perl -nle 'm|"file://(.*?/([^/]+))"| && qx[cd $1; if [ ! -z "\$(git diff)" ]; then git diff >
\$REPRO_DIR/$2.diff; fi]' $CONDA_PREFIX/lib/python*/site-packages/*.dist-info/direct_url.json
```

As you can see this recipe is used in a SLURM environment, so every new training will dump the environment specific to the SLURM job.

1. We save which `modules` were loaded, e.g. in cloud cluster/HPC setups you're like to be loading the CUDA and cuDNN libraries using this

   If you don't use `modules` then remove that entry

2. We dump the environment variables. This can be crucial since a single env var like `LD_PRELOAD` or `LD_LIBRARY_PATH` could make a huge impact on performance in some environments
3. We then dump the conda environment packages and their versions - this should work with any virtual python environment.
4. If you use a devel install with `pip install -e .` it doesn't know anything about the git clone repository it was installed from other than its git SHA. But the issue is that it's likely that you have modified the files locally and now `pip freeze` will miss those changes. So this part will go through all packages that are not installed into the conda environment (we find them by looking inside `site-packages/*.dist-info/direct_url.json`)

An additionally useful tool is conda-env-compare.pl which helps you to find out the exact differences `2` conda environments have.

Anecdotally, me and my colleague were getting very different training TFLOPs on a cloud cluster running the exact same code - literally launching the same slurm script from the same shared directory. We first compared our conda environments using conda-env-compare.pl and found some differences - I installed the exact packages she had to match her environment and it was still showing a huge performance difference. We then compared the output of `printenv` and discovered that I had `LD_PRELOAD` set up and she didn't - and that made a huge difference since this particular cloud provider required multiple env vars to be set to custom paths to get the most of their hardware.

HuggingFace Transformers notes

As this is one of the moment popular ML libraries this section will share some useful tools with HF `transformers` and others of their libraries.

(Disclaimer: I worked at HF for 3 years so I'm biased - for good :)

- Faster debug and development with tiny models, tokenizers and datasets
- Re-train hub models from scratch using finetuning examples

Faster debug and development with tiny models, tokenizers and datasets

If you're debugging problems and develop with full sized models and tokenizers you're likely not working in a very efficient way. Not only it's much more difficult to solve problem, the amount of waiting to get the program to restart and to get to the desirable point can be huge - and cumulatively this can be a huge drain on one's motivation and productivity, not talking about the resolution taking much longer, if at all.

The solution is simple:

Unless you're testing the quality of a model, always use a tiny random model with potentially tiny tokenizer.

Moreover, large models often require massive resources, which are typically expensive and can also can make a debugging process super complicated. For example any debugger can handle a single process, but if your model doesn't fit and require some sort of parallelization that requires multiple processes - most debuggers will either break or have issue giving you what you need. The ideal development environment is one process and a tiny model is guaranteed to fit on an even cheapest single smallest consumer GPU available. You could even use the free Google Colab to do development in a pinch if you have no GPUs around.

So the updated ML development mantra then becomes:

- the larger the model the better the final product generates
- the smaller the model the quicker the final product's training can be started

footnote: the recent research shows that larger isn't always better, but it's good enough to convey the importance of my communication.

Once your code is working, do switch to the real model to test the quality of your generation. But even in this case still try first the smallest model that produces a quality result. Only when you can see that the generation is mostly right use the largest model to validate if your work has been perfect.

Making a tiny model

Important: given their popularity and the well designed simple API I will be discussing HF `transformers` models. But the same principle can be applied to any other model.

TLDR: it's trivial to make a tiny HF `transformers` model:

1. Fetch the config object of a full size models
2. Shrink the hidden size and perhaps a few other parameters
3. Create a model from that shrunken config
4. Save this model. Done!

footnote: It's critical to remember that this will generate a random model, so don't expect any quality from its output.

Now let's go through the actual code and convert "google/mt5-small" into its tiny random counterpart.

```
from transformers import MT5Config, MT5ForConditionalGeneration

mname_from = "google/mt5-small"
mname_very_small = "mt5-tiny-random"
```

```
config = MT5Config.from_pretrained(mname_from)

config.update(dict(
    d_model=64,
    d_ff=256,
))
print("new config", config)

very_small_model = MT5ForConditionalGeneration(config)
print(f"num of params {very_small_model.num_parameters()}")

very_small_model.save_pretrained(mname_very_small)
```

As you can see it's trivial to do. And you can make it even smaller if you don't need the hidden size to be at least 64. For example try 8 - you just need to make sure that the number of attention heads isn't larger than hidden size.

Also please note that you don't need any GPUs to do that and you could do this even on a huge 176B parameter model like BLOOM-176B. Since you never load the actual original model, except its config object.

Before modifying the config you can dump the original parameters and choose to shrinks more dimensions. For example, using less layers makes it even smaller and easier to debug. So here is what you can do instead:

```
config.update(dict(
    vocab_size=keep_items+12,
    d_model=64,
    d_ff=256,
    d_kv=8,
    num_layers=8,
    num_decoder_layers=8,
    num_heads=4,
    relative_attention_num_buckets=32,
))
```

The original "google/mt5-small" model file was 1.2GB. With the above changes we got it down to 126MB.

We can then half its size by converting the model to fp16 (or bf16) before saving it:

```
very_small_model.half()
very_small_model.save_pretrained(mname_very_small)
```

this takes us to 64M file.

So you could stop here and your program will start much much faster already.

And there is one more step you could do to make it truly tiny.

What we haven't shrunken so far is the vocabulary dimension so 64x250k (hidden*vocab) is still huge. Granted this 250k vocab model is not typical - normally models models' vocab is ~30-50k, but even 30k is a lot if we want the model to be truly tiny.

So next we will look into various techniques to shrinking the tokenizer, as it defines our vocab size.

Making a tiny tokenizer

This task varies between a relatively simple procedure and a much more complex workout depending on the underlying tokenizer.

The following recipes have come from a few awesome tokenizer experts at Hugging Face, which I then adapted to my needs.

You probably don't really need to understand how these work until you actually need them, therefore if you're reading this for the first time you can safely jump over these to Making a tiny model with a tiny tokenizer.

Anthony Moi's version

Anthony Moi's tokenizer shrinker:

```
import json
from transformers import AutoTokenizer
from tokenizers import Tokenizer

vocab_keep_items = 5000
mname = "microsoft/deberta-base"

tokenizer = AutoTokenizer.from_pretrained(mname, use_fast=True)
assert tokenizer.is_fast, "This only works for fast tokenizers."
tokenizer_json = json.loads(tokenizer._tokenizer.to_str())
vocab = tokenizer_json["model"]["vocab"]
if tokenizer_json["model"]["type"] == "BPE":
    new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items }
    merges = tokenizer_json["model"]["merges"]
    new_merges = []
    for i in range(len(merges)):
        a, b = merges[i].split()
        new_token = "".join((a, b))
        if a in new_vocab and b in new_vocab and new_token in new_vocab:
            new_merges.append(merges[i])
    tokenizer_json["model"]["merges"] = new_merges
elif tokenizer_json["model"]["type"] == "Unigram":
    new_vocab = vocab[:vocab_keep_items]
elif tokenizer_json["model"]["type"] == "WordPiece" or tokenizer_json["model"]["type"] == "WordLevel":
    new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items }
else:
    raise ValueError(f"don't know how to handle {tokenizer_json['model']['type']}")
tokenizer_json["model"]["vocab"] = new_vocab
tokenizer._tokenizer = Tokenizer.from_str(json.dumps(tokenizer_json))
tokenizer.save_pretrained(".")
```

I later discovered that gpt2 seems to have a special token `"<|endoftext|>"` stashed at the very end of the vocab, so it gets dropped and code breaks. So I hacked it back in with:

```
if "gpt2" in mname:
```

```
        new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items-1 }
        new_vocab["<|endoftext|>"] = vocab_keep_items-1
    else:
        new_vocab = { token: i for token, i in vocab.items() if i < vocab_keep_items }
```

Lysandre Debut's version

Lysandre Debut' shrinker using `train_new_from_iterator`:

```
from transformers import AutoTokenizer

mname = "microsoft/deberta-base" # or any checkpoint that has a fast tokenizer.
vocab_keep_items = 5000

tokenizer = AutoTokenizer.from_pretrained(mname)
assert tokenizer.is_fast, "This only works for fast tokenizers."
tokenizer.save_pretrained("big-tokenizer")
# Should be a generator of list of texts.
training_corpus = [
    ["This is the first sentence.", "This is the second one."],
    ["This sentence (contains #) over symbols and numbers 12 3.", "But not this one."],
]
new_tokenizer = tokenizer.train_new_from_iterator(training_corpus, vocab_size=vocab_keep_items)
new_tokenizer.save_pretrained("small-tokenizer")
```

but this one requires a training corpus, so I had an idea to cheat and train the new tokenizer on its own original vocab which gave me:

```
from transformers import AutoTokenizer

mname = "microsoft/deberta-base"
vocab_keep_items = 5000

tokenizer = AutoTokenizer.from_pretrained(mname)
assert tokenizer.is_fast, "This only works for fast tokenizers."
vocab = tokenizer.get_vocab()
training_corpus = [ vocab.keys() ] # Should be a generator of list of texts.
new_tokenizer = tokenizer.train_new_from_iterator(training_corpus, vocab_size=vocab_keep_items)
new_tokenizer.save_pretrained("small-tokenizer")
```

which is almost perfect, except it now doesn't have any information about the frequency for each word/char (that's how most tokenizers compute their vocab, which if you need this info you can fix by having each key appearing `len(vocab) - ID times`, i.e.:

```
training_corpus = [ (k for i in range(vocab_len-v)) for k,v in vocab.items() ]
```

which will make the script much much longer to complete.

But for the needs of a tiny model (testing) the frequency doesn't matter at all.

Hack the tokenizer file approach

Some tokenizers can be be just manually truncated at the file level, e.g. let's shrink Llama2's tokenizer to 3k items:

```
# Shrink the orig vocab to keep things small (just enough to tokenize any word, so letters+symbols)
# ElectraTokenizerFast is fully defined by a tokenizer.json, which contains the vocab and the ids,
# so we just need to truncate it wisely
import subprocess
import shlex
from transformers import LlamaTokenizerFast

mname = "meta-llama/Llama-2-7b-hf"
vocab_keep_items = 3000

tokenizer_fast = LlamaTokenizerFast.from_pretrained(mname)
tmp_dir = f"/tmp/{mname}"
tokenizer_fast.save_pretrained(tmp_dir)
# resize tokenizer.json (vocab.txt will be automatically resized on save_pretrained)
# perl  -0777 -pi -e 's|(2999).*|$1},"merges": []}}|msg' tokenizer.json # 0-indexed, so
vocab_keep_items-1!
closing_pat = '},"merges": []}}'
cmd = (f"perl -0777 -pi -e 's|({vocab_keep_items-1}).*|$1{closing_pat}|msg' {tmp_dir}/tokenizer.json")
#print(f"Running:\n{cmd}")
result = subprocess.run(shlex.split(cmd), capture_output=True, text=True)
# reload with modified tokenizer
tokenizer_fast_tiny = LlamaTokenizerFast.from_pretrained(tmp_dir)
tokenizer_fast_tiny.save_pretrained(".")
```

Please remember that the outcome is only useful for functional testing - not quality work.

Here is the full version of [make_tiny_model.py] which includes both the model and the tokenizer shrinking.

SentencePiece vocab shrinking

First clone SentencePiece into a parent dir:

```
git clone https://github.com/google/sentencepiece
```

Now to the shrinking:

```
# workaround for fast tokenizer protobuf issue, and it's much faster too!
os.environ["PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION"] = "python"

from transformers import XLMRobertaTokenizerFast
```

```
mname = "xlm-roberta-base"

# Shrink the orig vocab to keep things small
vocab_keep_items = 5000
tmp_dir = f"/tmp/{mname}"
vocab_orig_path = f"{tmp_dir}/sentencepiece.bpe.model" # this name can be different
vocab_short_path = f"{tmp_dir}/spiece-short.model"
# HACK: need the sentencepiece source to get sentencepiece_model_pb2, as it doesn't get installed
sys.path.append("../sentencepiece/python/src/sentencepiece")
import sentencepiece_model_pb2 as model
tokenizer_orig = XLMRobertaTokenizerFast.from_pretrained(mname)
tokenizer_orig.save_pretrained(tmp_dir)
with open(vocab_orig_path, 'rb') as f: data = f.read()
# adapted from https://blog.ceshine.net/post/trim-down-sentencepiece-vocabulary/
m = model.ModelProto()
m.ParseFromString(data)
print(f"Shrinking vocab from original {len(m.pieces)} dict items")
for i in range(len(m.pieces) - vocab_keep_items): _ = m.pieces.pop()
print(f"new dict {len(m.pieces)}")
with open(vocab_short_path, 'wb') as f: f.write(m.SerializeToString())
m = None

tokenizer_fast_tiny = XLMRobertaTokenizerFast(vocab_file=vocab_short_path)
tokenizer_fast_tiny.save_pretrained(".")
```

Making a tiny model with a tiny tokenizer

So now you can shrink the vocab size to as small as the tokenizer allows, that is you need to have at least enough tokens to cover the target alphabet and special characters, and usually 3-5k tokens is more than enough. Sometimes you could make it even small, after all the original ASCII charset has only 128 characters.

If we continue the MT5 code from earlier in this chapter and add the tokenizer shrinking code from the previous section, we end up with this script mt5-make-tiny-model.py and when we run it - our end model file is truly tiny - 3.34 MB in size! As you can see the script also has code to validate that the model can actually work with the modified tokenizer. The results will be garbage, but the intention is to test that the new model and the tokenizer are functional.

Here is another example fsmt-make-super-tiny-model.py - here you can see I'm creating a totally new tiny vocab from scratch.

I also recommend to always store the building scripts with the model, so that you could quickly fix things or make similar versions of the model.

Also be aware that since HF `transformers` needs tiny models for their testing, you are very likely to already find one for each architecture available mostly from https://huggingface.co/hf-internal-testing (except they didn't include the code of how they were made, but you can now figure it out based on these notes).

Another hint: if you need a slightly different tiny model, you can also start with an already existing tiny model and adapt it instead. Since it's random it's really only about getting the right dimensions. For example if the tiny model you found has 2 layers but you need 8, just resave it with this larger dimension and you're done.

Making a tiny dataset

Similar to models and tokenizers it helps to have a handy tiny version of a dataset you work with a lot. As usual this won't

help with quality testing, but it's perfect for launching your program really fast.

footnote: the impact of using a tiny dataset won't be as massive as using a tiny model, if you're using already pre-indexed Arrow file datasets, since those are already extremely fast. But say you want the iterator to finish an epoch in 10 steps. Instead of editing your code to truncate the dataset, you could just use a tiny dataset instead.

This process of making a tiny dataset is somewhat more difficult to explain because it'd depend on the builder of the original model, which can be quite different from each other, but perhaps you can correlate my recipes to your datasets.

But the concept is still very simple:

1. Clone the full dataset git repo
2. Replace its full data tarball with a tiny one that contains just a few samples
3. Save it - Done!

Here are some examples:

- stas/oscar-en-10k
- stas/c4-en-10k
- stas/openwebtext-10k

In all of these I took the original tarball, grabbed the first 10k records, tarred it back, used this smaller tarball and that was that. The rest of the builder script remained mostly the same.

And here are some examples of synthetic datasets, where instead of just shrinking the original tarball, I untar'ed it, manually chose the representative examples and then wrote a script to build any size of desired dataset based on those few representative samples:

- stas/general-pmd-synthetic-testing and the unpacker
- stas/cm4-synthetic-testing - and the unpacker

These are also the complex examples where each sample is more than a text entry, but may have multiple text entries and images as well.

The unpacker is what expands each complex multi-record sample into its own sub-directory, so that now you can easily go and tweak it to your liking. You can add image, remove them, make text records smaller, etc.. You will also notice that I'm shrinking the large images into tiny 32x32 images, so again I'm applying the important principle of tiny across all dimensions that don't break the requirements of the target codebase.

And then the main script uses that structure to build a dataset of any desired length.

And here is for example the instructions of deploying these scripts for stas/general-pmd-synthetic-testing:

```
# prep dataset repo
https://huggingface.co/new-dataset => stas/general-pmd-synthetic-testing
git clone https://huggingface.co/datasets/stas/general-pmd-synthetic-testing
cd general-pmd-synthetic-testing

# select a few seed records so there is some longer and shorter text, records with images and without,
# a few variations of each type
rm -rf data
python general-pmd-ds-unpack.py --dataset_name_or_path \
general_pmd/image/localized_narratives__ADE20k/train/00000-00002 --ids 1-10 --target_path data

cd data

# shrink to 32x32 max, keeping ratio
```

```
mogrify -format jpg -resize 32x32\> */*jpg

# adjust one record to have no image and no text
cd 1
rm image.jpg text.txt
touch image.null text.null
cd -

cd ..

# create tarball
tar -cvzf data.tar.gz data

# complete the dataset repo
echo "This dataset is designed to be used in testing. It's derived from general-pmd/
localized_narratives__ADE20k \
dataset" >> README.md

# test dataset
cd ..
datasets-cli test general-pmd-synthetic-testing/general-pmd-synthetic-testing.py --all_configs
```

I also recommend to always store the building scripts with the dataset, so that you could quickly fix things or make similar versions of the dataset.

Similar to tiny models, you will find many tiny datasets under https://huggingface.co/hf-internal-testing.

Conclusion

While in the domain of ML we have the dataset, the model and the tokenizer - each of which can be made tiny and enable super-speed development with low resource requirements, if you're coming from a different industry you can adapt the ideas discussed in this chapter to your particular domain's artifacts/payloads.

Backup of all scripts in this chapter

Should the original scripts this chapter is pointing to disappear or the HF hub is down while you're reading this, here is the local back up of all of them.

note-to-self: to make the latest backup of files linked to in this chapter run:

```
perl -lne 'while (/(https.*?.py)\)/g) { $x=$1; $x=~s/blob/raw/; print qq[wget $x] }' make-tiny-models.md
```

Re-train Hub Models From Scratch Using Finetuning Examples

HF Transformers has awesome finetuning examples https://github.com/huggingface/transformers/tree/main/examples/pytorch, that cover pretty much any modality and these examples work out of box.

But what if you wanted to re-train from scratch rather than finetune.

Here is a simple hack to accomplish that.

We will use `facebook/opt-1.3b` and we will plan to use bf16 training regime as an example here:

```
cat << EOT > prep-bf16.py
from transformers import AutoConfig, AutoModel, AutoTokenizer
import torch

mname = "facebook/opt-1.3b"

config = AutoConfig.from_pretrained(mname)
model = AutoModel.from_config(config, torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained(mname)

path = "opt-1.3b-bf16"

model.save_pretrained(path)
tokenizer.save_pretrained(path)
EOT
```

now run:

```
python prep-bf16.py
```

This will create a folder: `opt-1.3b-bf16` with everything you need to train the model from scratch. In other words you have a pretrained-like model, except it only had its initializations done and none of the training yet.

Adjust to script above to use `torch.float16` or `torch.float32` if that's what you plan to use instead.

Now you can proceed with finetuning this saved model as normal:

```
python -m torch.distributed.run \
--nproc_per_node=1 --nnode=1 --node_rank=0 \
--master_addr=127.0.0.1 --master_port=9901 \
examples/pytorch/language-modeling/run_clm.py --bf16 \
--seed 42 --model_name_or_path opt-1.3b-bf16 \
--dataset_name wikitext --dataset_config_name wikitext-103-raw-v1 \
--per_device_train_batch_size 12 --per_device_eval_batch_size 12 \
```

```
--gradient_accumulation_steps 1 --do_train --do_eval --logging_steps 10 \
--save_steps 1000 --eval_steps 100 --weight_decay 0.1 --num_train_epochs 1 \
--adam_beta1 0.9 --adam_beta2 0.95 --learning_rate 0.0002 --lr_scheduler_type \
linear --warmup_steps 500 --report_to tensorboard --output_dir save_dir
```

The key entry being:

```
--model_name_or_path opt-1.3b-bf16
```

where `opt-1.3b-bf16` is your local directory you have just generated in the previous step.

Sometimes it's possible to find the same dataset that the original model was trained on, sometimes you have to use an alternative dataset.

The rest of the hyper-parameters can often be found in the paper or documentation that came with the model.

To summarize, this recipe allows you to use finetuning examples to re-train whatever model you can find on [the hub](https://huggingface.co/models).

A Back up of scripts

This is a backup of scripts discussed in Faster debug and development with tiny models, tokenizers and datasets.

- c4-en-10k.py
- cm4-synthetic-testing.py
- fsmt-make-super-tiny-model.py
- general-pmd-ds-unpack.py
- general-pmd-synthetic-testing.py
- m4-ds-unpack.py
- mt5-make-tiny-model.py
- openwebtext-10k.py
- oscar-en-10k.py

Resources

Publicly available training LLM/VLM logbooks

Logbooks and chronicles of training LLM/VLM are one of the best sources to learn from about dealing with training instabilities and choosing good hyper parameters.

If you know of a public LLM/VLM training logbook that is not on this list please kindly let me know or add it via a PR. Thank you!

The listing is in no particular order other than being grouped by the year.

2021

- BigScience pre-BLOOM 108B training experiments (2021): chronicles | the full spec and discussions (backup: 1 | 2)

2022

- BigScience BLOOM-176B (2022): chronicles-prequel | chronicles | the full spec and discussions (backup: 1 | 2 | 3)
- Meta OPT-175B (2022): logbook | Video (backup: 1)
- THUDM GLM-130B (2022): en logbook | Mandarin version (backup: 1 | 2)

2023

- HuggingFace IDEFICS-80B multimodal (Flamingo repro) (2023): Learning log | Training Chronicles (backup: 1 | 2)
- BloombergGPT 50B LLM - section C in BloombergGPT: A Large Language Model for Finance

Book Building

Important: this is still a WIP - it mostly works, but stylesheets need some work to make the pdf really nice. Should be complete in a few weeks.

This document assumes you're working from the root of the repo.

Installation requirements

1. Install python packages used during book build

```
pip install -r build/requirements.txt
```

2. Download the free version of [Prince XML](). It's used to build the pdf version of this book.

Build html

```
make html
```

Build pdf

```
make pdf
```

It will first build the html target and then will use it to build the pdf version.

Check links and anchors

To validate that all local links and anchored links are valid run:

```
make check-links-local
```

To additionally also check external links

```
make check-links-all
```

use the latter sparingly to avoid being banned for hammering servers.

Move md files/dirs and adjust relative links

e.g. `slurm` => `orchestration/slurm`

```
src=slurm
dst=orchestration/slurm

mkdir -p orchestration
git mv $src $dst
perl -pi -e "s|$src|$dst|" chapters-md.txt
python build/mdbook/mv-links.py $src $dst
git checkout $dst
make check-links-local
```