Hands-on Tutorial On Efficient Use of the Cluster Misha with Focus on AI/ML Applications

Center for Neurocomputation and Machine Intelligence
Wu Tsai Institute
Yale University

Awni Altabaa, Ping Luo

10/30/2024

Agenda

- Introduction to Misha
- Introduction to W&B
- PyTorch performance tuning
- Checkpointing and restarting

What will be covered

- Introduction to Misha
  - Setups on Misha that are different from the other YCRC clusters
    - Multi-instance GPU (MIG) node in the 'gpu_devel' partition
    - QoS-based SLURM GPU allocation protocol
  - General tips and tricks on efficient use of the cluster
- Experiment tracking and reproducibility with Weights and Biases
- PyTorch optimization
  - Compilation and Automatic Mixed Precision
- Checkpointing and restarting
  - Principle
  - The CPL Python library

What will not be covered

- Topics that are important but not covered here
  - Intro to Linux
  - Intro to HPC
- Attend YCRC hands-on tutorials or watch their pre-recorded videos

Part I
Introduction to Misha

Cluster Misha

- Named after Misha Mahowald
  - Computational neuroscientist
  - Invented silicon retina
- A collaborative effort between WTI and YCRC
- Current configuration
  - 26 CPU nodes
  - Two large memory nodes
  - 16 GPU nodes, each with 4 GPUs
    - 4 H100 nodes + 6 A100 nodes + 6 A40 nodes, a mix of Center nodes and PI-purchased nodes
    - The Center will purchase more H100 and H200 nodes in the next few months.
    - PIs are welcome to purchase nodes with priority access and preemption privileges.

Storage on Misha

- Each user has a home, project, and scratch with limits on disk space and file count
  - Home limits cannot be increased
  - Project limits can be increased with justification
  - Scratch limits are usually not allowed to be increased
- Exceeding any of the limits may cause job failures
- Check your disk usage with utility script and OOD app
  - The shell script: getquota
  - OOD getquota app: utilities -> getquota

Clean up Home Space

- Home limits are fixed – no increase in disk quota and file count limit
- When it is full, many tasks will fail – a common reason for OOD failure
- Find out which files/directories take the most space in $HOME `du -h -d 2 $HOME`
- Common “disk hogs” in home
  - Conda directory: $HOME/.conda
  - User R lib directory: $HOME/R
  - OOD RStudio Server temporary working directory:
    - $HOME/ondemand/data/sys/dashboard/batch_connect/sys/ycrc_rstudio_server
  - OOD Job Composer job working directory:
    - $HOME/ondemand/data/sys/myjobs
- How to relocate a “disk hog”
  - Example: relocate conda directory

```
nohup mv $HOME/.conda $PROJECT/.conda &
ln -s $PROJECT/.conda $HOME/.conda
```

CPU, Host Memory, and GPU

- CPU count, host memory size, GPU count

Node type	CPU Count	Total Memory	GPU Count
H100	48	975	4
A100	32	1000	4
A40	32	975	4

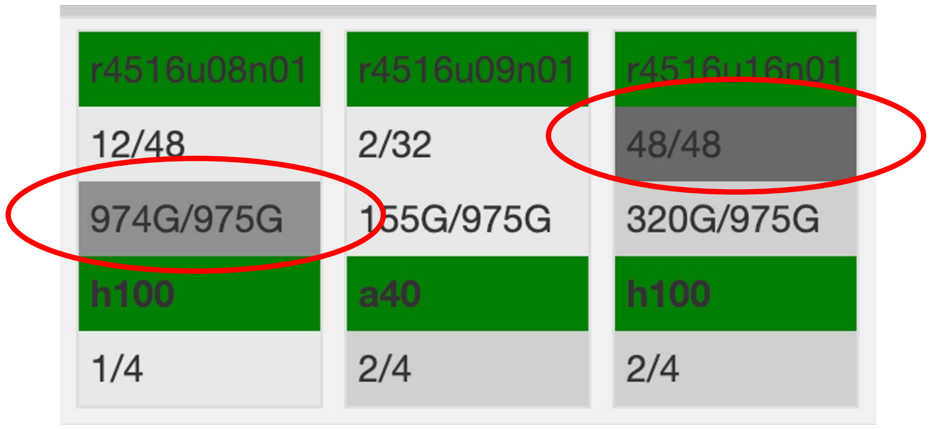
- For each GPU requested: CPU and memory $\leq$ total/gpu_cnt

Node type	CPU Count	Memory Size
H100	<= 12	<= 243
A100	<= 8	<= 250
A40	<= 8	<= 243

- These are recommendations, not hard limits.
  - Some applications may need more than the maximum recommended to run

Cluster Node Status OOD App

- Use the cluster node status app to check node usage
  - May reduce your job wait time by helping you choose available GPUs.
  - Discover inefficient use of nodes.
  - What’s wrong with the following node usage (thanks Sizhuang He for capturing and sending it to me!)



r4516u08n01	r4516u09n01	r4516u16n01
12/48	2/32	48/48
974G/975G	155G/975G	320G/975G
h100	a40	h100
1/4	2/4	2/4

Multi-Instance GPU (MIG)

- New capability supported by A30, A100, H100, H200, etc.
- Divide a physical GPU into multiple GPU instances, each with its own memory, cache, and streaming multiprocessors.
- For workload s with reduced cores and memory requirements.
- MIG doesn’t support multi-gpu jobs.
  - Do not run multi-gpu jobs in ‘gpu-devel.’
- ‘gpu_devel’ has four MIG-enabled A100s on one node.
  - Each GPU card supports 5 MIG instances:
    - Two instances with 20gb VRAM: a100.MIG.20gb
    - Three instances with 10gb VRAM: a100.MIG.10gb
  - 20 jobs can run simultaneously on a single node
    - In total, they cannot exceed 32 CPU cores and 1000G host memory
- https://yalewti-cnmi.github.io/misha/mig

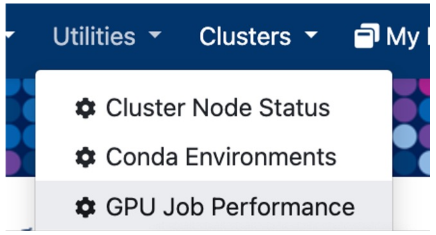
How to Choose Partitions and Nodes

- GPU Partitions
  - ‘gpu-devel’
    - developing and debugging code
    - testing code with a relatively small dataset
    - Cannot run PyTorch in VSCode Proxy and multi-gpu jobs
  - ‘gpu’
    - Production runs
    - Anything that cannot run in ‘gpu-devel’
      - For example: multi-gpu workload, running PyTorch in VSCode Proxy

- GPU types
  - h100, a100, a40
  - a100.MIG.20gb, a100.MIG.10gb
  - Performance, availability, and speedups

h100	a100	a40	a100.MIG.20gb
638 s	905 s	878 s	907 s

- GPU job performance monitoring tool on OOD
  - If GPU and VRAM utilization is low, consider using low-end GPU with less VRAM.
  - Optimize your code



QoS-based GPU Resource Allocation

- A new protocol to ensure the quality of service for groups that purchased GPUs in Misha,
    - Doesn't divide GPU nodes into small PI partitions based on the number of nodes they purchased.
    - A QoS tag for each contributor group defines the number and type of GPUs the group is guaranteed to access.
    - Non-priority jobs may be preempted to free up resources for contributor jobs.
    - A one-hour grace period
    - Preempted jobs can use the grace period to do checkpointing and cleanup.
    - More about checkpointing and restarting later
- https://yalewti-cnmi.github.io/misha/qos-slurm-protocol/

Conda Environments – Dos and Don’ts

- Dos
  - When building a Conda environment, submit an interactive batch job using ‘salloc.’

```
salloc --mem=20g [other options]
```

- Don’ts
  - Build a Conda environment on a login node
    - Login nodes have a strict memory limit – 1 GiB
    - Most builds will fail
  - Submit a job from a shell session when a Conda environment is activated.
    - You will experience strange error when running your code that needs to activate a Conda environment.

VSCode Proxy and Code Server

- Some users like to run the VSCode SSH extension to connect to the cluster from their local machine
  - Pros: users get the same environment as their local machine
  - Cons: it runs on a login node with limited memory and CPU time; no access to GPUs
- Use VSCode Proxy instead
- Code Server:
  - When to use:
    - If you don’t have a local VSCode
    - If you like to access everything in a browser
  - Disadvantage: it doesn’t have all the extensions available in VSCode
- https://yalewti-cnmi.github.io/misha/vscode-proxy/

Part II
Introduction to Experiment Tracking and Reproducibility with W&B

What is “Experiment Tracking”?

- Organize and manage experimental runs
- Track and log:
    - hyperparameters and configurations
    - performance metrics over time
    - store artifacts, code versions, etc.
    - visualization & collaboration tools

Why Do Experiment Tracking?

- Never lose your results: a way to organize data (avoid complex directory structure)
- Always track the exact configuration and experiment that produced a particular set of results.
- Share with collaborators.
- A convenient way to quickly visualize and inspect your results
- Improved reproducibility and transparency for your research

Demo: What Does a W&B Project Look Like

Basic Usage

- wandb.init: start an experimental run
- wandb.log: log a metric in the current experimental run
  - Incrementally log metrics (e.g. , throughout training)
  - Log images, videos, matplotlib plots, etc.
  - Log histograms (e.g., histogram of gradients)
- wandb.finish: end the current experimental run

Docs: https://docs.wandb.ai/ref/python/log/

Code Demo: How to Use W&B

W&B: Fetch Results after Logging

- Logs are hosted on W&B’s server.
- Can use W&B API (e.g., through Python package) to fetch this data.
  - For example, to create figures for paper.

https://docs.wandb.ai/ref/python/public-api/api/#methods

Part III
Misc PyTorch Optimization

Compilation and Automatic Mixed Precision

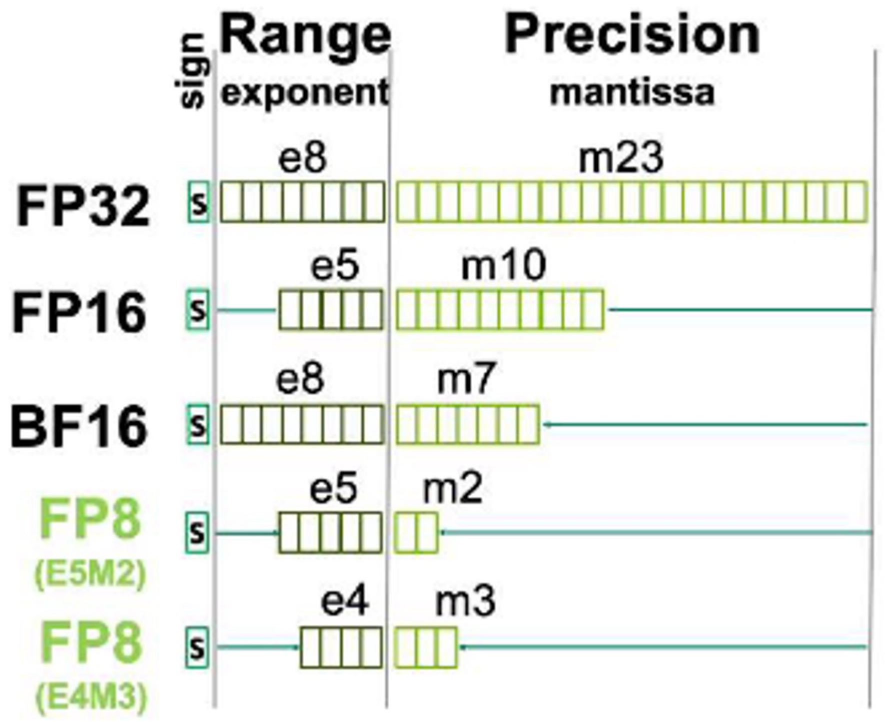
Introduction to Automatic Mixed Precision

Understanding Precision in GPU Computation

- "Precision": number of bits used to represent a floating-point number
  - e.g., 8-bits, 16-bits, 32-bits, etc.
- Higher precision => more accurate floating-point calculations.
- Computation time is proportional to the number of bits.
- Trade-off: more bits means higher-precision, but slower computation.

Role of Floating-Point Representation & bfloat6

- FP32
  - Standard single-precision format;
  - good balance of precision & range;
  - works well for a variety of computations
- FP16
  - Standard half-precision format
  - Faster, less memory usage compared to FP32
- BF16
  - Format designed for DL applications
  - Same dynamic range as FP32, but lower precision
  - Same number of bits as FP16, but more emphasis on dynamic range than precision



Source: https://resources.nvidia.com/en-us-tensor-core

Runtime at Different Levels of Precision

- Theoretically, directly proportional to # of bits
- E.g., from H100 whitepaper:

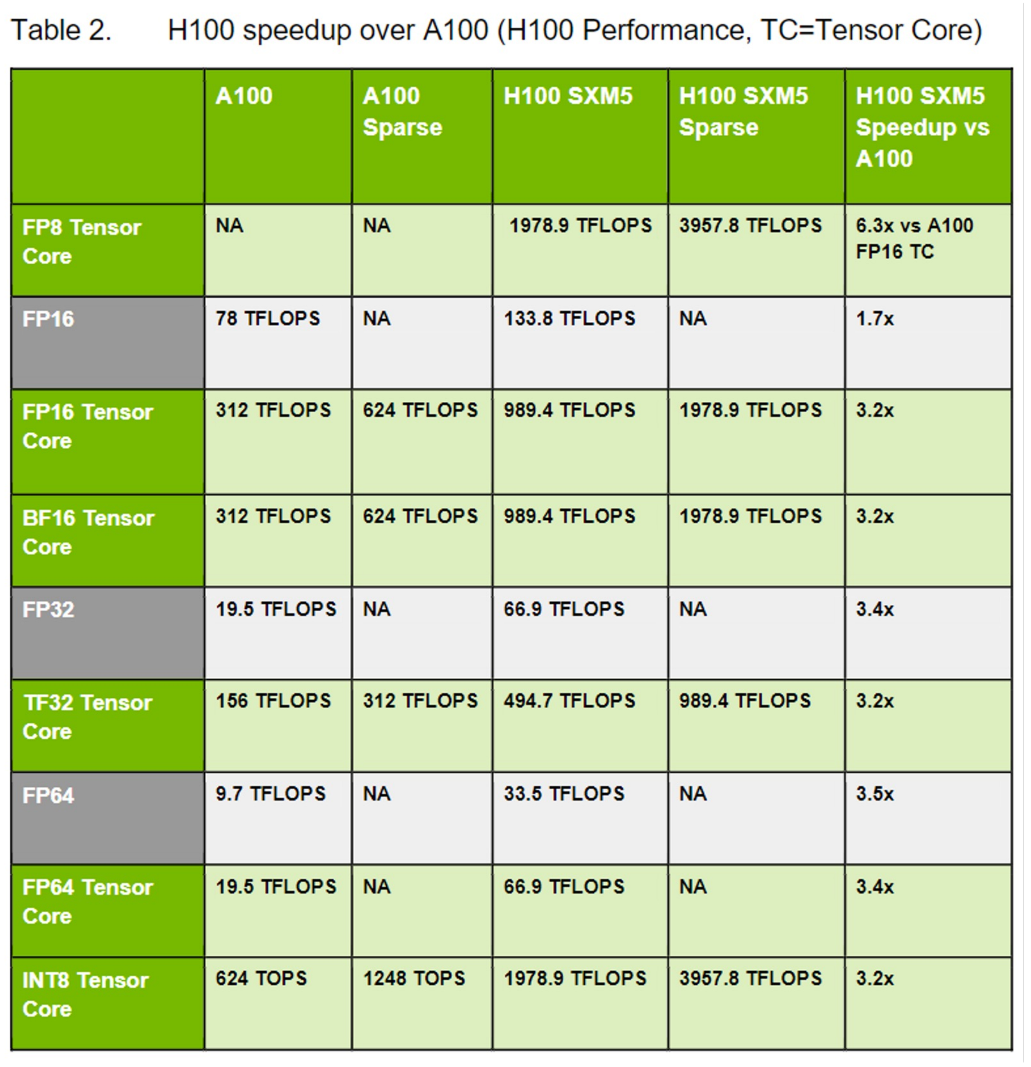


Table 2. H100 speedup over A100 (H100 Performance, TC=Tensor Core)

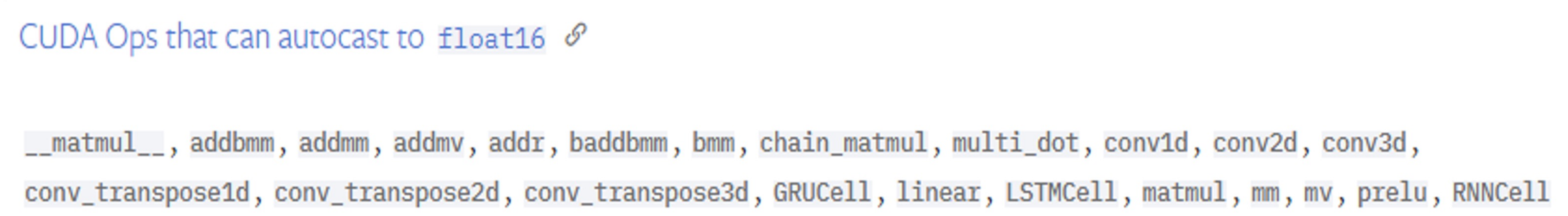
	A100	A100 Sparse	H100 SXM5	H100 SXM5 Sparse	H100 SXM5 Speedup vs A100
FP8 Tensor Core	NA	NA	1978.9 TFLOPS	3957.8 TFLOPS	6.3x vs A100 FP16 TC
FP16	78 TFLOPS	NA	133.8 TFLOPS	NA	1.7x
FP16 Tensor Core	312 TFLOPS	624 TFLOPS	989.4 TFLOPS	1978.9 TFLOPS	3.2x
BF16 Tensor Core	312 TFLOPS	624 TFLOPS	989.4 TFLOPS	1978.9 TFLOPS	3.2x
FP32	19.5 TFLOPS	NA	66.9 TFLOPS	NA	3.4x
TF32 Tensor Core	156 TFLOPS	312 TFLOPS	494.7 TFLOPS	989.4 TFLOPS	3.2x
FP64	9.7 TFLOPS	NA	33.5 TFLOPS	NA	3.5x
FP64 Tensor Core	19.5 TFLOPS	NA	66.9 TFLOPS	NA	3.4x
INT8 Tensor Core	624 TOPS	1248 TOPS	1978.9 TFLOPS	3957.8 TFLOPS	3.2x

Automatic Mixed Precision

- General idea of automatic mixed-precision: use higher precision (more bits) for operations that require it, and lower precision for more stable operations
- How it works:
  - PyTorch keeps a list of “stable” and ”unstable” primitive operations.
  - If AMP is turned on, it automatically casts stable operations to lower precision and keeps high precision for less stable operations.
  - Results in significantly faster runtime overall (2-3x faster), with little sacrifice to performance.

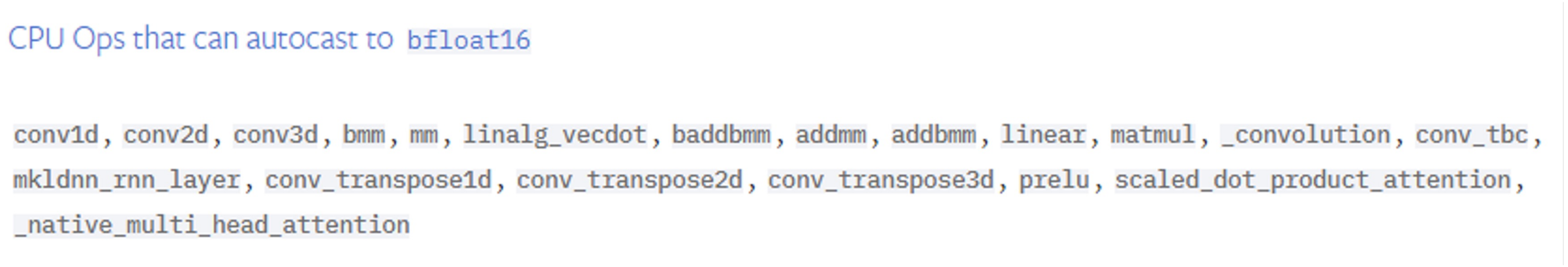
How to Use Automatic Mixed Precision in PyTorch

- Very simple: use ‘torch.autocase’ context manager
  - It will automatically cast to different levels of precision depending on CUDA operation
  - Note: can autocase to FP16, but this format is not designed for deep learning applications, and its dynamic range is relatively small. If using FP16, need to use “Gradient Scaling”. Modern GPUs, including all of Misha’s GPUs support BF16 which has large dynamic range and doesn’t require gradient scaling. gradient scaling.



CUDA Ops that can autocast to `float16`

`__matmul__`, `addbmm`, `addmm`, `addmv`, `addr`, `baddbmm`, `bmm`, `chain_matmul`, `multi_dot`, `conv1d`, `conv2d`, `conv3d`, `conv_transpose1d`, `conv_transpose2d`, `conv_transpose3d`, `GRUCell`, `linear`, `LSTMCell`, `matmul`, `mm`, `mv`, `prelu`, `RNNCell`



CPU Ops that can autocast to `bfloat16`

`conv1d`, `conv2d`, `conv3d`, `bmm`, `mm`, `linalg_vecdot`, `baddbmm`, `addmm`, `addbmm`, `linear`, `matmul`, `_convolution`, `conv_tbc`, `mkldnn_rnn_layer`, `conv_transpose1d`, `conv_transpose2d`, `conv_transpose3d`, `prelu`, `scaled_dot_product_attention`, `_native_multi_head_attention`

Code Demo: Automatic Mixed Precision in PyTorch

Introduction to torch.compile

Compilation: Eager Execution vs Compiled

- Eager Execution:
  - Default Default mode of operation in PyTorch
  - Operations executed line-by-line as they are called (similar to standard Python)
  - Advantages: intuitive, flexible, and easy to debug
  - Disadvantages: incurs overhead because each operation has to go through interpreter
- Compiled Code:
  - Allow PyTorch to “look ahead” at full model code, and compile it to CUDA
  - Operations are captured and transformed to a static representation
  - With this static graph representation, PyTorch can perform several optimizations

Optimizations and Expected Speedups

- Speedup mainly comes from reducing Python overhead and GPU read/write operations
- Speedup varies by model architecture, batch size, etc.
- Optimizations from static analysis of computation graph:
  - Operator fusion: combine multiple ops into single kernel, reducing memory I/O ops
  - Constant folding: pre-compute parts of model that have constant value
  - Common subexpression elimination: remove redundant computation and reuse prev results
  - Memory planning : allocate memory more efficiently, reduce I/O or alloc/dealloc overhead

Usage

- Again, very simple:

```
model = torch.compile(model)
```

- Overhead on first training loop, but faster after that.
- Usually worth it

Demo: torch.compile

References and More Information

- https://pytorch.org/docs/stable/amp.html
- https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html
- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
- https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html

Part IV
Checkpointing and Restarting

SLURM Preemption

- On Misha, a regular (non-priority) job can utilize any PI-purchased GPUs, but it may be preempted.
- Misha SLURM is configured with a one-hour grace period for preempted jobs to stop running properly.
- When a job is selected for preemption, SLURM sends SIGCONT and SIGTERM to notify the job of its imminent termination. When the grace period is over, SLURM sends SIGCONT, SIGTERM, and SIGKILL to terminate the job.
- The first SIGTERM is sent to job steps only. To catch it in the user code and trigger checkpointing and other tasks, the user code must be launched with ‘srun.’

How to Catch Signal in Python Code

simple.py

```
#!/usr/bin/env python3.11
import signal
import sys
import time
SIGTERM = 15

def signal_handler(sig, frame):
    print('A signal arrived.’)
    time.sleep(10)

signal.signal(SIGTERM, signal_handler)
time.sleep(600)
print('Done')
```

simple.job

```
#!/bin/bash
#SBATCH –p gpu
#SBATCH ––gres=gpu:a100:1
#SBATCH –n 1 ––ntasks–per–node=1
date
srun ./simple.py
echo "finished"
```

CPL Python Library

- Why implement a library
  - Provide a simple way of checkpointing and automatic restarting, with add-on features including email notification and event logs.
- How to install `pip install cpl@git+https://github.com/YaleWTI-CNMI/checkpointing`
- https://github.com/YaleWTI-CNMI/checkpointing

Simplest Example

```
from cpl import cpl

mycpl = cpl.CPL()
checkpt_file = Path(mycpl.get_checkpoint_fn())

(restart from checkpt_file)

(within the main loop)
    if mycpl.check():
        (checkpoint and clean up)
        break
```

See examples/example1

Example 2 – Configure CPL

- You can configure the library using ‘cpl.yml.’
  - Set up email notifications. The default is no email notifications.
    - A job is selected for preemption.
    - Checkpointing and cleanup is done.
  - Log major events when using the library. The default is no log.
  - Provide a checkpoint file name. The default name is “checkpoint.ckp”.

cpl.yml

```
email_server: mail.yale.edu
email_address: ping.luo@yale.edu
email_types:
  signal_caught: True
  checkpoint_handler_done: True

logfile: cpl.log
loglevel: INFO

checkpoint_fn: model_checkpoint.ckp
```

See examples/example2

Example 3 – Delayed Checkpointing

- When a signal has arrived, postpone the checkpointing and continue running the code for a period specified in ‘cpl.yml.’ This is useful for utilizing the one-hour grace period for some useful work.

cpl.yml

```
email_server: mail.yale.edu
email_address: ping.luo@yale.edu
email_types:
  signal_caught: True
  checkpoint_handler_done: True
logfile: cpl.log
loglevel: INFO
checkpoint_fn: model_checkpoint.ckp

delay: 5 # in minutes
```

See examples/example3

Example 4 – Function for Checkpointing and Cleanup

- For modular programming, implement checkpointing and cleanup as a function.

```
def my_checkpoint(kwargs):
    checkpt_file = kwargs['filepath’]
    (checkpoint and clean up)

from cpl import cpl
mycpl = cpl.CPL()
checkpt_file = Path(mycpl.get_checkpoint_fn())
(restart from checkpt_file)
(within the main loop)
    if mycpl.check(checkpoint_handler=my_checkpoint,filepath=checkpt_file):
        break
```

See examples/example4

Example 5 – Automatic Checkpointing and Restarting

- Must set ‘back=False” in CPL.check(): `mycpl.check(back=False)`
- Must have ‘--requeue’ in your SLURM job script.
- How does it work?
  - When ‘CPL.check(back=False),’ the call to the method will not return to its caller after checkpointing and cleanup. The job stays in CPL.check() until the grace period is over and it is preempted.
  - Since the job has ‘--requeue’, SLURM puts it back in the queue after it is preempted.
  - When the resource is available again, the job is scheduled to run. The program reads the checkpoint file and restarts from where it stops

See examples/example5

Automatic Checkpointing and Restarting Best Practices

- Always do delayed-checkpointing so your job will not waste the grace period.
- Use $SLURM_RESTART_COUNT to rename the job output file so it is not overwritten each time the job is restarted.

```
[ "x${SLURM_RESTART_COUNT}" == "x" ] && SLURM_RESTART_COUNT=0
srun python ex5.py > ex5.out.${SLURM_JOB_ID}.${SLURM_RESTART_COUNT}
```

See examples/example5/ex5.job

Call for Collaboration

- Do you want to use any vacant nodes whenever possible?
- Are you interested in automatically checkpointing and restarting your (non-priority) jobs?
- Please contact ping.luo@yale.edu

Acknowledgment

- We thank the **Yale Center for Research Computing** for their support in building and maintaining Misha.