

NAIRR Pilot National Artificial Intelligence
Research Resource Pilot

NAIRR Pilot

National Artificial Intelligence
Research Resource Pilot

Aptos or Arial 18pt White
Name of Presentation
Title
Name
Date/ Session
AI Workshop Denver, CO April 2-3, 2025



Session Summary: Submitting an AI Job (90 minutes)

This session provides a practical introduction to running AI jobs on computational resources, with a focus on tools, techniques, and optimization strategies. Key topics include:

1. AI Frameworks Overview:

- Introductions to **TensorFlow** and **PyTorch**, including their features and use cases.
- Exploration of additional AI tools and libraries for specialized applications.

2. Hardware Considerations:

- Differences between **GPUs** and **CPUs** for AI workloads and how to select the right resource.

3. Reusable Resources:

- Example containers preconfigured for TensorFlow, PyTorch, and other frameworks to simplify job setup.

4. Optimizing Resource Usage

- Maximize computational efficiency and reduce runtime, including parallelization, batch computing, and effective memory usage.

Participants will leave with a clear understanding of how to set up and submit AI jobs while optimizing performance on their chosen computational platform.



























AI Frameworks Overview



- Introductions to **TensorFlow** and **PyTorch**,
 - including their features and use cases.
- Exploration of additional AI tools and libraries for specialized applications.

Top AI & ML Frameworks (3/23/25 Google search)

 TensorFlow	▼	 PyTorch	▼	 Scikit-learn	▼
 Keras	▼	 Caffe	▼	 Theano	▼
 Apache MXNet	▼	 Amazon SageMaker	▼	 Apache Spark	▼
 H2O	▼	 Hugging Face	▼	 Torch	▼
 XGBoost	▼	 Apache Mahout	▼	 Shogun	▼
 Accord.NET	▼	 Apache SINGA	▼	 Azure ML Studio	▼
 Deeplearning4j	▼	 JAX	▼	 OpenAI	▼
 Chainer	▼	 Microsoft CNTK	▼	 Microsoft Cognitive Toolkit	▼

Checkout: <https://www.valuecoders.com/blog/technology-and-apps/winning-with-ai-and-machine-learning-frameworks/>



Popular AI & ML Frameworks

- Tensorflow
 - Runs on CPU, GPU
 - Google Tensorflow Processing Units (TPUs). GPUs are commonly used for deep learning model training and inference.
 - Keras: high-level API for TensorFlow; provides a user-friendly interface for building and training neural networks
 - Some examples:
 - <https://medium.com/@golnaz.hosseini/step-by-step-tutorial-image-classification-with-keras-7dc423f79a6b>
 - <https://medium.com/@golnaz.hosseini/beginner-tutorial-image-classification-using-pytorch-63f30dcc071c>



PyTorch vs Tensorflow

- PyTorch popular in research community
- Design emphasizes a more “Pythonic” approach:
 - concise, easier to debug, more flexible
- PyTorch is known for its dynamic computational graph, which allows for more flexibility and easier debugging compared to TensorFlow's static graph.
- PyTorch gaining traction in industry
- Easier to debug
- TensorFlow holds larger market share
- known for robustness in production environments
- TensorFlow 2.0 and later versions have simplified the API,

TensorFlow Overview

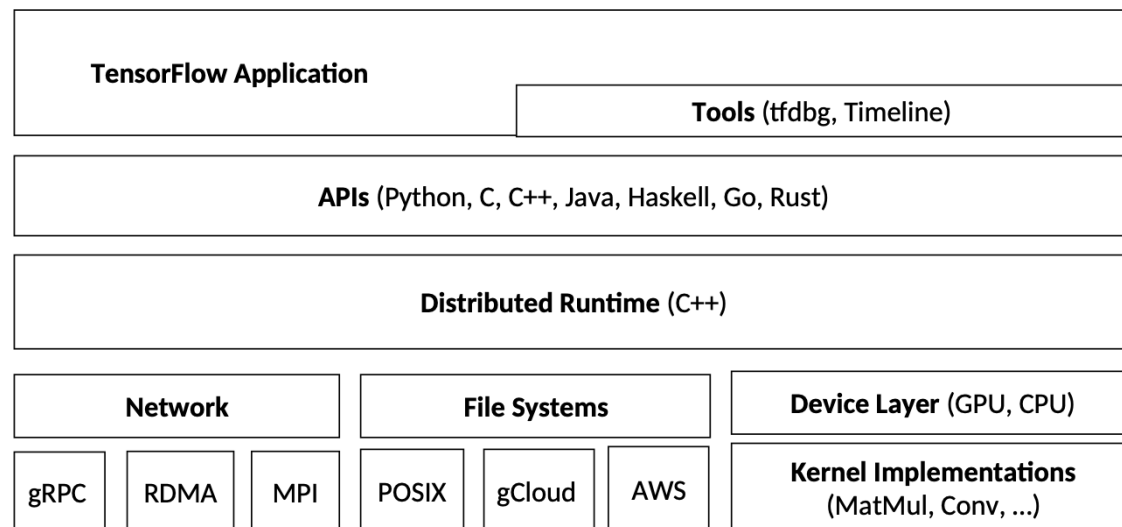
Tensorflow

<https://medium.com/@golnaz.hosseini/beginner-tutorial-image-classification-using-pytorch-63f30dcc071c>

pytorch-63f30dcc071c

Keras: high-level API for TensorFlow;
provides a user-friendly interface for building
and training neural networks

<https://medium.com/@golnaz.hosseini/step-by-step-tutorial-image-classification-with-keras-7dc423f79a6b>

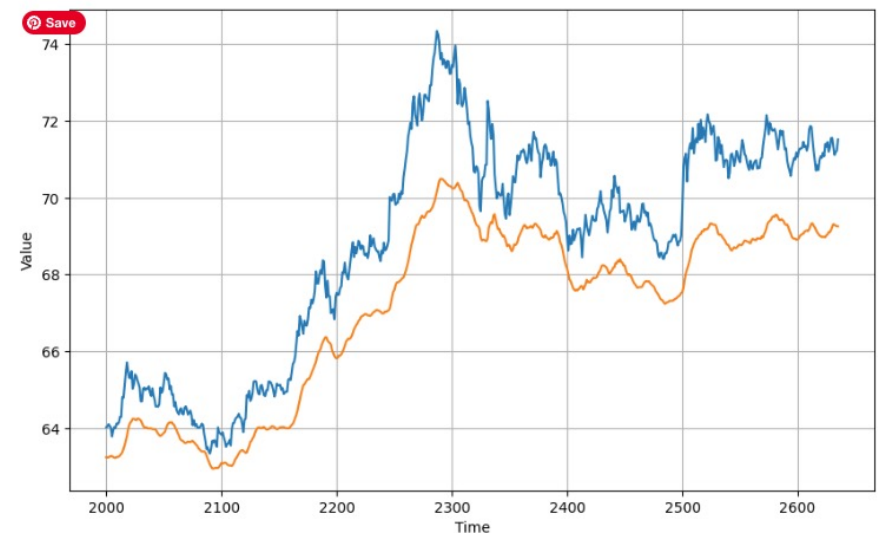


TensorFlow software stack



TensorFlow: "Simple" Time Series

- Expanse Notebooks:
Tensorflow_Simple_Training
- <https://medium.com/analytics-vidhya/sequence-model-time-series-prediction-using-tensorflow-2-0-665257beb25f>
- <https://www.geeksforgeeks.org/time-series-forecasting-using-recurrent-neural-networks-rnn-in-tensorflow/>





PyTorch Overview

- Hands-on “Learn the Basics” PyTorch:
 - <https://pytorch.org/tutorials/beginner/basics/intro.html>



AI/ML Examples

- “Hello World”
 - <https://developers.google.com/codelabs/tensorflow-1-helloworld#1>
 - <https://parthdevai.medium.com/hello-world-program-with-ai-artificial-intelligence-ae8acd86c71c>
- Image classifications



Hello World of AI/ML/DL: Tensorflow

- Tensorflow: Hello World of Deep Learning with Neural Networks



Datasets

- <https://www.geeksforgeeks.org/mnist-dataset/>



Hands-on: AI Frameworks

- List slides to use for hands-on

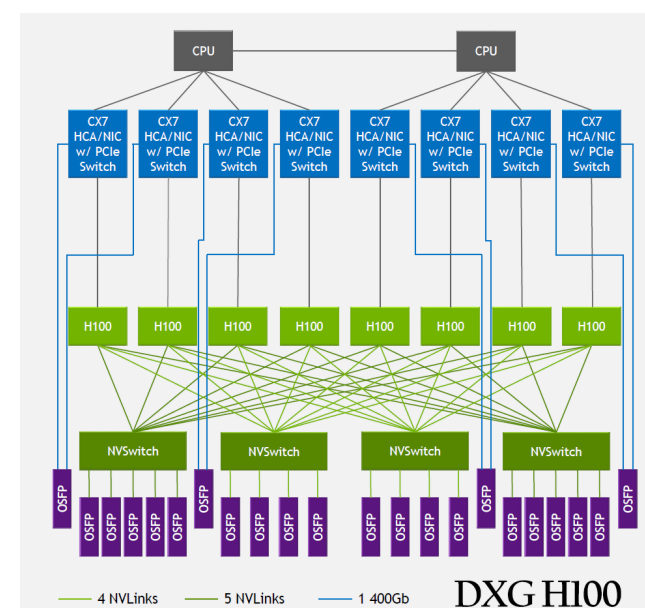
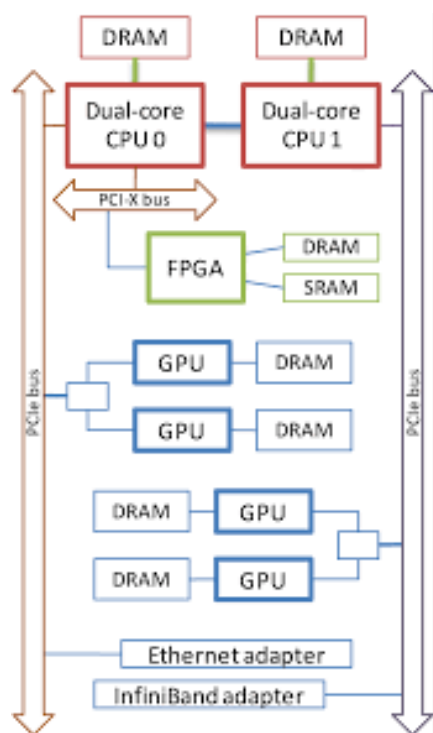


2. Hardware Considerations

Differences Between GPUs and CPUs for AI Workloads

CPUs (Central Processing Units) and **GPUs** (Graphics Processing Units) are optimized for different types of tasks. When selecting the right resource for AI workloads, understanding the key differences is crucial.

- 1. Architecture and Parallelism
- 2. Performance for AI Workloads
- 3. Memory and Data Handling
- 4. Power Efficiency



<https://www.naddod.com/blog/gpu-cluster-network-in-large-language-model>

A Heterogeneous Multi-Accelerator Cluster.

https://www.researchgate.net/publication/265873345_QP_A_Heterogeneous_Multi-Accelerator_Cluster (2009)



Architecture and Parallelism

CPU (Central Processing Unit):

- Designed for General-purpose Computing: CPUs are built to handle a wide variety of tasks and are optimized for sequential processing. Ideal for tasks that require strong single-threaded performance.
- Few Cores (Typically 4-16 cores): CPUs have fewer, but more powerful cores, optimized for tasks that need high single-thread performance (e.g., complex control logic, sequential algorithms).
- Limited Parallelism: better suited for serial for small-scale parallel tasks (e.g., handling OS, logic operations, I/O operations, and light computational work).

GPU (Graphics Processing Unit):

- Designed for Highly Parallel Workloads: optimized for high-level parallel computation, useful for AI/ML workloads, where multiple operations can be performed simultaneously.
- Thousands of Cores: smaller, simpler optimized for parallel execution. This allows them to handle a massive number of tasks simultaneously, making them perfect for matrix multiplications, convolutions, and other parallelizable operations common in deep learning.
- Optimized for throughput, not latency: GPUs excel in throughput (performing many operations in parallel), rather than latency (minimizing time taken for individual operation).



Performance for AI Workloads

CPUs for AI:

- **Smaller Scale Workloads:** CPUs perform better for workloads where tasks are not highly parallelizable (e.g., small models, inference on small data, or tasks that require fast single-threaded processing).
- **Flexibility:** CPUs can handle a variety of tasks like handling system I/O, managing control flows, and running complex logic that isn't easily parallelizable.
- **Latency-Sensitive Tasks:** For tasks that require low latency and fast context switching (e.g., running inference in real-time for a limited number of requests), CPUs are often preferred.

GPUs for AI:

- **Training Large Models:** GPUs are particularly suited for training deep learning models, which require the simultaneous computation of large matrices (e.g., matrix multiplications in deep neural networks).
- **Massive Parallelism:** Tasks like convolution in CNNs, backpropagation in neural networks, and other matrix-heavy operations benefit from the parallel architecture of GPUs, which can handle thousands of operations at once.
- **Faster Model Training and Inference:** GPUs can train AI models significantly faster (up to 10x or more) compared to CPUs due to their parallel structure, especially for large datasets and deep models.



Memory and Data Handling

- CPU Memory (RAM):
 - CPUs have a relatively small, fast cache (L1, L2, L3) and rely on system RAM for large data access. The memory is more flexible, supporting random access across diverse memory patterns.
 - However, for deep learning models with massive datasets, the memory bandwidth of a CPU can become a bottleneck.
- GPU Memory (VRAM):
 - GPUs typically have high-bandwidth memory (VRAM), optimized for quick data transfers to and from the GPU cores.
 - The VRAM is crucial when training large deep learning models. GPUs usually come with 8GB to 40GB+ of VRAM (depending on the card), which is essential for processing large batches of data and training deep neural networks.
 - The GPU's memory is optimized for handling large volumes of matrix data, which is common in AI workloads.



Power Efficiency

- CPU:
 - CPUs are generally more power-efficient for single-threaded tasks or light workloads. They use less power for lower levels of parallelism and single-core performance.
 - However, when scaling up to AI workloads (which are highly parallel), CPUs become less efficient in terms of power usage relative to performance.
- GPU:
 - GPUs tend to consume significantly more power compared to CPUs, especially when fully utilized during deep learning model training or large-scale data processing. This is because of the vast number of cores and parallel tasks being performed.
 - However, they deliver better performance for parallel tasks, so even though power consumption is higher, the performance-to-power ratio is often much better than CPUs for deep learning tasks.



When to use a CPU vs. a GPU in your ML workflow

Stage	CPU	GPU	Low latency database
Data preprocessing	Good for general-purpose tasks like data cleaning and feature engineering for small to medium datasets.	Accelerates parallel operations like image transformations or matrix computations, especially on large datasets.	Fast data access for preprocessing. Essential for real-time data streaming and updates during preprocessing.
Exploratory data analysis (EDA)	Efficient for statistical analysis, summary generation, and visualizations. Excels at ad-hoc operations.	Accelerates parallel data-intensive tasks like visualization and complex statistics on large datasets.	Quick access to and efficient filtering of large real-time datasets for analysis.
Model selection and training	Good for training traditional ML models like decision trees and logistic regression, good enough for smaller datasets.	Parallelizes and accelerates training of computationally-intensive algorithms on large datasets, highly effective for batch processing and matrix computations.	Stores and updates training data in real-time, enables streaming data models and real-time model training. Used to store intermediary results and model checkpoints during distributed training.
Model evaluation and refinement	Handles small-to-moderate-scale model evaluation tasks such as cross-validation, metric calculations, and light model refinement.	Excels at evaluating complex models on large datasets quickly, especially large-scale batch testing.	Stores model performance metrics for comparison and retrieval, enabling quick feedback loops during refinement.
Model deployment	Suitable for lightweight and smaller models, moderate inference demands where batch processing isn't required. Good for IoT/edge small-scale or infrequent inference.	Uses parallel processing to accelerate computationally heavy high-throughput applications over large data sets such as image processing, ensemble models, and NLP.	Critical for real-time applications where models require fast data retrieval during inference such as recommendation engines, fraud detection, streaming data analysis.
Ongoing monitoring and maintenance	Handles routine monitoring and maintenance tasks such as logging, metrics, and running scripts.	Useful for real-time retraining and updating of models, best for real-time model adaptation.	Storage and retrieval of monitoring data in real-time at scale, fast logging of metrics allows for real-time model monitoring.

<https://aerospike.com/blog/cpu-vs-gpu/>



How to Select the Right Resource for Your AI Workload

- Choosing the right resource—whether a CPU or a GPU—depends on the nature of your AI workload. Here's how to decide which one is best suited for your task:
 - For Training Deep Learning Models:
 - For Inference (Model Deployment)
 - For Small AI Models and Lightweight Tasks
 - For Edge and Mobile AI Applications
 - For Speeding Up AI Research (Prototyping)
 - For Multi-Task and High-Throughput Applications:



For Training Deep Learning Models:

- GPU: GPUs are the best choice for training large models (e.g., neural networks like CNNs, RNNs, Transformers). They can process large datasets much faster because of their parallelism, significantly reducing training times.
- Recommended GPUs:
 - NVIDIA Tesla A100 or NVIDIA V100 for high-end, research-grade workloads.
 - NVIDIA RTX 3080/3090 or NVIDIA RTX A6000 for more accessible yet powerful options.
 - Google TPUs (Tensor Processing Units) are another option if you're working within Google Cloud.



For Inference (Model Deployment):

- CPU: If you are deploying smaller models, or the inference task is not computationally intense (e.g., lightweight models for edge devices or applications with strict latency requirements), a CPU might suffice. CPUs are also more suitable when running AI tasks in environments with limited resources, such as in cloud-based systems with lower hardware costs.
- GPU: For inference of larger models or when you need faster throughput (e.g., real-time AI applications), GPUs are a better fit. This is especially true for models that involve high computational workloads like large NLP models (e.g., BERT, GPT).



For Small AI Models and Lightweight Tasks

- CPU: If your models are smaller, such as decision trees, support vector machines (SVM), or shallow neural networks, and the dataset is not particularly large, CPUs can perform well and are more cost-effective.
- GPU: Not necessary for these types of tasks unless you're running a large-scale deployment where many predictions need to be made in parallel.



4. For Edge and Mobile AI Applications

- CPU: In edge devices or mobile environments, CPUs are typically preferred due to power and memory limitations. Many mobile devices also come with specialized AI processors like Apple's Neural Engine or Qualcomm's AI Engine, designed to optimize deep learning models with low power consumption.
- GPU: Some mobile devices have integrated GPUs, like Apple's M1 chip, which can be used for AI inference, but the resource limitations of mobile GPUs generally mean that only optimized models (quantized, pruned) are viable.



5. For Speeding Up AI Research (Prototyping):

- GPU: If you are running experiments on a variety of deep learning models and need to iterate quickly, GPUs are the better choice due to their ability to handle large datasets and complex models. They can drastically reduce the time it takes to iterate on model architectures and hyperparameters.



6. For Multi-Task and High-Throughput Applications

- GPU: When you need to process large batches of data or run multiple models concurrently, GPUs are designed for throughput and are much better at handling these parallel tasks compared to CPUs.



Hands-on: Hardware Considerations

- Launch PyTorch MNIST app
- Run it on GPU and CPU
- Vary a few variables
- Measure runtime
- Compare performance



3. Reusable Resources



From the perspective of a scientist using AI/ML code, "reusable resources" refer to tools, environments, and pre-configured systems that allow easy and efficient development, experimentation, and deployment of machine learning models. These resources help save time, improve reproducibility, and streamline workflows.

Some

- Cloud-based Environments
- Development Environments
- Pre-built Frameworks & Libraries
- Framework-Specific Solutions
- Machine Learning Pipelines



Pre-configured Containers (Docker, Singularity, etc.)

Definition: These are pre-built environments (usually Docker or Singularity containers) that contain all the dependencies and libraries needed for machine learning tasks. Containers ensure that the code runs consistently across different systems and help avoid compatibility issues.

Examples:

- Docker containers with specific versions of TensorFlow, PyTorch, or scikit-learn.
- Singularity containers used in high-performance computing (HPC) environments, allowing researchers to package and run ML models with ease.

Benefits:

- Saves time on environment setup.
- Promotes reproducibility by ensuring the same setup is used across experiments.
- Easy to share with collaborators.



Pre-configured Cloud-based Environments

Definition: Cloud-based platforms provide scalable, on-demand computing resources for machine learning. These environments often come with pre-configured machine learning tools and the ability to quickly scale resources based on the computational needs of a model.

Examples:

- Google Colab: A free cloud-based environment that provides access to GPUs and pre-installed ML libraries.
- Amazon SageMaker: A fully managed service to build, train, and deploy ML models at scale.
- Microsoft Azure ML: A platform that offers end-to-end tools for building, training, and deploying ML models in the cloud.

Benefits:

- Access to high-powered resources (e.g., GPUs, TPUs) without the need for physical infrastructure.
- Scalable, depending on the needs of your model.
- Collaborative features for team-based development



Development Environments

Definition: Development environments such as integrated development environments (IDEs), notebooks, or interactive shells make it easier for scientists to write, test, and debug machine learning code efficiently.

Examples:

- *Jupyter Notebooks*: A powerful tool for writing and sharing code, visualizations, and documentation. It's particularly popular in data science and machine learning for experimentation.
- *VSCode*: A widely used IDE with extensions for Python and machine learning frameworks, supporting debugging, code completion, and version control.
- *MATLAB*: IDE platform for design, development, and testing of AI/ML data science and statistical models.

Benefits:

- Efficient code development and experimentation.
- Interactive execution of code, useful for quick testing and debugging.
- Support for visualizations and plotting for better insights.



Pre-built Frameworks and Libraries

Definition: These are libraries and frameworks designed to simplify machine learning tasks. Pre-built tools help to avoid reinventing the wheel by offering common functionalities such as data preprocessing, model training, and evaluation.

Examples:

- TensorFlow, PyTorch, Keras: These libraries offer high-level APIs and pre-built functions for building neural networks and machine learning models.
- Scikit-learn: A popular library for classical machine learning algorithms.
- XGBoost, LightGBM: Libraries for gradient boosting models that are widely used in structured data problems.

Benefits:

- Saves development time by providing well-tested implementations of algorithms.
- Encourages the use of state-of-the-art models and methods.
- Reduces errors in implementation by relying on standardized libraries.



Framework-Specific Solutions

Definition: These are tools and resources specifically built around a particular framework or technology to improve development speed, simplify deployment, or automate tasks. They're tailored to fit the specific requirements of certain machine learning frameworks.

Examples:

- TensorFlow Hub: A library for reusable machine learning modules and pre-trained models in TensorFlow.
- PyTorch Lightning: A lightweight wrapper for PyTorch that abstracts away boilerplate code, simplifying model training and experimentation.
- FastAI: A library built on top of PyTorch that provides high-level abstractions for training models.

Benefits:

- Highly specialized for a specific framework, improving ease of use.
- Often provides pre-built models that save time and effort in training.
- Standardizes workflows for a specific framework, making them more efficient.



Machine Learning Pipelines

Definition: These are predefined sequences of steps that automate data processing, model training, and evaluation. ML pipelines help streamline and standardize workflows, and they are often used to improve the reproducibility and scalability of experiments.

Examples:

- Apache Airflow: A platform to programmatically author, schedule, and monitor workflows, often used for orchestrating machine learning pipelines.
- Kubeflow: A Kubernetes-native solution for deploying, monitoring, and managing ML workflows and models in production.
- MLflow: An open-source platform for managing the end-to-end machine learning lifecycle, including experimentation, reproducibility, and deployment.

Benefits:

- Automates repetitive tasks like model training, testing, and deployment.
- Ensures that processes are repeatable and auditable.
- Facilitates collaboration and consistency between different team members or environments.



Summary of "Reusable Resources" for AI/ML Scientists:

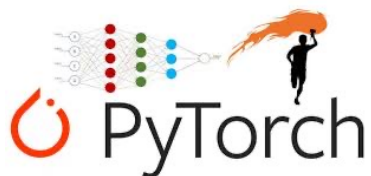
These reusable resources provide pre-configured, scalable, and interoperable environments, enabling AI/ML scientists to streamline their workflows and focus on experimentation and model development instead of infrastructure setup.

CATEGORY	EXAMPLES
Pre-configured Containers	Docker containers (e.g., TensorFlow, PyTorch), NVIDIA NGC containers, Singularity containers
Cloud-based Environments	Google Colab, Kaggle Kernels, Amazon SageMaker, Microsoft Azure ML
Development Environments	Jupyter Notebooks, Anaconda Environments
Pre-built Frameworks & Libraries	TensorFlow Hub, PyTorch Hub, Hugging Face Transformers, OpenCV, scikit-learn
Framework-Specific Solutions	TensorFlow Lite, ONNX
Machine Learning Pipelines	MLflow, Kubeflow



Working With Containers

- We'll focus on common set
- Containers
 - Docker
 - Singularity
- ML Frameworks
 - TensorFlow
 - PyTorch
- Focus on PyTorch MNIST application
 - <https://github.com/pytorch/examples/tree/main/mnist>





Running AI Jobs Using a Singularity container

- Locate the container you want to use:
 - create one, pull from an online source; use ones on your system
 - Expanse: /cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif
 - Expanse: /cm/shared/apps/containers/singularity/tensorflow-latest.sif
- Install/access Singularity: you need to run from this to have correct software ENV

```
[mthomas@login01 mnist-pyth-version]$ module load singularitypro/4.1.2
[mthomas@login01 mnist-pyth-version]$ which singularity
/cm/local/apps/singularitypro/4.1/bin/singularity
```
- Choose method to run application
 - Interactive: CLI (Command line) on login node; CLI on Expanse interactive node
 - Batch Script
 - Notebooks: using galyeo on Expanse or the OOD portal
- Launch Singularity shell:

```
/cm/local/apps/singularitypro/4.1/bin/singularity shell /cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif
```


Interactive: Running on login node: it's possible

1. Launch container so you have right libraries
2. Run the MNIST pytorch application: get Pthread type of error.
3. Need to control number of OPENBLAS threads or program memory is too large for node.
4. Change arguments to MNIST app s so it runs

```
1 thomas@login01] mnist-pyth-version]$
/cm/local/apps/singularitypro/4.1/bin/singularity shell
/cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif
Singularity> python3 main.py

2 OpenBLAS blas_thread_init: pthread_create failed for thread 57 of
64: Resource temporarily unavailable
OpenBLAS blas_thread_init: RLIMIT_NPROC 2000 current, 2000 max

Singularity> export OPENBLAS_NUM_THREADS=16 3
Singularity> python3 main.py --batch-size 4 --test-batch-size 4 --
epochs 4 --no-cuda
ARGS: Namespace(batch_size=4, test_batch_size=4, epochs=4,
lr=1.0, gamma=0.7, no_cuda=True, no_mps=False, dry_run=False, 4
seed=1, log_interval=10, save_model=False)
Train Epoch: 1 [0/60000 (0%)] Loss: 2.355359
```

```
1 [||||] 7.9% 17 [||||] 7.9% 33 [||||] 11.0% 49 [||||]
2 [||||] 11.7% 18 [||||] 7.6% 34 [||||] 8.3% 50 [||||]
3 [||||] 7.9% 19 [||||] 24.5% 35 [||||] 7.9% 51 [||||]
4 [||||] 8.2% 20 [||||] 8.0% 36 [||||] 8.3% 52 [||||]
5 [||||] 8.3% 21 [||||] 6.9% 37 [||||] 7.9% 53 [||||]
6 [||||] 9.0% 22 [||||] 7.9% 38 [||||] 21.8% 54 [||||]
7 [||||] 13.7% 23 [||||] 7.2% 39 [||||] 11.4% 55 [||||]
8 [||||] 7.6% 24 [||||] 6.9% 40 [||||] 9.3% 56 [||||]
9 [||||] 8.9% 25 [||||] 5.9% 41 [||||] 7.6% 57 [||||]
10 [||||] 8.2% 26 [||||] 7.6% 42 [||||] 7.6% 58 [||||]
11 [||||] 8.0% 27 [||||] 6.9% 43 [||||] 8.6% 59 [||||]
12 [||||] 7.6% 28 [||||] 7.9% 44 [||||] 7.2% 60 [||||]
13 [||||] 9.9% 29 [||||] 7.6% 45 [||||] 11.0% 61 [||||]
14 [||||] 8.7% 30 [||||] 7.2% 46 [||||] 12.4% 62 [||||]
15 [||||] 8.0% 31 [||||] 7.6% 47 [||||] 7.9% 63 [||||]
16 [||||] 8.9% 32 [||||] 7.2% 48 [||||] 8.6% 64 [||||]
Mem [|||||] 22.5G/125G Tasks: 1922, 940 thr, 1054 kthr; 2 running
Swp [|||||] 1.42G/12.0G Load average: 27.52 11.44 8.22
Uptime: 8 days, 09:42:41

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
323286 mgujral 20 0 95368 15148 8176 S 1.0 0.0 0:00.64 /usr/lib/systemd/systemd --user
324547 mthomas 20 0 7801M 546M 212M R 1.4 0.4 0:01.62 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324548 mthomas 20 0 7801M 546M 212M R 1.4 0.4 0:01.61 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324550 mthomas 20 0 7801M 546M 212M R 1.7 0.4 0:01.44 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324551 mthomas 20 0 7801M 546M 212M R 0.7 0.4 0:01.59 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324552 mthomas 20 0 7801M 546M 212M R 0.7 0.4 0:01.39 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324553 mthomas 20 0 7801M 546M 212M R 1.0 0.4 0:01.40 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324554 mthomas 20 0 7801M 546M 212M R 1.0 0.4 0:01.49 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324555 mthomas 20 0 7801M 546M 212M R 1.0 0.4 0:01.34 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
324558 mthomas 20 0 7801M 546M 212M R 0.7 0.4 0:01.38 python3 main.py --batch-size 4 --test-batch-size 4 --epochs 4 --no-cuda
326154 zhanglab 20 0 95372 15164 8184 S 0.7 0.0 0:00.57 /usr/lib/systemd/systemd --user
327419 root 20 0 135M 3384 2024 S 0.0 0.0 0:00.14 sshd: pwolberg [pam]
441063 esebastian 20 0 95720 14440 7216 S 1.0 0.0 0:59.76 /usr/lib/systemd/systemd --user
452668 gkulo 20 0 96000 14928 7204 S 1.0 0.0 5:12.68 /usr/lib/systemd/systemd --user
484530 adhayal 20 0 95880 13932 7200 S 1.0 0.0 6:34.62 /usr/lib/systemd/systemd --user
496977 smedapati 20 0 95716 14696 7216 S 1.4 0.0 1:57.94 /usr/lib/systemd/systemd --user
546649 keyajoshi 20 0 95712 14656 7240 S 0.3 0.0 0:54.11 /usr/lib/systemd/systemd --user
```



Interactive: Running on interactive node (Expanse)

1. Request interactive node
2. Launch the PyTorch Singularity container.
3. Run the MNIST application from the command line.
4. You can see results as they are computed.

```
[mthomas@login01] mnist-pyth-version]$ srun --partition=gpu-debug  
--pty --account=use300 --ntasks-per-node=10 --nodes=1 --mem=96G --  
gpus=1 -t 00:30:00 --wait=0 --export=ALL /bin/bash  
srun: job 37743618 queued and waiting for resources  
srun: job 37743618 has been allocated resources  
[mthomas@exp-7-59] mnist-pyth-version]$  
/cm/local/apps/singularitypro/4.1/bin/singularity shell  
/cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif  
  
3 Singularity> python3 main.py  
ARGS: Namespace(batch_size=64, test_batch_size=1000, epochs=14,  
lr=1.0, gamma=0.7, no_cuda=False, no_mps=False, dry_run=False,  
seed=1, log_interval=10, save_model=False)  
Train Epoch: 1 [0/60000 (0%)] Loss: 2.299825  
[SNIP]  
4 Train Epoch: 14 [58240/60000 (97%)] Loss: 0.044036  
Train Epoch: 14 [58880/60000 (98%)] Loss: 0.002974  
Train Epoch: 14 [59520/60000 (99%)] Loss: 0.001555  
  
Test set: Average loss: 0.0269, Accuracy: 9920/10000 (99%)  
Singularity>
```



```
[mthomas@login01]$ cat run-pytorch-gpu-shared.sh
#!/usr/bin/env bash
#SBATCH --job-name=pytorch-gpu-shared
#### Change account below
#SBATCH --account=use300
#SBATCH --partition=gpu-shared
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=16
#SBATCH --mem=90G
#SBATCH --gpus=1
#SBATCH --time=00:30:00
#SBATCH --output=pytorch-gpu-shared.o%j.%N
declare -xr SINGULARITY_MODULE='singularitypro/3.11'

module purge
module load "${SINGULARITY_MODULE}"
module list
#printenv

#### We will run from the scratch directory local to the node
cd /scratch/$USER/job_${SLURM_JOBID}

time -p singularity exec --bind /expance,/scratch --nv
/cm/shared/apps/containers/singularity/pytorch/pytorch_2.2.1-openmpi-4.1.6-mofed-5.8-2.0.3.0-cuda-12.1.1-
ubuntu-22.04.4-x86_64-20240412.sif python3 $SLURM_SUBMIT_DIR/main.py

#### List scratch directory
ls /scratch/$USER/job_${SLURM_JOBID}
```

Running Jobs Using the Batch Queue



Running Jobs Using the Batch Queue

```
mthomas@login01]$ cat run-pytorc
```

```
[mthomas@login01]$ sbatch run-pytorch-gpu-shared.sh
Submitted batch job 37743866
[mthomas@login01]$ squeue -u mthomas
  JOBID PARTITION   NAME     USER ST   TIME  NODES NODELIST(REASON)
   37743866 gpu-share pytorch-  mthomas PD    0:00      1 (Priority)
[mthomas@login01]$ [mthomas@login01]$ squeue -u mthomas
  JOBID PARTITION   NAME     USER ST   TIME  NODES NODELIST(REASON)
   37743866 gpu-share pytorch-  mthomas R    0:19      1 exp-6-58
[mthomas@login01]$ ll
total 64
drwxr-xr-x 3 mthomas use300   8 Mar 25 23:53 .
drwxr-xr-x 4 mthomas use300   4 Mar 24 00:08 ..
drwxr-xr-x 3 mthomas use300   3 Mar 25 21:16 data
-rw-r--r-- 1 mthomas use300 5669 Mar 25 22:40 main.py
-rw-r--r-- 1 mthomas use300 2969 Mar 25 23:54 pytorch-gpu-shared.o37743866.exp-6-58
-rw-r--r-- 1 mthomas use300 399 Mar 25 21:15 README.txt
-rw-r--r-- 1 mthomas use300 874 Mar 25 21:44 run-pytorch-cpu-shared.sh
-rw-r--r-- 1 mthomas use300 835 Mar 25 21:44 run-pytorch-gpu-shared.sh
```



Hands-on: Reusable Resources

- Clone AI Unlocked Repo
- Navigate to the Track1-AI-Jobs folder
- Find the "containers" folder
- Repeat jobs as shown in slides x-z using the PyTorch MNIST main.py file
 - Interactive: CLI (Command line) on login node; CLI on Expanse interactive node
 - Batch Script
 - Notebooks: using galyeo on Expanse or the OOD portal



Strategies for Optimizing Resource Usage



Optimizing Resource Usage: 5 Key Strategies

- Strategies to Maximize Computational Efficiency
- Strategies to Reduce Runtime
- Strategies for Parallelization
- Strategies for Batching
- Strategies for Effective Memory Usage



Strategy 1: Maximize Computational Efficiency

Maximizing computational efficiency ensures algorithms use least amount of computational power for best performance

STRATEGY	WHAT
<i>Model Pruning</i>	<p><i>What:</i> Remove unimportant neurons (fundamental processing units) or weights (strength of connections between neurons)</p> <p><i>Why:</i> Reduces the amount of computation and memory overhead required.</p> <p><i>Example:</i> CNN pruning can be used to eliminate neurons with very small weights, thus reducing the number of multiplications and additions during inference.</p>
<i>Quantization</i>	<p><i>What:</i> Reduce precision of the model's weights and activations (e.g., from 32-bit floating point to 16-bit or 8-bit integers).</p> <p><i>Why:</i> Reduces memory computational requirements, making operations faster on specialized hardware like GPUs or TPUs.</p> <p><i>Example:</i> TensorFlow Lite and PyTorch support quantization techniques that can reduce model size and improve performance on embedded systems.</p>
<i>Efficient Layer Operations</i>	<p><i>What:</i> Replace expensive operations (e.g., large matrix multiplications) with more efficient operations.</p> <p><i>Why:</i> Certain operations like convolutions and matrix multiplications can be optimized using techniques like Winograd's algorithm or FFT-based convolution.</p> <p><i>Example:</i> Convolution operations in CNNs can be sped up using FFT (Fast Fourier Transform) methods, especially for large images.</p>
<i>Transfer Learning</i>	<p><i>What:</i> Use pre-trained models and fine-tune them for your task.</p> <p><i>Why:</i> Reduces the amount of training needed from scratch, thus reducing both computational time and resource usage.</p> <p><i>Example:</i> Fine-tuning pre-trained ResNet model on custom dataset can save time/resources compared to training a model from scratch.</p>



Strategy 2: Reduce Runtime

Reducing the time it takes for an AI/ML model to process data and make predictions or decisions.

STRATEGY	WHAT
Asynchronous Training	<p><i>What:</i> Use asynchronous updates to reduce waiting times for gradient synchronization.</p> <p><i>Why:</i> In distributed training, you avoid bottlenecks where all workers must sync up before continuing.</p> <p><i>Example:</i> Asynchronous stochastic gradient descent (SGD) can be used in distributed deep learning, where workers update parameters without waiting for all to complete.</p>
Early Stopping	<p><i>What:</i> Stop training once the model's performance stops improving.</p> <p><i>Why:</i> Saves time by preventing unnecessary computation when a model has already converged.</p> <p><i>Example:</i> Implement early stopping based on validation accuracy, where if the accuracy plateaus for a certain number of epochs, training is halted.</p>
Optimized Gradient Descent Algorithms	<p><i>What:</i> Use more efficient variants of gradient descent (Adam, AdaGrad, RMSProp) that adjust learning rates automatically.</p> <p><i>Why:</i> These methods can converge faster than vanilla SGD by dynamically adapting the learning rate during training.</p> <p><i>Example:</i> Using Adam optimizer (Adaptive Moment Estimation) can speed up convergence compared to standard SGD, especially in the presence of sparse gradients.</p>
Hyperparameter Optimization	<p><i>What:</i> Use automated tools (Optuna, Ray Tune) to optimize hyperparameters efficiently rather than manually tuning them.</p> <p><i>Why:</i> Optimizing hyperparameters can lead to faster convergence and better performance, saving computation time.</p> <p><i>Example:</i> Optimizing learning rate, batch size, network architecture with Bayesian Optimization methods TO reduce training time.</p>



Strategy 3: Parallelization

Parallelization helps divide workloads across multiple processors or systems, improving efficiency and reducing time taken to train models or process data.

STRATEGY	WHAT
Data Parallelism	<p><i>What:</i> Split the dataset into smaller chunks, each processed by a separate worker. Each worker computes gradients independently and they are then averaged/synchronized.</p> <p><i>Why:</i> Distributes the training load, speeding up training.</p> <p><i>Example:</i> PyTorch Distributed Data Parallelism / TensorFlow Mirrored Strategy parallelize training across multiple GPUs.</p>
Model Parallelism	<p><i>What:</i> Divide the model into different parts, where each part runs on a different device or processor.</p> <p><i>Why:</i> Useful for very large models that do not fit into a single device's memory.</p> <p><i>Example:</i> In large models (like GPT-3), different layers or parts of the model can be distributed across multiple GPUs, each handling a specific section of the model.</p>
Tensor Parallelism	<p><i>What:</i> Partition large tensors (matrices or arrays) across multiple devices and perform operations in parallel.</p> <p><i>Why:</i> Helps scale the computation of large models that cannot fit on a single device.</p> <p><i>Example:</i> Horovod or TensorFlow's XLA implement tensor parallelism to distribute tensor operations across multiple GPUs.</p>
Pipeline Parallelism	<p><i>What:</i> Split a deep learning model into smaller stages, where each stage is processed in parallel and results are passed along the pipeline.</p> <p><i>Why:</i> Improves throughput by reducing idle times for each processor and using them in parallel.</p> <p><i>Example:</i> In language models, 1 GPU can process 1 layer of the model while another GPU processes a different layer.</p>



Strategy 4: Batching or Batch Computing

Batch computing involves processing multiple inputs at once rather than one at a time, which maximizes computational throughput.

STRATEGY	WHAT
Batching for Training	<p>What: Process multiple training samples in parallel (batch-wise) instead of training with a single sample at a time.</p> <p>Why: Reduces the overhead of repetitive computation and allows better hardware utilization.</p> <p>Example: Using larger batch sizes (e.g., 64, 128, 256) allows for more parallelism and faster computations on GPUs, leading to faster training.</p>
Mini-Batch Gradient Descent	<p>What: Break the dataset into smaller batches for gradient descent updates rather than using the entire dataset (which is computationally expensive).</p> <p>Why: Speeds up training, provides a good trade-off between training time and model convergence.</p> <p>Example: Instead of using Full Batch Gradient Descent, use Mini-Batch Gradient Descent with batches of size 32 or 64 to speed up learning.</p>
Dynamic Batching	<p>What: Dynamically adjust the batch size depending on the system's load and available memory.</p> <p>Why: Helps maintain an optimal processing load and ensures efficient resource utilization.</p> <p>Example: In a real-time inference scenario, you can dynamically adjust the batch size of incoming requests based on system capacity.</p>



Strategy 5: Effective Memory Usage

Optimizing memory usage ensures that models fit into memory constraints and run efficiently.

STRATEGY	WHAT
Model Quantization	<i>What:</i> Reducing the precision of the model's weights and activations (e.g., from 32-bit to 8-bit integers). <i>Why:</i> Reduces memory usage and speeds up computation without sacrificing too much accuracy. <i>Example:</i> TensorFlow Lite provides tools for converting models to use quantized weights, significantly reducing the model's memory footprint.
Memory-Mapped I/O	<i>What:</i> Use memory-mapped files to load large datasets into memory in chunks, allowing for data to be accessed on demand without loading the entire dataset. <i>Why:</i> Reduces memory usage and allows handling of large datasets without overloading system memory. <i>Example:</i> Using HDF5 file format for large datasets allows them to be memory-mapped for fast access.
Gradient Checkpointing	<i>What:</i> Instead of storing all intermediate activations during the forward pass, recompute them during the backward pass to save memory. <i>Why:</i> Reduces memory usage, especially in models with a large number of layers. <i>Example:</i> Libraries like PyTorch and TensorFlow provide implementations of gradient checkpointing to save memory during training.
Efficient Data Structures	<i>What:</i> Use efficient data structures like sparse matrices when your data is sparse (contains many zeros). <i>Why:</i> Save memory by only storing non-zero elements and corresponding indices. <i>Example:</i> Using <code>scipy.sparse</code> for sparse matrices to store data in a more memory-efficient manner.



Summary of Key Differences and When to Use Each Resource

Factor	CPU	GPU
Workload Type	General-purpose computing, low-parallel tasks	Highly parallel tasks, deep learning
Core Count	Few powerful cores (4–16)	Thousands of small cores
Parallelism	Limited, optimized for serial execution	Massive parallelism for simultaneous computation
Performance	Strong single-threaded performance	Better performance on large datasets and complex models
Memory	High-speed RAM, smaller capacity	High-bandwidth VRAM, large capacity
Power Efficiency	More power-efficient for small tasks	Less power-efficient for intensive workloads
Best Use Cases	Small models, inference, general-purpose tasks	Training deep learning models, large data processing

Conclusion:

- **Use CPUs** for light AI workloads, real-time inference on smaller models, and general-purpose applications.
- **Use GPUs** for training deep learning models, handling large datasets, and performing high-throughput, parallelizable computations.
- The choice of resource also depends on factors like budget, the scale of your AI workload, and whether you're working on research or production deployment. GPUs are generally more expensive but offer significantly better performance for deep learning tasks, making them essential for most AI research and production workloads.



Thank You!

Q&A

If you have problems, please contact **consult@sdsc.edu**

<https://github.com/sdsc-complecs/interactive-computing/>



Resources

- [AI Unlocked GitHub Repo](#), including this presentation:
 - <https://github.com/sdsc-complecs/interactive-computing/>
- SDSC Training Resources
 - HPC/CI On-Demand Training Catalog: <https://www.sdsc.edu/education/on-demand-learning/index.html>
 - [HPC Example Code](#): <https://github.com/sdsc-hpc-training-org/hpctr-examples>
 - Hands-on HPC/CI Training: <https://hpc-training.sdsc.edu/>
 - Running notebooks
 - Using galileo: <https://github.com/mkandes/galileo>
 - [Expanse Notebooks Collection](#): <https://github.com/sdsc-hpc-training-org/Expanse-Notebooks>
 - Expanse :
 - Project page: expanse.sdsc.edu
 - User Guide: https://expanse.sdsc.edu/support/user_guides/expanse.html
- Problems? Contact consult@sdsc.edu