Best Practices in ML

Dr. Dimitris Dellis
ntell [at] grnet.gr

GRNET

Athens, 11-12 June 2020

Many files I

- Typical in training phase, many files are used (commonly images, text, sound, video etc.)
- In genereal inefficient for large numbers. Typical cases use order of millions files.
- Even super fast parallel filesystems can not cope with many jobs using these numbers.
- Possibilities for Performance Enhancement.
  - Local Scratch (SSD ?)
    - Have your files in few tarballs, zip etc.

Many files II

- Inside Job script, copy the tarball(s) to the machine's scratch, untar, use them, remove them. All input/output still in your permanent workspace.
- Pros : Faster but at least a factor of two, no shared filesystem load, no network etc.
- Cons : Local scratch that is SSD is not always available. Limited to single node runs.

Many files III

- Put all files in few large files from where you can read from (typically) Python.
  If the file format supports parallel access much better.
  Keep these large files in your workspace.
  No significant overhead for parallel filesystem - no meta data operations, only streaming. One mature, well tested solution is the use of HDF5 format used in many similar situations (Climate science).

HDF5 I

- Before run, pack data in (one file as example) HDF5, using Python, probably resize or other operations before store in case of images.
- Do not forget after HDF5 creation to remove (if you have them in zip/tar, or pack in tar/zip)
- Switch to interactive source code.

HDF5 I

- Some results :
- 1000 images

Operation	Files on Disk	HDF5
Read	13.56	1.09
Write	8.59	0.90

Batch size

- GPU (and CPU) memory is limited.
- Batches of processing. Probably you already use it.
- Calculate maximum batch size from available memory and your data size.
- Remember : Offloading data to gpu has maximum performance when you send small number of batches of size almost up to fill the memory.
- GPUs are very fast and they need enough data to work with efficiency.
  Transfer to/from GPU is expensive.
  Many small transfers : Lower bandwidht, High percentage of GPU idle due to no data to work with.

Parallelization I

- Check parallelization frameworks, select mature, well tuned - not the first from internet.
  - For Single node, more frameworks are available, not all appropriate for HPC, not all efficient.
  - Many ignore the concept of allocated resources and try to use whole node cores. It may be acceptable on a desktop but not for shared resources.
  - Limited frameworks that work efficiently with more than one node.
  - Many of these use master/slave architecture, using sockets on usually ethernet - ignoring the high speed interfaces that should be used in HPC environment.

Parallelization II

- Horovod looks to be mature enough, it uses MPI for IPC communication (that uses shared memory for intra and High speed for internode communication.
- Wheels vs build, drivers compilers
  - With Python there are few possibilities to install packages.
  - pip : looks for package, downloads, install.
    - It is fast and easy.
    - Two types of packages : Source that are compiled on install machine or wheel precompiled packages.
    - One should be lucky enough in order to work. If it works, probably with performance penalty due to portability.
  - Install from Source.

Parallelization III

- Not the easiest way.
- You compile with software stack existing on machine, optimized for this machine.
- Probably you need to specify the compiler flags and probably library paths to efficiently use the hardware.

- Speed up GPU Feed to efficiently use resources
  - Use multiprocessing to use GPU in parallel

Containers

- What Containers are ?
- Well known containers : Docker and Singularity.
- Pros and Cons of Docker and Singularity.
- Use of singularity is probably the only portable choice.
- And then ?
- You have the pay the performance fee for portability.
- A fully portable image is agnostic for existing hardware, it should run on old cpus, with recent cpu features (that give the performance of recent cpus) ignored, use the maximum portable compute capability and parameters of GPUs,
- Typically, less than half the performance of native builds.
- Container images optimized for recent hardware, probably

Use of resources

- More than 1 GPU : Doing nothing in code usually does nothing on others except reserve.
- There are many ways and tools to split work across gpus.
- nvidia-smi or tensorboard to check gpu usage and tune details of use.