Center of Excellence: Supercomputing for AI & Big Data

Distributed Artificial Intelligence

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center

Outline

Importance of AI

Basics of AI

Problems

Calculation AI Models and Requirements

Solutions: Parallel Processing

Conclusion Spring School

Past Present and Future

AGI

Artificial General Intelligence

ANI

Artificial Narrow Intelligence

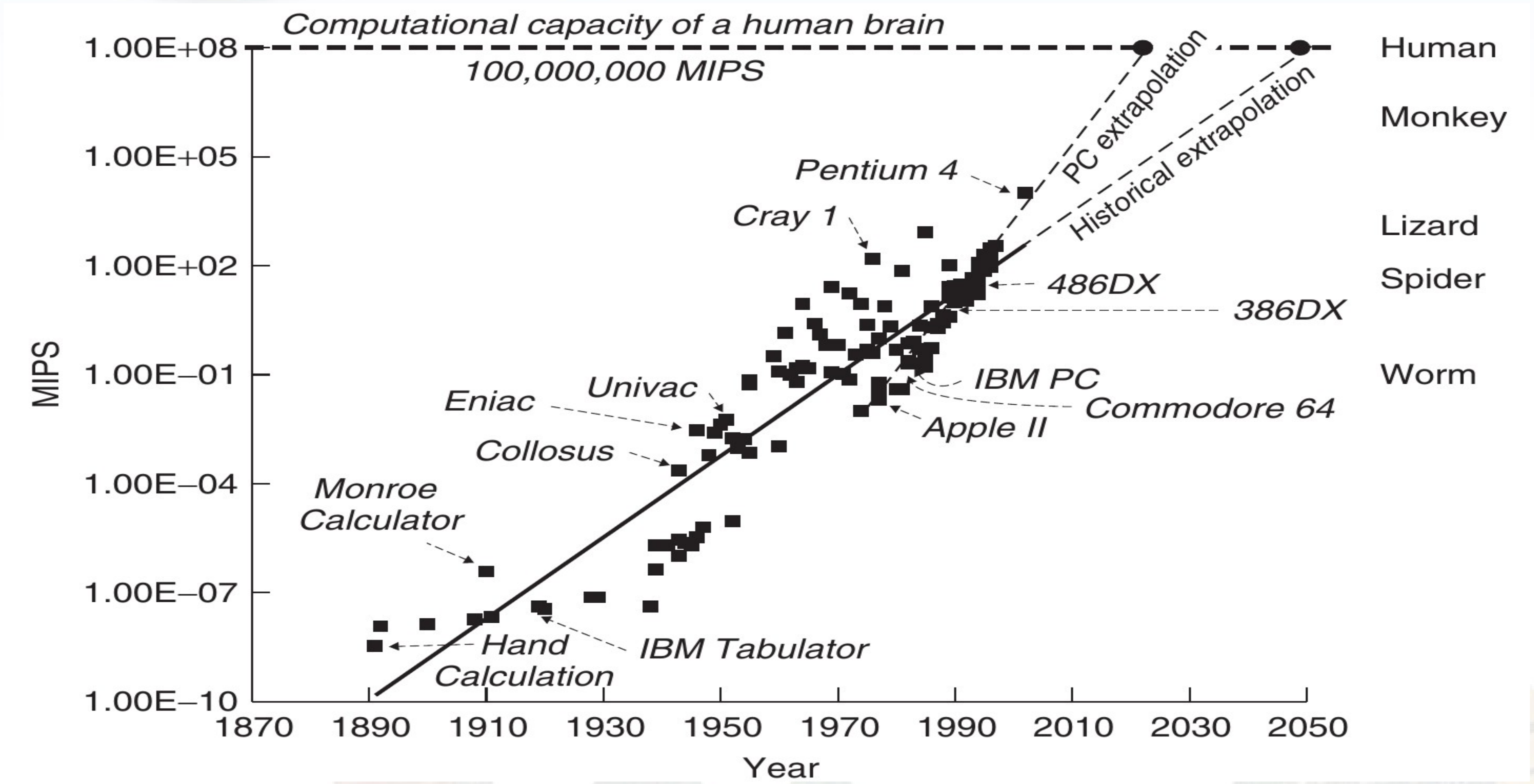
Information/Big Data

Complex Adaptive Algorithms

Computing Resources

**"Methods that scale with computation are the future of Artificial Intelligence"
— Rich Sutton,**

Computational Capability ?

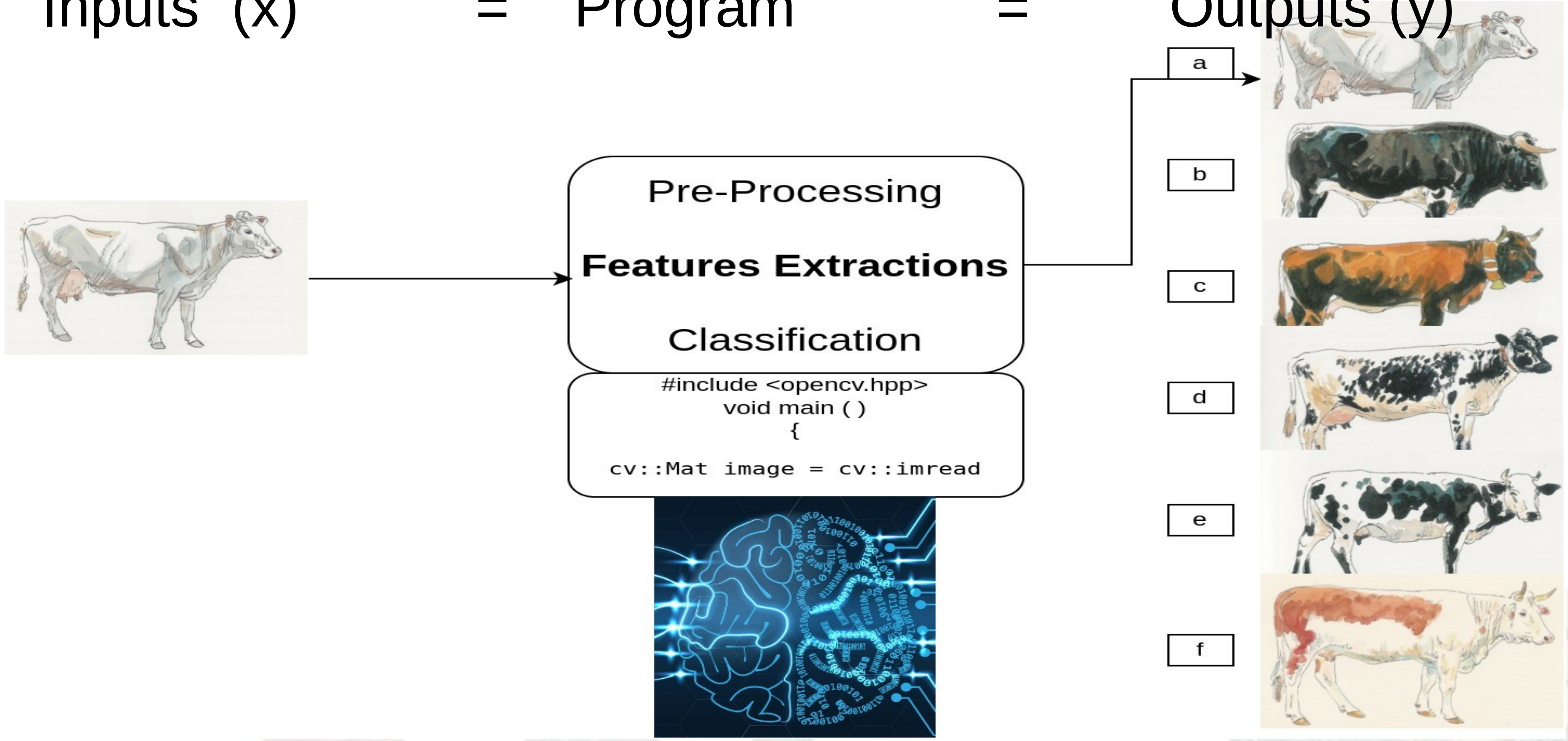


It is estimated that sometime between the years **2025 and 2050**, a **personal computers** will exceed the calculation power of a human brain.

What is AI? : Conventional Method

Inputs (x) = Algorithm = Decisions

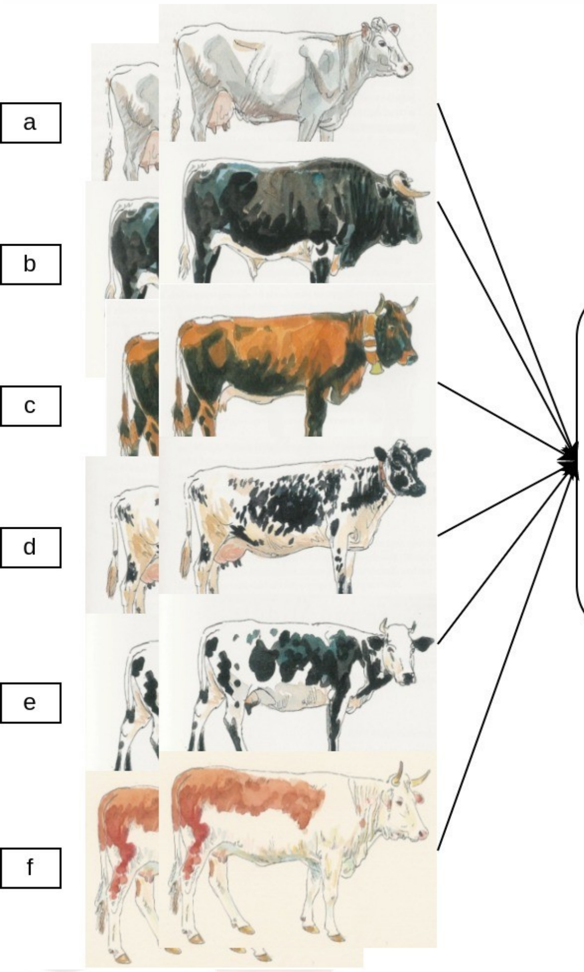
Inputs (x) = Program = Outputs (y)



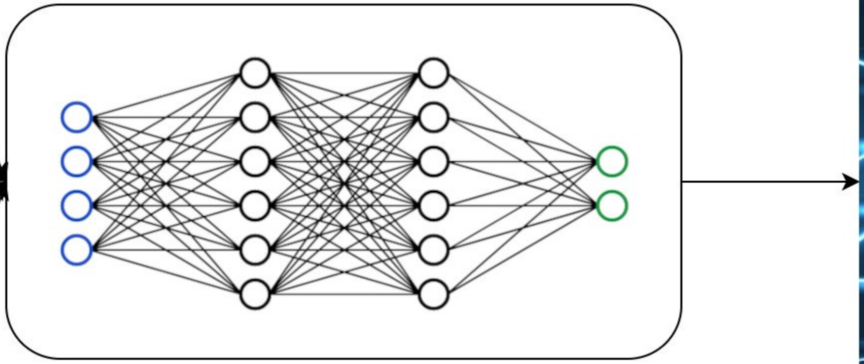
```
#include <opencv.hpp>
void main ()
{

cv::Mat image = cv::imread
```

What is AI?

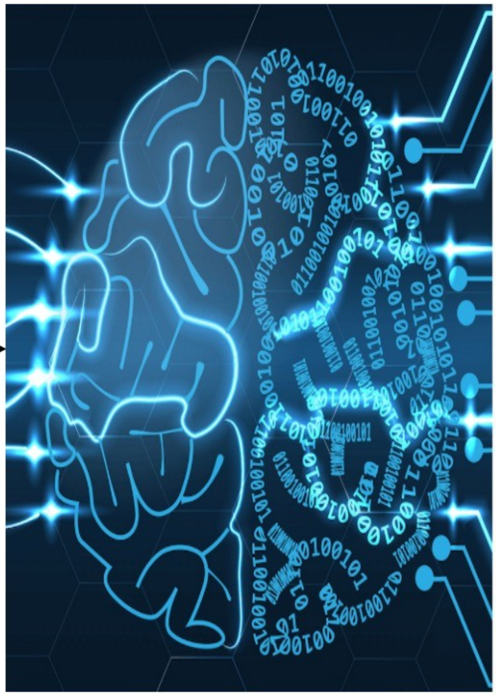
(Labeled) Outputs (y) = F (x) (AI Model) => **Program**



Edges => Objects => Shapes



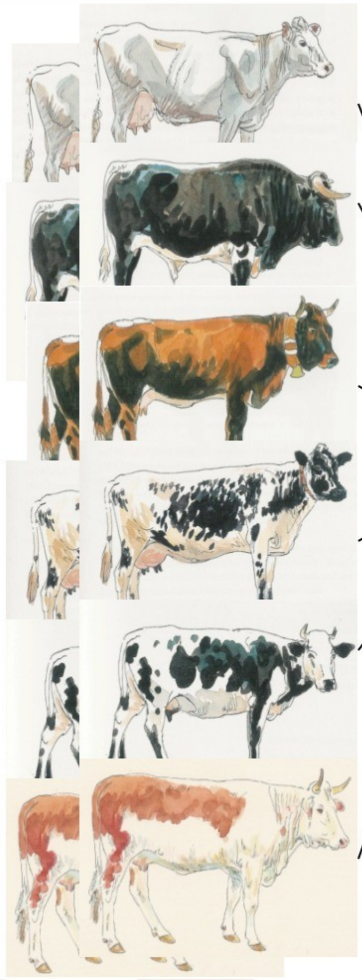
Parameters =>
(Weights, Biases, AF, Embedding, Hyper-parameters)



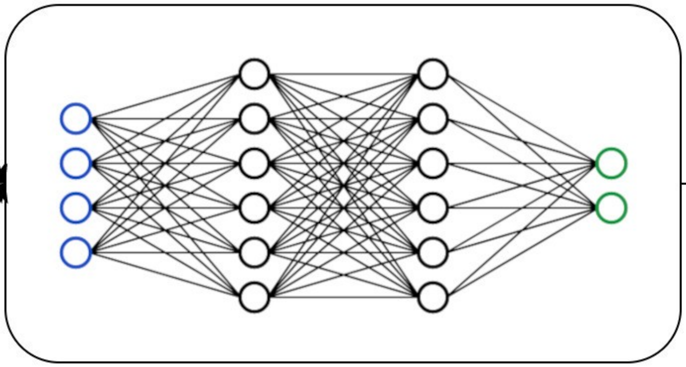
What is AI?

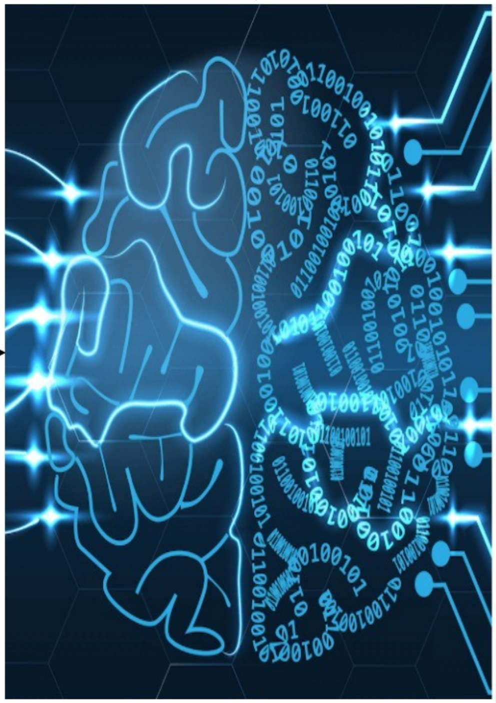
Inputs (y) = F (x) (AI Model) => **Program**

Reward



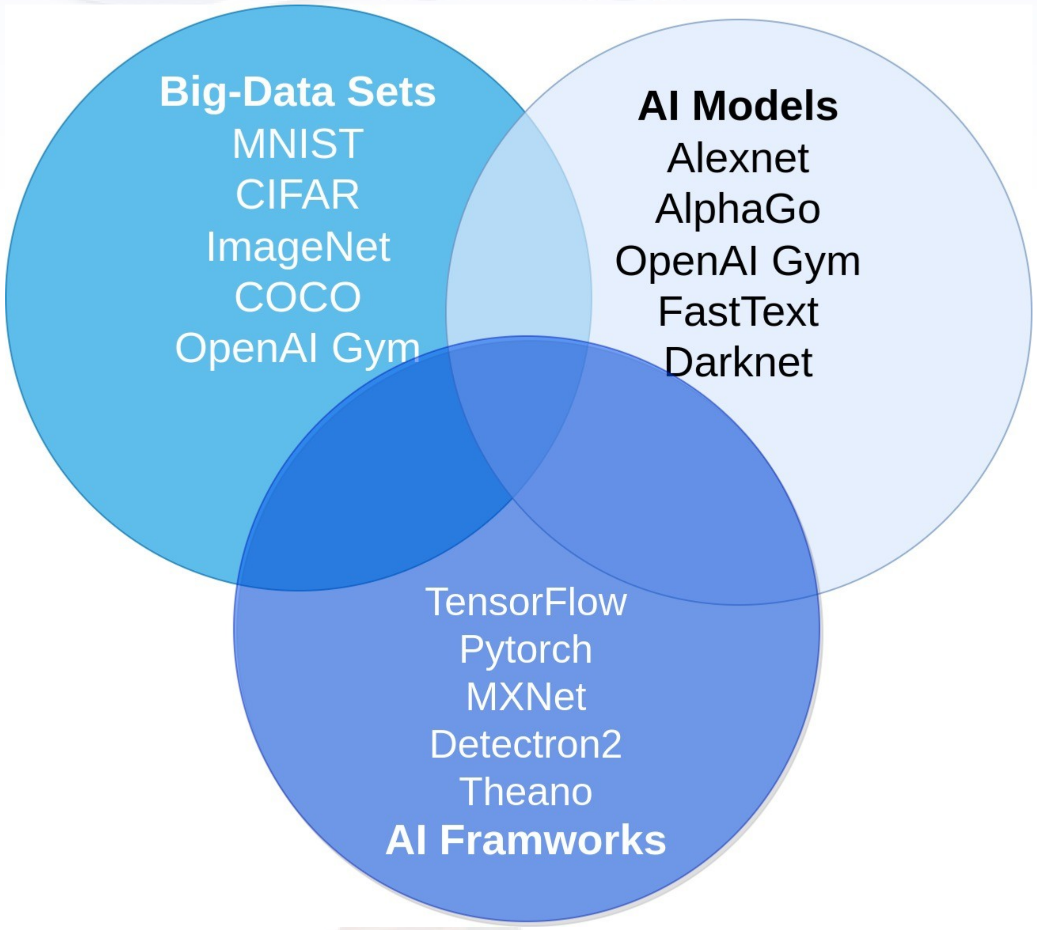
Edges => Objects => Shapes





Parameters =>
(Weights, Biases, AF, Embedding,
Hyper-parameters)

Open-Source

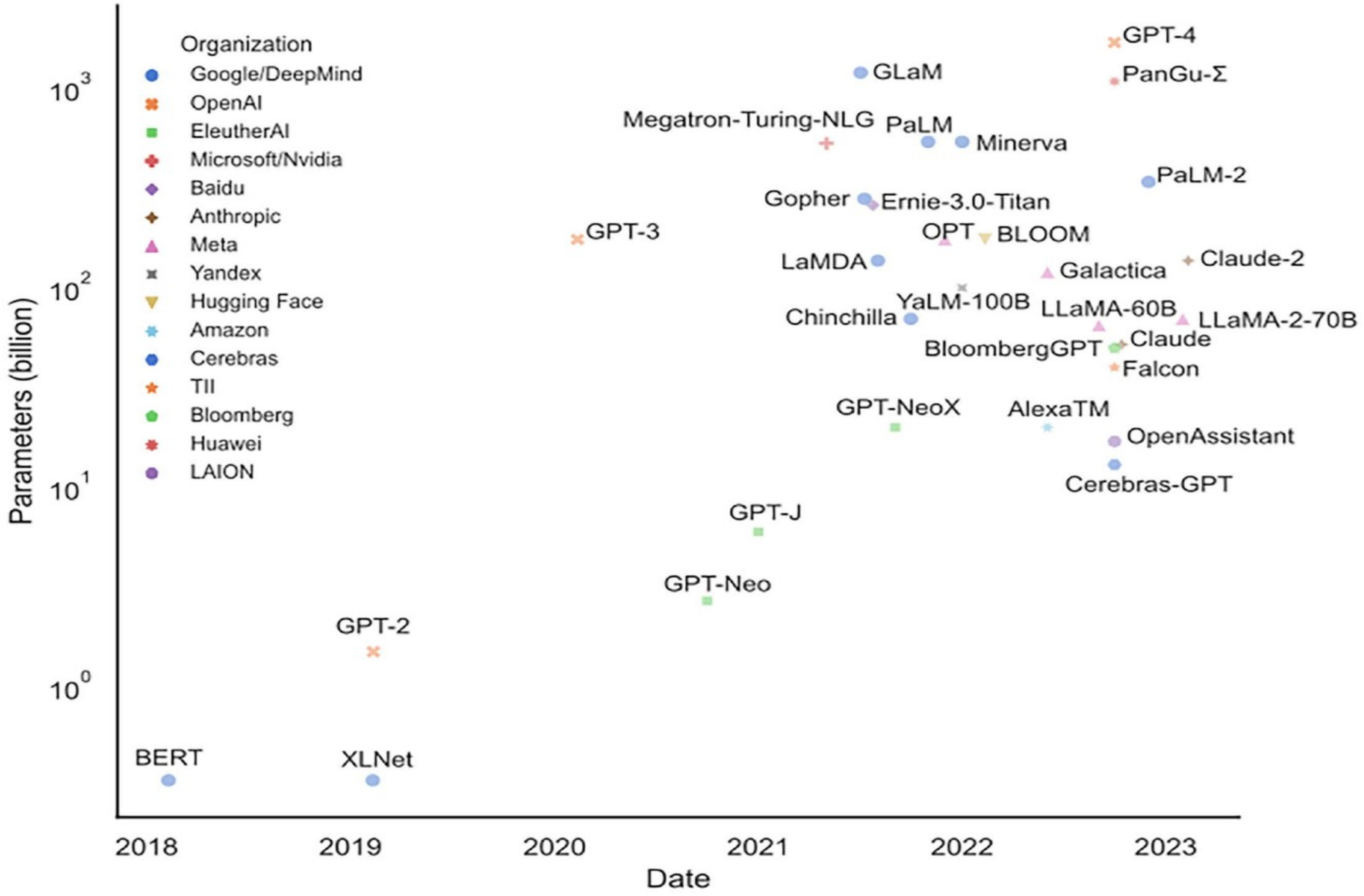


+

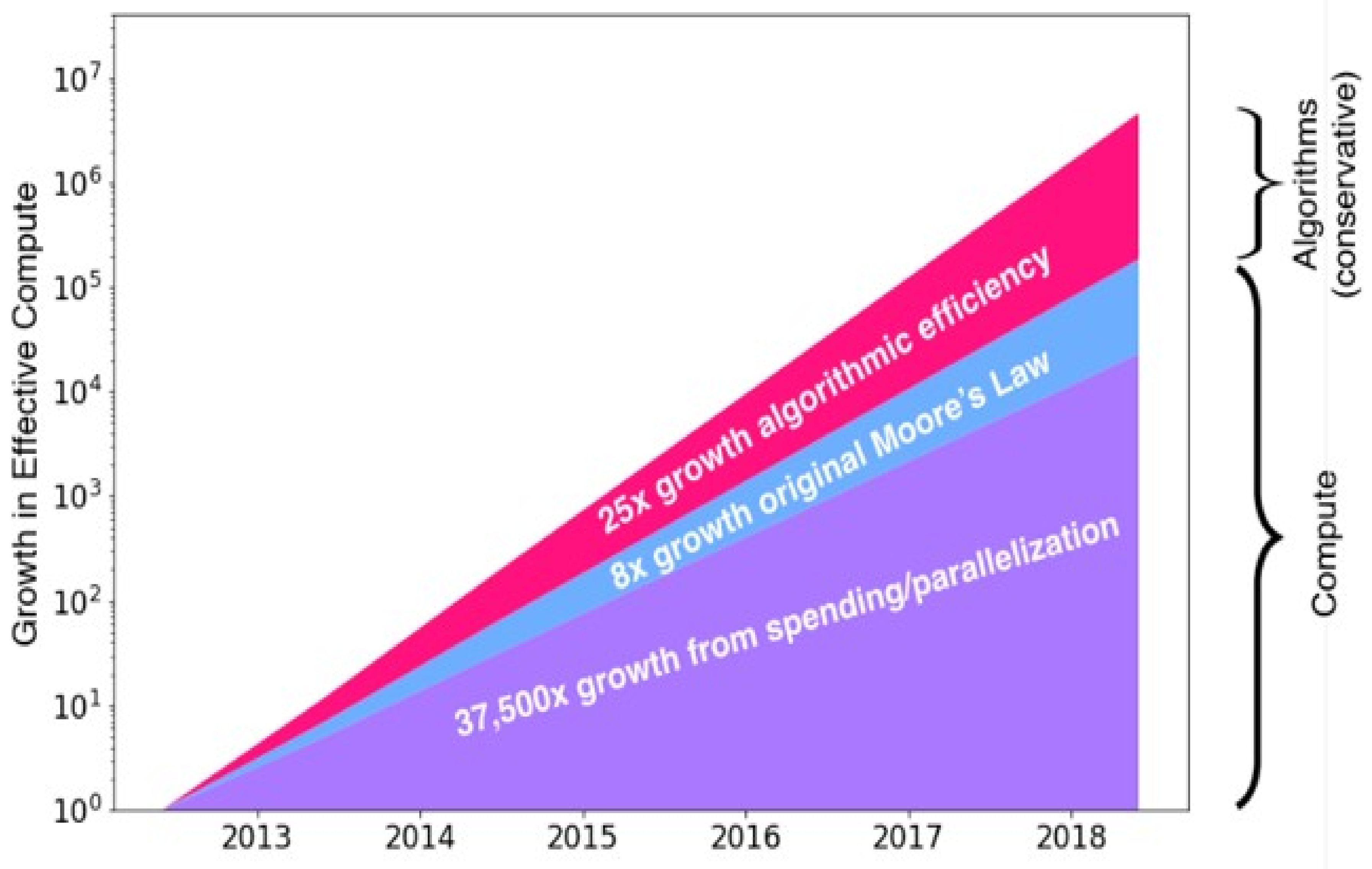
Business Strategy

Big datasets, and open-source DL framework, play an important role to create “big” algorithms.

AI Model Parameters



Efficiency Growth



https://arxiv.org/pdf/2005.04305.pdf

ChatGPT: Use Case

- The cost to train 175 billion parameter language model is nearly $12 million dollars (200x the price of GPT-2)

OpenAI GPT-3 paper.

ChatGPT: Use Case

- The cost to train 175 billion parameter language model is nearly $12 million dollars (200x the price of GPT-2)
- Google presents a model for Multilingual translation quality with 600 Billion parameters. The training takes 22 Years on 1TPU.

OpenAI GPT-3 paper.

ChatGPT: Use Case

- The cost to train 175 billion parameter language model is nearly $12 million dollars (200x the price of GPT-2)
- Google presents a model for Multilingual translation quality with 600 Billion parameters. The training takes 22 Years on 1TPU.
- While distributing the training over 2048 TPUs, achived results in 4 days.

OpenAI GPT-3 paper.

AI Requirements?

DL requires **Large Dataset** and **Complex Computations**.

In addition, the large sets of **weights with complex configurations**, which require a **lot of resources** to store both in the **training mode**, and in the **prediction/inference mode**.

- Optimizing the server memory for storing large models?
- Maximizing multiple GPU resources to increase the speed of training?
- Spreading out or distribute the training process in order to train large deep learning model?

AI Applications Requirements:

Counting operations in the model:

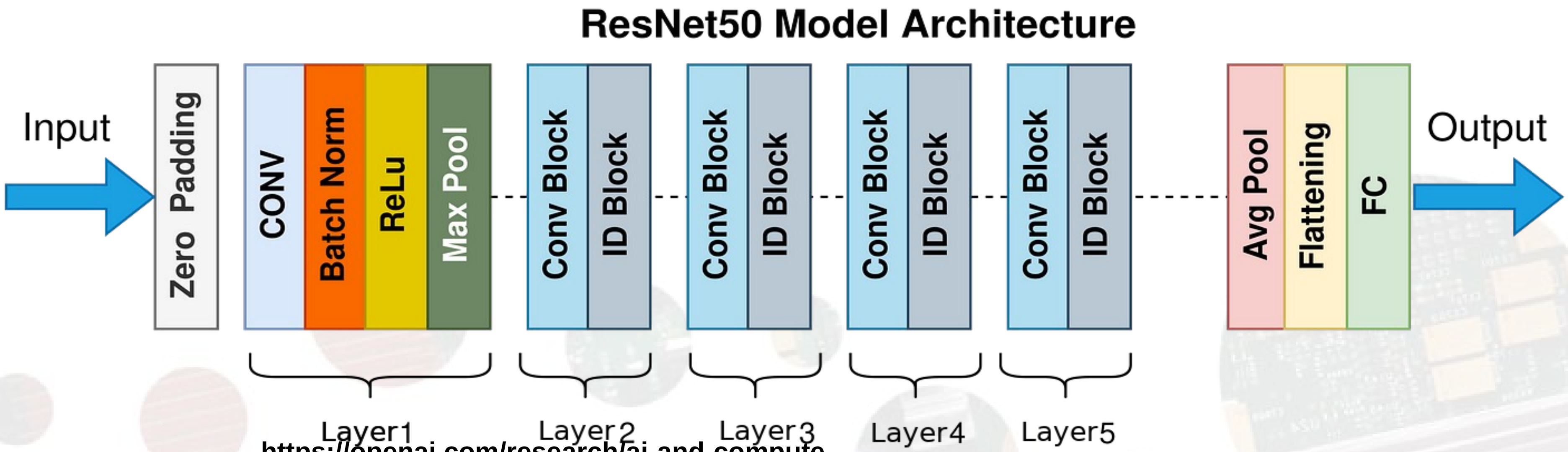
(add-multiplies per forward pass) * (2 FLOPs - add-multiply) * (3 for forward and backward pass) * (number of examples in dataset) * (number of epochs)

= (11.4 * 10^9) * 2 * 3 * (1.2 * 10^6 images) * 128 **(=> Resnet-151 model)**

= 1.050624×10^{19} = 10 EFLOPS

= 10,000 PFLOPS / (24 x 60 x 60) = 0.1157 PFLOP-days

= 115 TFLOPS/day (Cost?, Power?, Maintenance? CE?)



https://openai.com/research/ai-and-compute

AI Applications Requirements:

Counting operations in the model:

(add-multiplies per forward pass) * (2 FLOPs - add-multiply) * (3 for forward and backward pass) * (number of examples in dataset) * (number of epochs)

= (11.4 * 10^9) * 2 * 3 * (1.2 * 10^6 images) * 128 **(=> Resnet-151 model)**

= 10,000 PF = **0.1157 PFLOPS-days = 115 TFLOPS/day**

FLOPS/Byte:

10,000 PFLOPS / ((224 * 224 * 4 * 3) * 1.2M * 128e)

= 108KFLOPS/Byte

https://openai.com/research/ai-and-compute

AI Applications Requirements:

Counting operations in the model:

(add-multiplies per forward pass) * (2 FLOPs - add-multiply) * (3 for forward and backward pass) * (number of examples in dataset) * (number of epochs)

= (11.4 * 10^9) * 2 * 3 * (1.2 * 10^6 images) * 128 (=> Resnet-151 model)

= 10 EF = 0.1157 PFLOPS-days = 115 TFLOPS - day

FLOPS/Byte:

10,000 PFLOPS / ((224 * 224 * 4 * 3) * 1.2M * 128e)

= 108KFLOPS/Byte

Machine Time:

Number of GPUs * (TFLOPS of RTX4070TI) * days trained * estimated utilization

= 2 * (41.58 * TFLOPS) * 1 * 0.33 = 2.366496 EFLOPS-Day

https://openai.com/research/ai-and-compute

AI Applications Requirements:

Counting operations in the model:

(add-multiplies per forward pass) * (2 FLOPs - add-multiply) * (3 for forward and backward pass) * (number of examples in dataset) * (number of epochs)

= (11.4 * 10^9) * 2 * 3 * (1.2 * 10^6 images) * 128 **(=> Resnet-151 model)**

= **10 EF** = 0.1157 PFLOPS-days = **115 TFLOPS - day**

FLOPS/Byte:

10,000 PFLOPS / ((224 * 224 * 4 * 3) * 1.2M * 128e)

= 108KFLOPS/Byte

Machine Time:

Number of GPUs * (TFLOPS of RTX4070TI) * days trained * estimated utilization

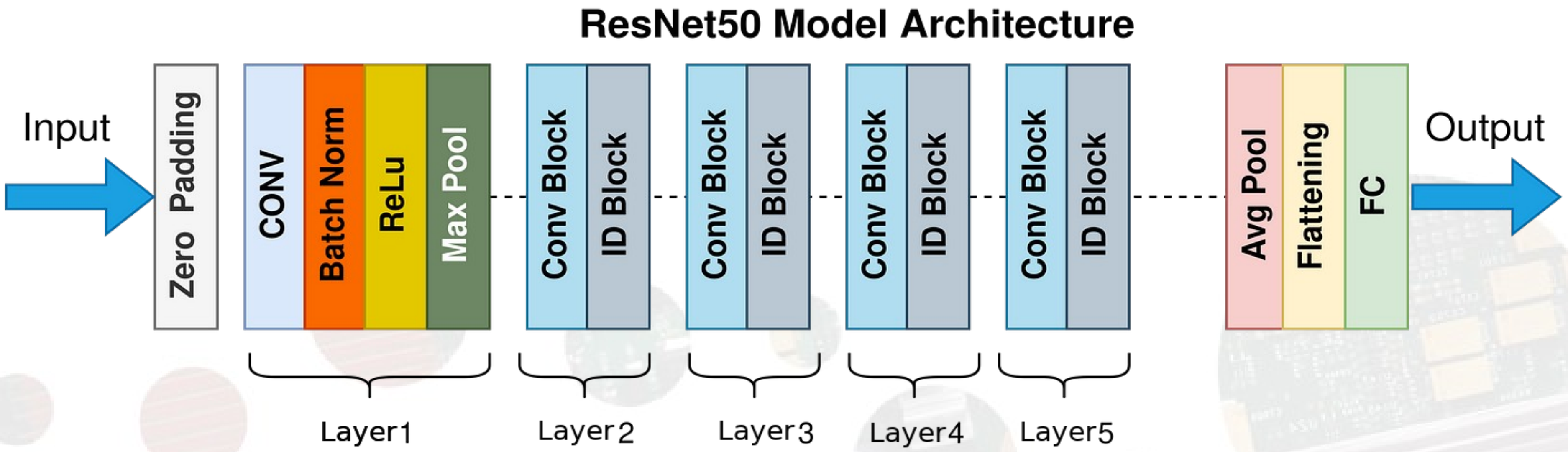
= 2 * (41.58 * TFLOPS) * 1 * 0.33 **= 2.366496 EFLOPS-Day**

= 11.83248 EFLOPS- 5days or 5 Machines @ 1Day

https://openai.com/research/ai-and-compute

Scalable Performance / Accuracy

- Parallelism
  - Data
  - Model
- Training
  - Sync, Async
  - Strategy



Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) is an iterative algorithm for **finding optimal values**, and is one of the most popular algorithms for training in AI.

It involves multiple rounds of training, where the results of each round are incorporated into the model in preparation for the next round.

The rounds can be run on multiple devices either synchronously or asynchronously.

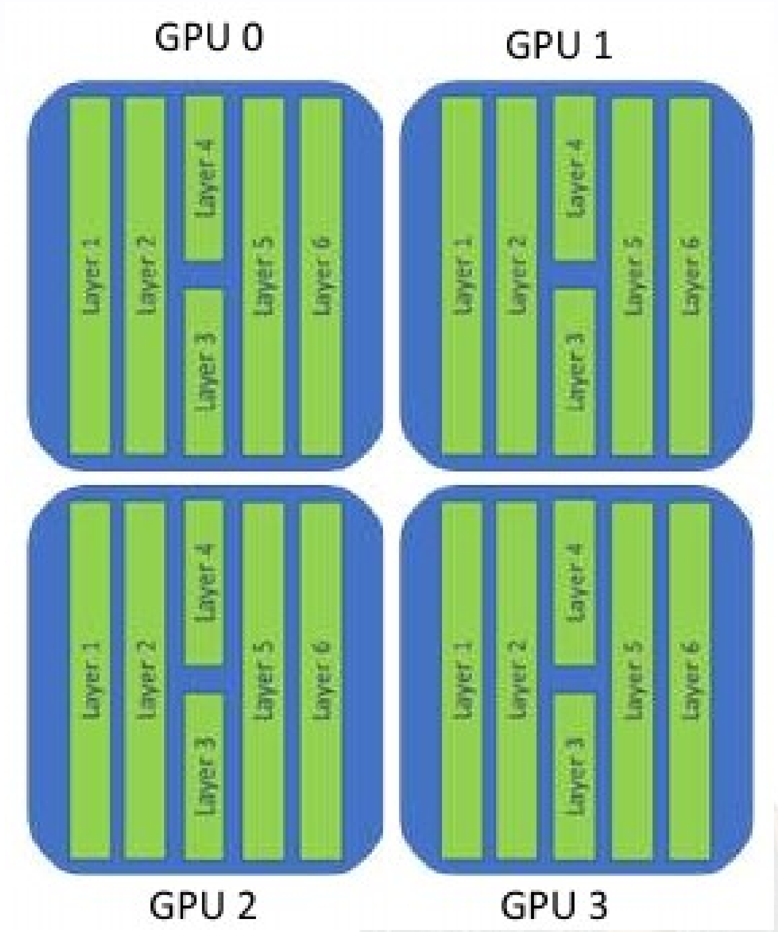
Each SGD iteration runs on a mini-batch of training samples (Facebook had a large mini-batch size of 8,092 images).

Data Parallelism

In this mode, the training data is divided into multiple subsets, and each one of them is run on the same replicated model in a different GPU (worker nodes).

These will need to synchronize the model parameters (or its “gradients”) at the end of the batch computation to ensure they are training a consistent model.

Therefore, each device must send all its changes to all models on the nodes.

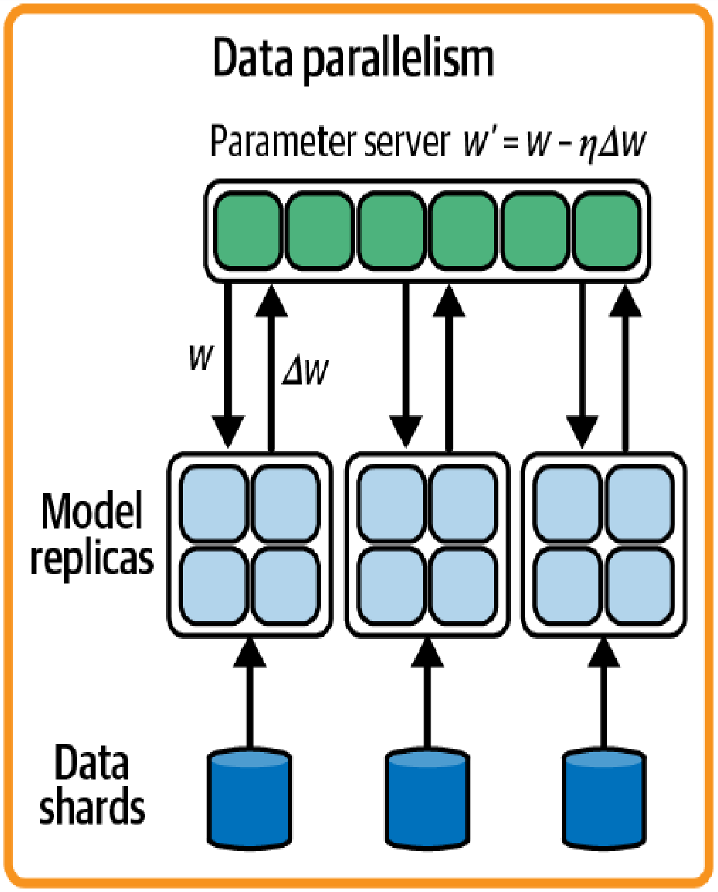


Data Parallelism Features

Scale with the amount of data available, and it speeds up the rate at which the entire dataset contributes to the optimization.

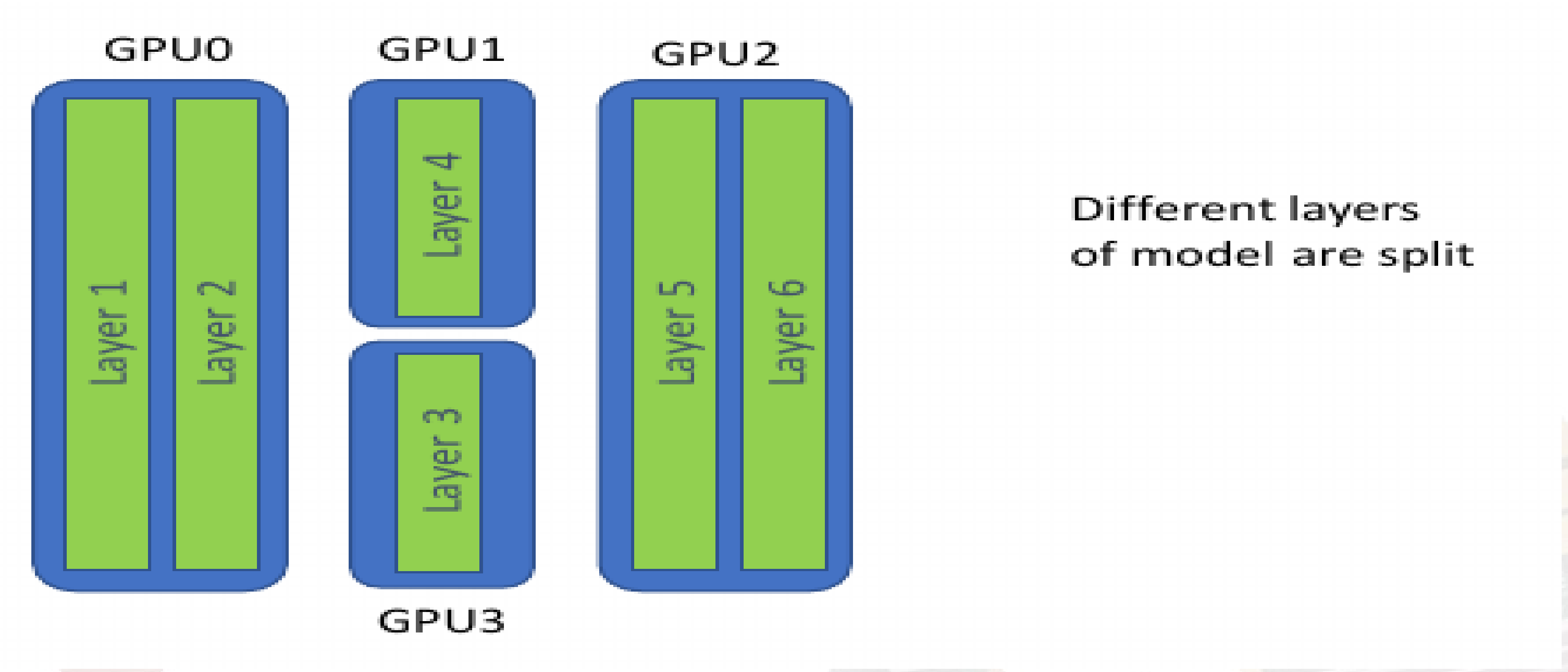
Also, it requires **less communication between nodes**, as it benefits from a high number of computations per weight.

On the other hand, the **entire model needs to fit in the memory** of each node entirely, and thus it is mainly used for speeding computation of convolutional neural networks with large training datasets (CPU-bound problem)



Model Parallelism

If the **Model is too big to fit in a single machine**. Then model parallelism might be a better idea. The most frequent use case is modern natural language processing models such as GPT-2 and GPT-3, which contain billions of parameters (GPT-2 has in fact 1.5 billion parameters).



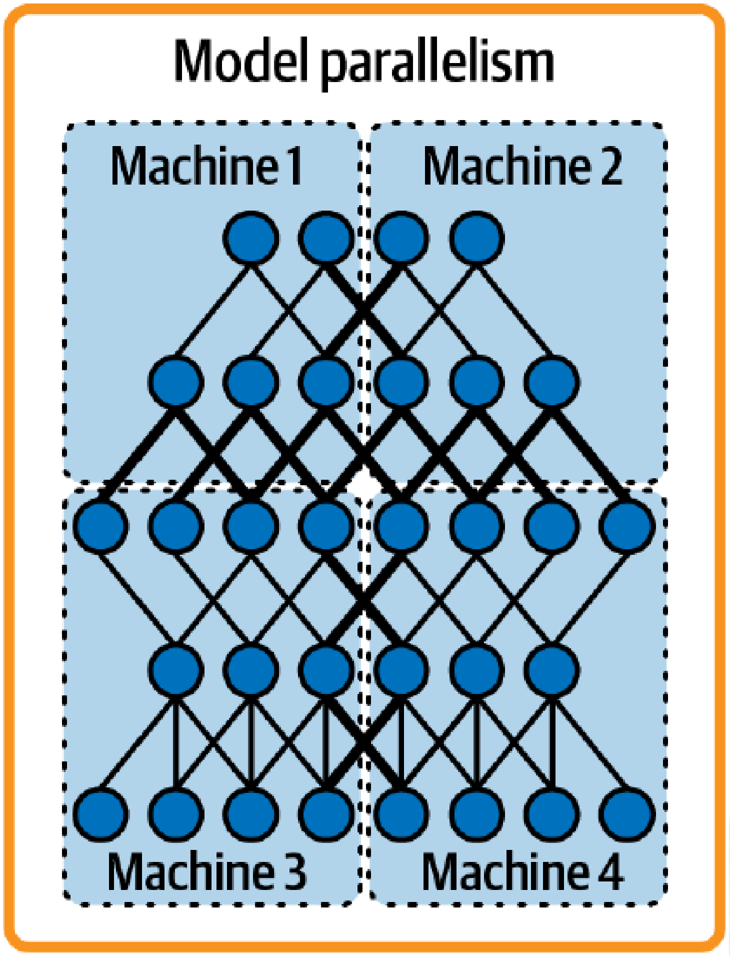
Model Parallelism

Partitions the network layers across multiple GPUs. That is, **each GPU** takes as input the data flowing into a **particular layer**, processes data across several subsequent layers in the neural network, and then sends the data to the next GPU.

This is also known as **Network Parallelism** as the model will be segmented into different parts that can run concurrently, and each one will run on the same training data in different nodes.

It may decrease the **communication needs, as workers need only to synchronize the shared parameters** (usually once for each forward or backward-propagation step)

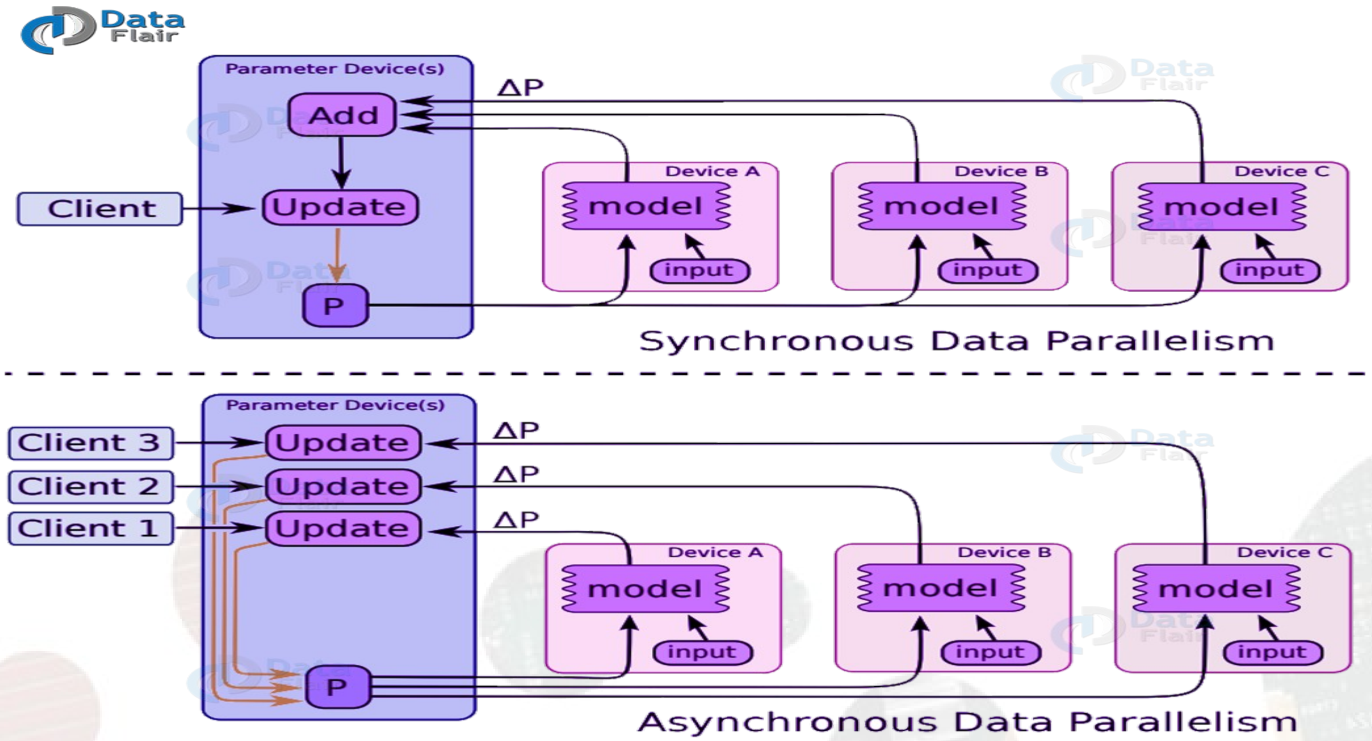
It can work well for **GPUs in a rack that shares a high-speed bus such as NVLink or InfiniBand.** Model parallelism is more tricky to make work as compared with data parallelism.



Types of Training

Sync training, all workers/accelerators train over different slices of input data and aggregate the gradients in each step.

Async training, all workers/accelerators are independently trained over the input data and update variables in an asynchronous manner.



Sync Training

Slices of data transferred to different worker/accelerator.

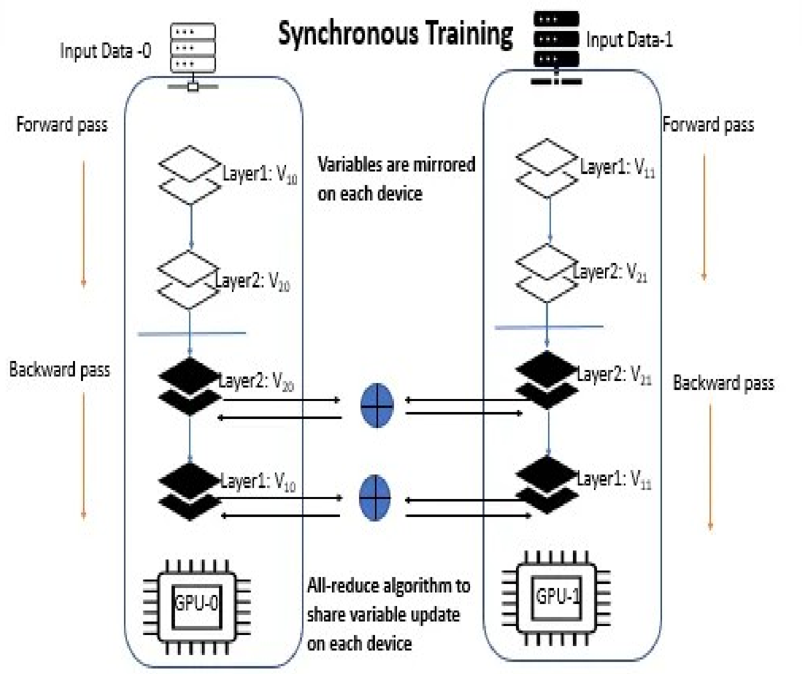
Each device has a full replica of the model and it is trained only on a part of the data. The forward pass begins at the same time in all of them. They all compute a different output and gradients.

Afterwards all the devices are communicating with each other and aggregate the gradients using the **all-reduce algorithm.**

Once the gradients are combined, they are sent back to all of the devices.

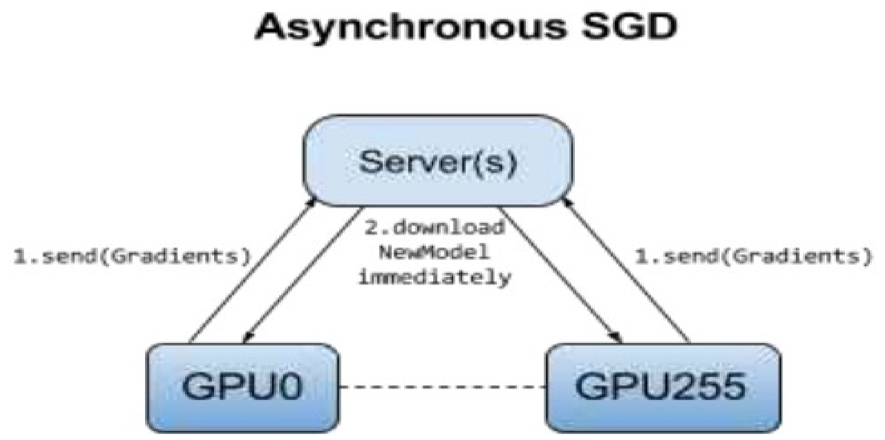
Each device continues with the backward pass, updating the local copy of the weights normally.

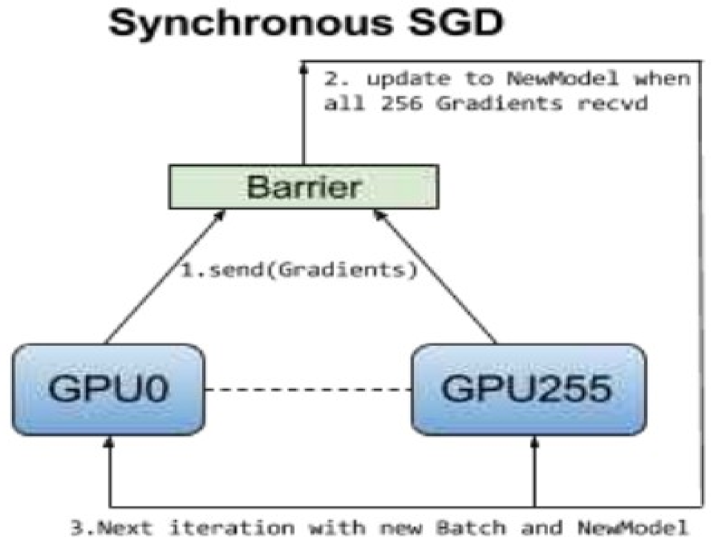
The next forward past doesn't begin until all the variables are updated.



Asynchronous Training

Asynchronous training, the devices **do not wait for the model updates from other devices**. The devices run independently and share results as peers, or communicate through one or more central servers known as "parameter" servers.
Each device runs a loop that reads data, computes the gradients, sends them (directly or indirectly) to all devices, and updates the model to the latest version.




Synchronous SGD
2. update to NewModel when all 256 Gradients recvd
Barrier
1.send(Gradients)
GPU0
GPU255
3.Next iteration with new Batch and NewModel

TensorFlow Distributed

Distributed TensorFlow applications consist of a **cluster containing** one or more parameter **servers and workers.**

The workers calculate gradients during training, and are placed on a GPU.

Parameter servers only need to aggregate gradients and broadcast updates, so they are typically placed on CPUs, not GPUs.

One of the workers, the **chief worker**, coordinates model training, initializes the model, counts the number of training steps completed, monitors the session, saves logs for TensorBoard, and saves and restores model checkpoints to recover from failures. The chief worker also manages failures, ensuring fault tolerance if a worker or parameter server fails. If the chief worker itself dies, training will need to be restarted from the most recent checkpoint.

One disadvantage of Distributed TensorFlow, part of core TensorFlow, is that **programmer has to manage the starting and stopping of servers explicitly.** This means keeping track of the IP addresses and ports of all your TensorFlow servers in your program, and starting and stopping those servers manually.

Tensorflow Strategies

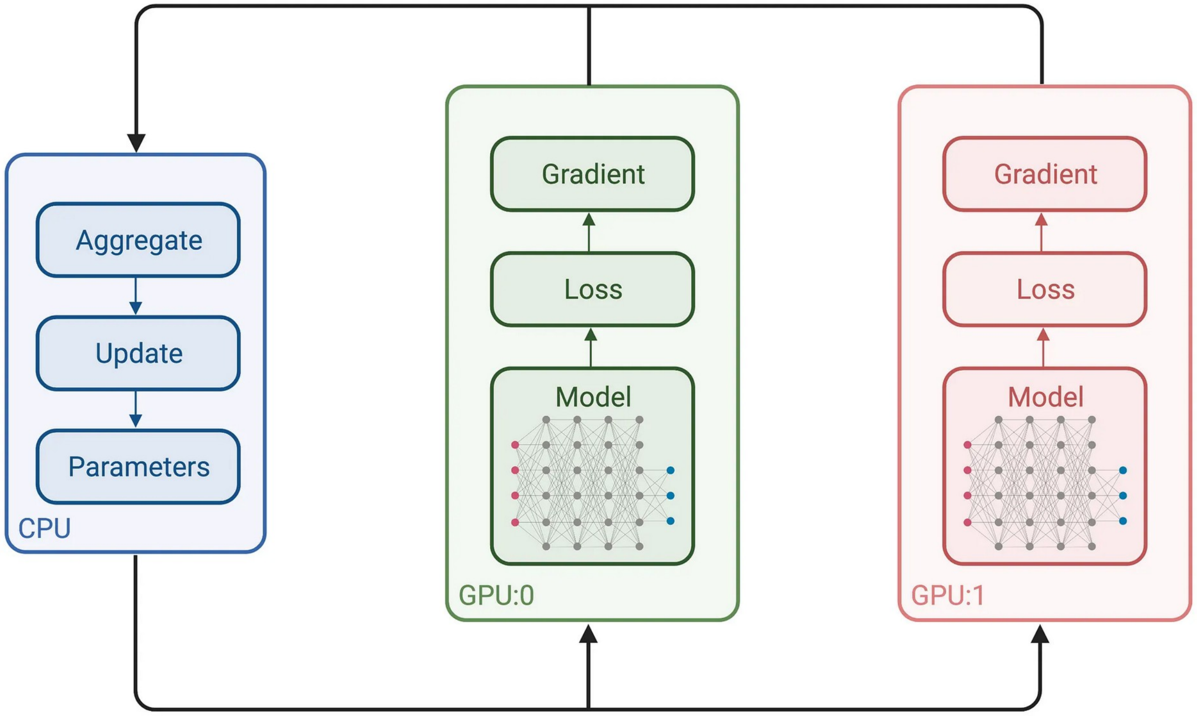
TensorFlow strategies are designed to facilitate the training of machine learning models on distributed computing environments.

- **Data Parallelism**
- **Model Parallelism**
- **Efficient Training on Multiple Devices**
- **Scaling to Multiple Machines**
- **Synchronization and Communication**

Distribute MirroredStategy

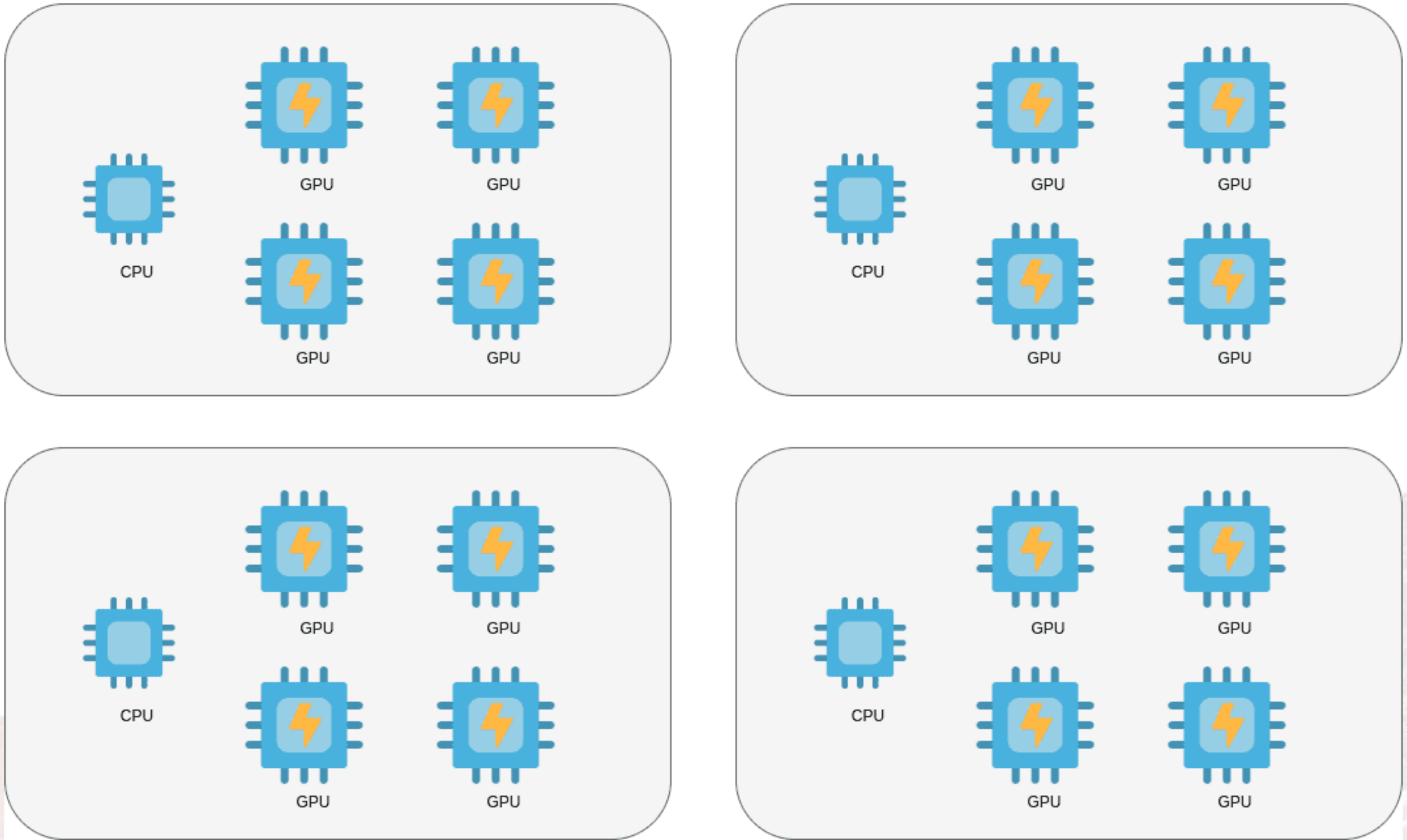
tf.distribute.MirroredStrategy supports **synchronous distributed training on multiple GPUs on one machine**. It creates one replica per GPU device. Each variable in the model is mirrored across all the replicas. Together, these variables form a single conceptual variable called MirroredVariable.

These variables are kept in sync with each other by applying identical updates.



MultiWorkerMirroredStrategy

MultiWorkerMirroredStrategy implements training on many workers. Again, it creates copies of all variables across all workers and runs the **training in a sync manner**.

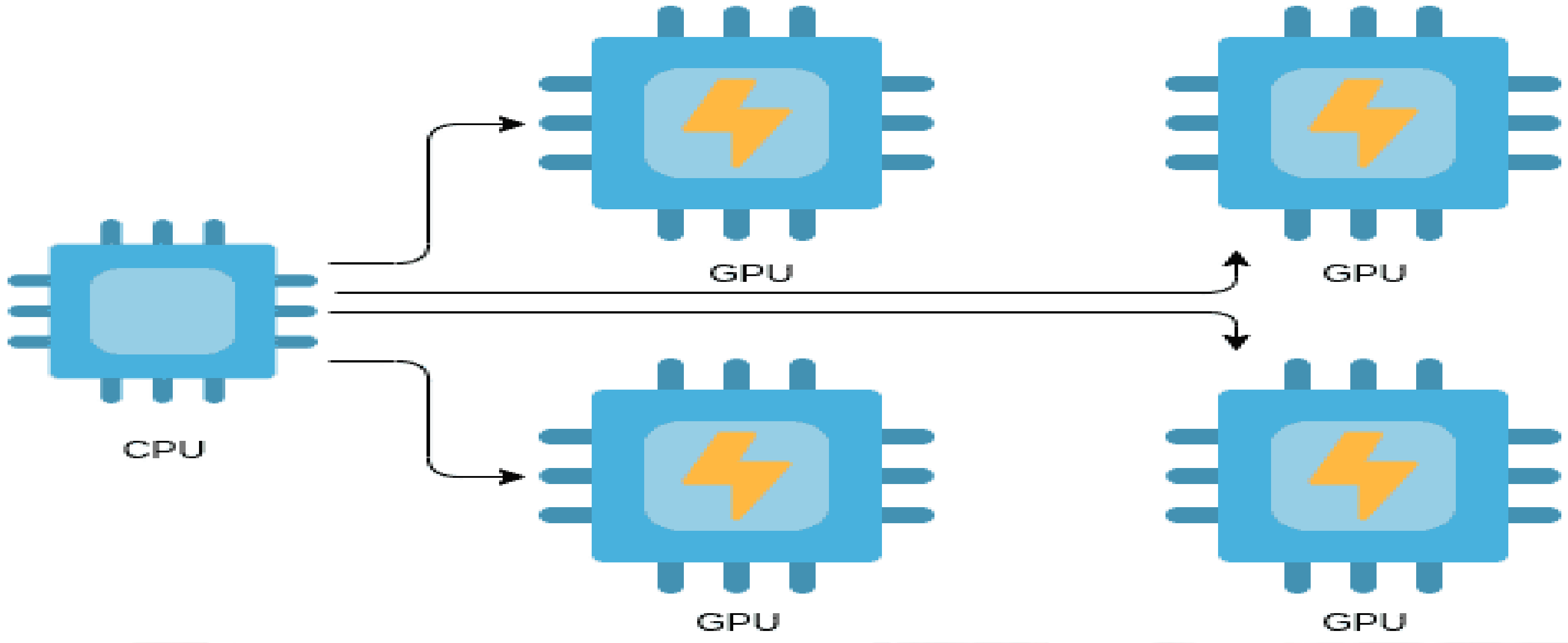


Central Storage Stratigey

Applies only to environments when a single machine with multiple GPUs.

When the GPU’s might not be able to store the entire model, the CPU works as central storage unit which holds the global state of the model.

The variables are not mirrored into the different devices but they are all in the CPU.

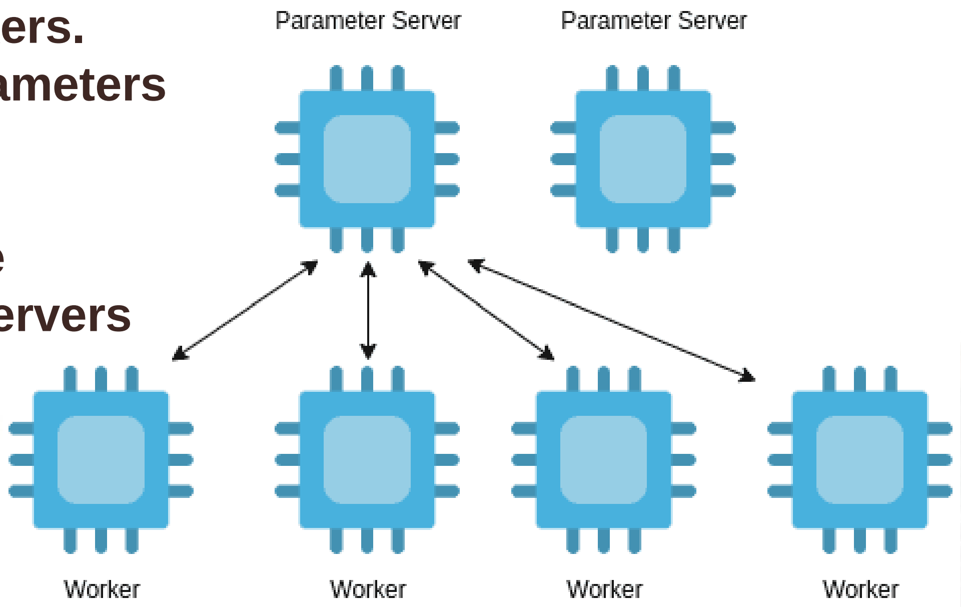


Parameter server architecture

Asynchronous Training: The parameter servers utilizes parallel SGD, the algorithm starts by broadcasting the model to the workers (devices).

In each training iteration, each worker reads its own split from the mini-batch, computing its own gradients, and sending those gradients to one or more parameter servers. The parameter servers aggregate all the gradients from the workers and wait until all workers have completed before they calculate the new model for the next iteration, which is then broadcast to all workers.

- **Replicate the model in all of the workers.**
- **Each training worker fetches the parameters from the parameter servers.**
- **Performs a training loop.**
- **Once the worker is done, it sends the gradients back to all the parameter servers which update the model weights.**

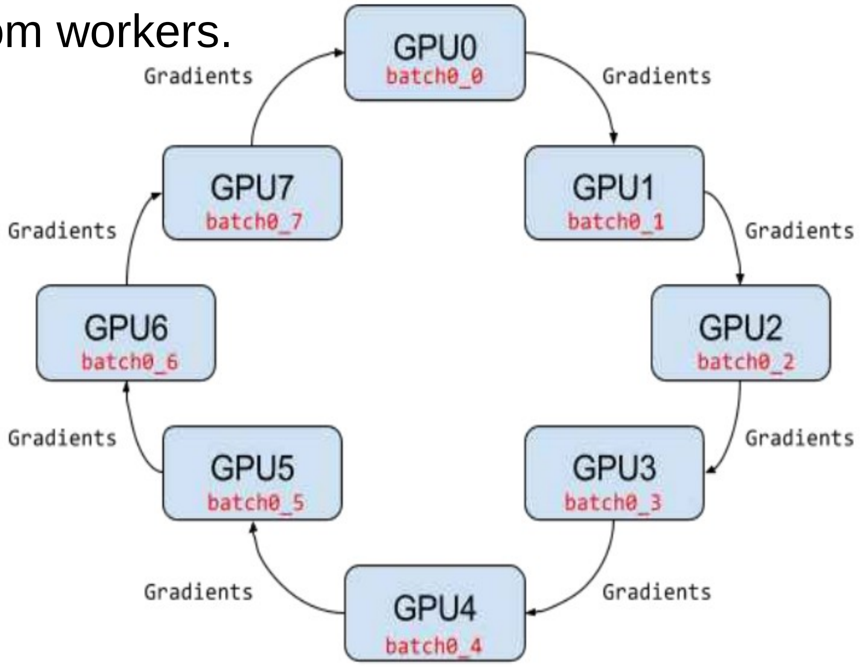


Ring-allreduce architecture

No central server that aggregates gradients from workers.

Each worker reads its own split for a mini-batch, calculates its gradients, sends its gradients to its successor neighbor on the ring, and receives gradients from its predecessor neighbor on the ring.

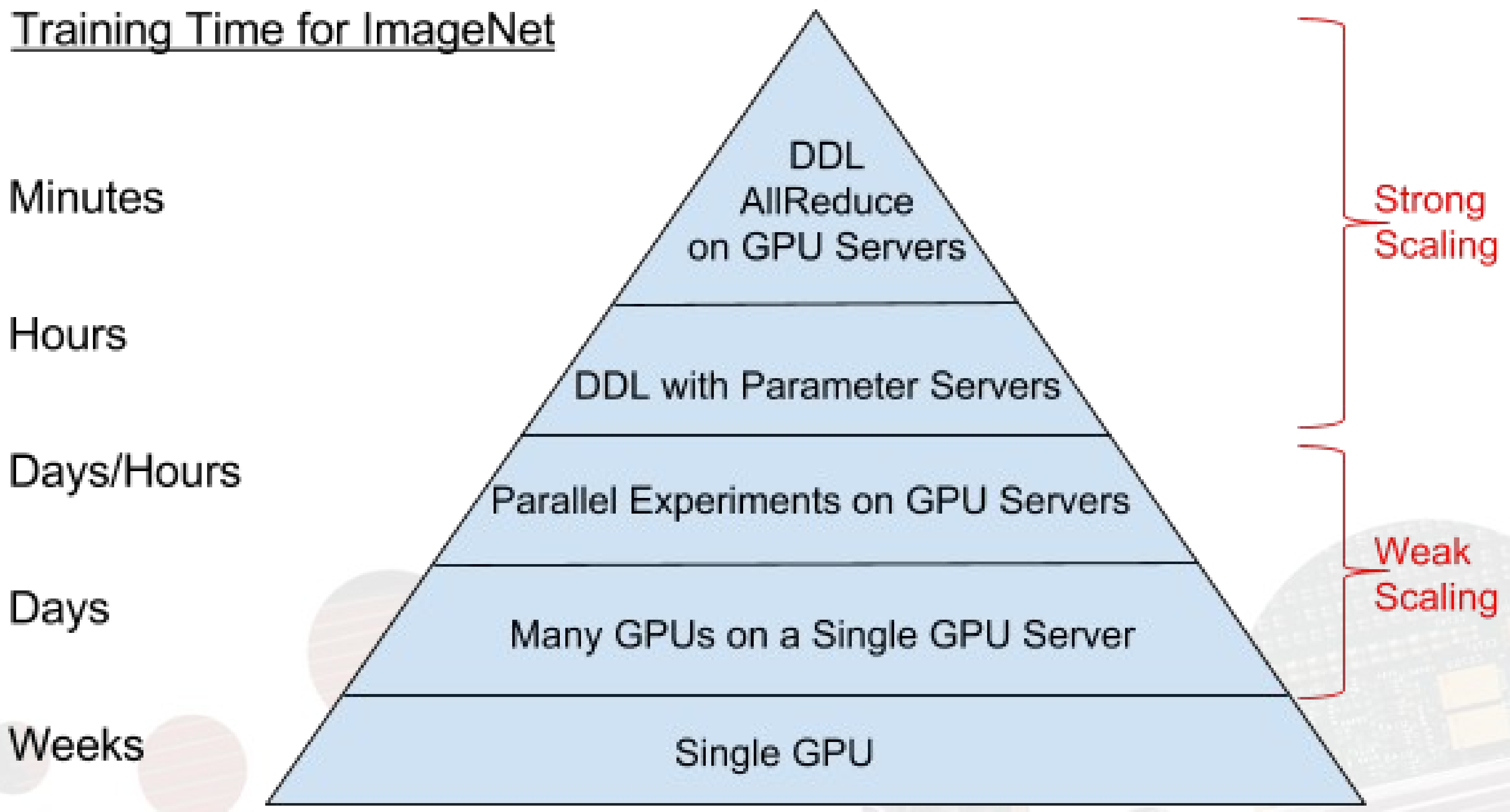
For a ring with N workers, all workers will have received the gradients necessary to calculate the updated model after N-1 gradient messages are sent and received by each worker.

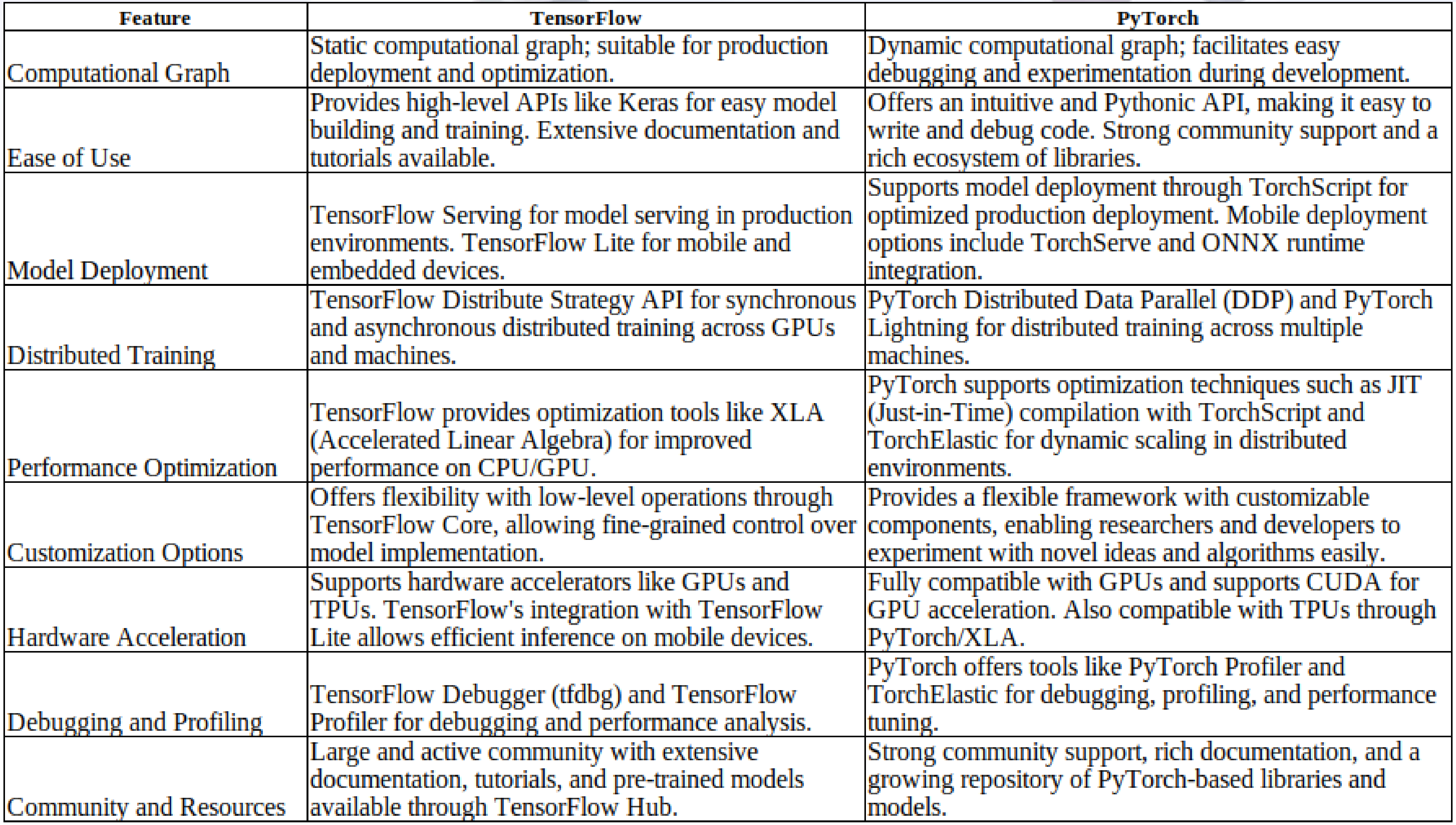


Ring-allreduce is bandwidth optimal, as it ensures that the available upload and download network bandwidth at each host is fully utilized (in contrast to the parameter server model).

Ring-allreduce can also overlap the computation of gradients at lower layers in a deep neural network with the transmission of gradients at higher layers, further reducing training time.

Training Strategies: Efficiency Over Time


Training Time for ImageNet
Minutes
Hours
Days/Hours
Days
Weeks
DDL AllReduce on GPU Servers
DDL with Parameter Servers
Parallel Experiments on GPU Servers
Many GPUs on a Single GPU Server
Single GPU
Strong Scaling
Weak Scaling



Feature	TensorFlow	PyTorch
Computational Graph	Static computational graph; suitable for production deployment and optimization.	Dynamic computational graph; facilitates easy debugging and experimentation during development.
Ease of Use	Provides high-level APIs like Keras for easy model building and training. Extensive documentation and tutorials available.	Offers an intuitive and Pythonic API, making it easy to write and debug code. Strong community support and a rich ecosystem of libraries.
Model Deployment	TensorFlow Serving for model serving in production environments. TensorFlow Lite for mobile and embedded devices.	Supports model deployment through TorchScript for optimized production deployment. Mobile deployment options include TorchServe and ONNX runtime integration.
Distributed Training	TensorFlow Distribute Strategy API for synchronous and asynchronous distributed training across GPUs and machines.	PyTorch Distributed Data Parallel (DDP) and PyTorch Lightning for distributed training across multiple machines.
Performance Optimization	TensorFlow provides optimization tools like XLA (Accelerated Linear Algebra) for improved performance on CPU/GPU.	PyTorch supports optimization techniques such as JIT (Just-in-Time) compilation with TorchScript and TorchElastic for dynamic scaling in distributed environments.
Customization Options	Offers flexibility with low-level operations through TensorFlow Core, allowing fine-grained control over model implementation.	Provides a flexible framework with customizable components, enabling researchers and developers to experiment with novel ideas and algorithms easily.
Hardware Acceleration	Supports hardware accelerators like GPUs and TPUs. TensorFlow's integration with TensorFlow Lite allows efficient inference on mobile devices.	Fully compatible with GPUs and supports CUDA for GPU acceleration. Also compatible with TPUs through PyTorch/XLA.
Debugging and Profiling	TensorFlow Debugger (tfdbg) and TensorFlow Profiler for debugging and performance analysis.	PyTorch offers tools like PyTorch Profiler and TorchElastic for debugging, profiling, and performance tuning.
Community and Resources	Large and active community with extensive documentation, tutorials, and pre-trained models available through TensorFlow Hub.	Strong community support, rich documentation, and a growing repository of PyTorch-based libraries and models.

Parallel training with TF

- **TensorFlow API to distribute training**
- Across multiple GPUs, multiple machines, or TPUs, with minimal code changes.
- **Provide good performance out of the box**.
- Synchronous training is supported via **all-reduce** and Async through **parameter server architectur**e.
- Distributed TensorFlow applications consist of a **cluster containing** one or more parameter **servers and workers.**

TF Cluster Configuration

Local Virtual Cluster:

```
tf.train.ClusterSpec({"local": ["localhost:2222", "localhost:2223"]})
```

Bare-metal Cluster:

```
tf.train.ClusterSpec({
"worker": [ "node01:2222", "node02:2222" ],
"ps": [ "master" ]})
...
if FLAGS.job_name == "ps":
server.join()
elif FLAGS.job_name == "worker":
....
```

Training Process

The training process runs for n **epochs**, where n is a user-defined hyperparameter.

Each epoch executes for several steps, and at each **step**, the model sees a **batch** of data.

Batch is used to compute gradients and update the model’s **parameters** (or **weights**).

Finally, a training **epoch** is complete when the model goes through every example in the dataset once. So, the `train_step` function in the code snippet above defines a training **step** that will run for n **epochs**.

Epoch and Step-Size

A 10 images dataset the training epoch accesses all the images.

Batch size defines the loading mechanism for GPU memory.

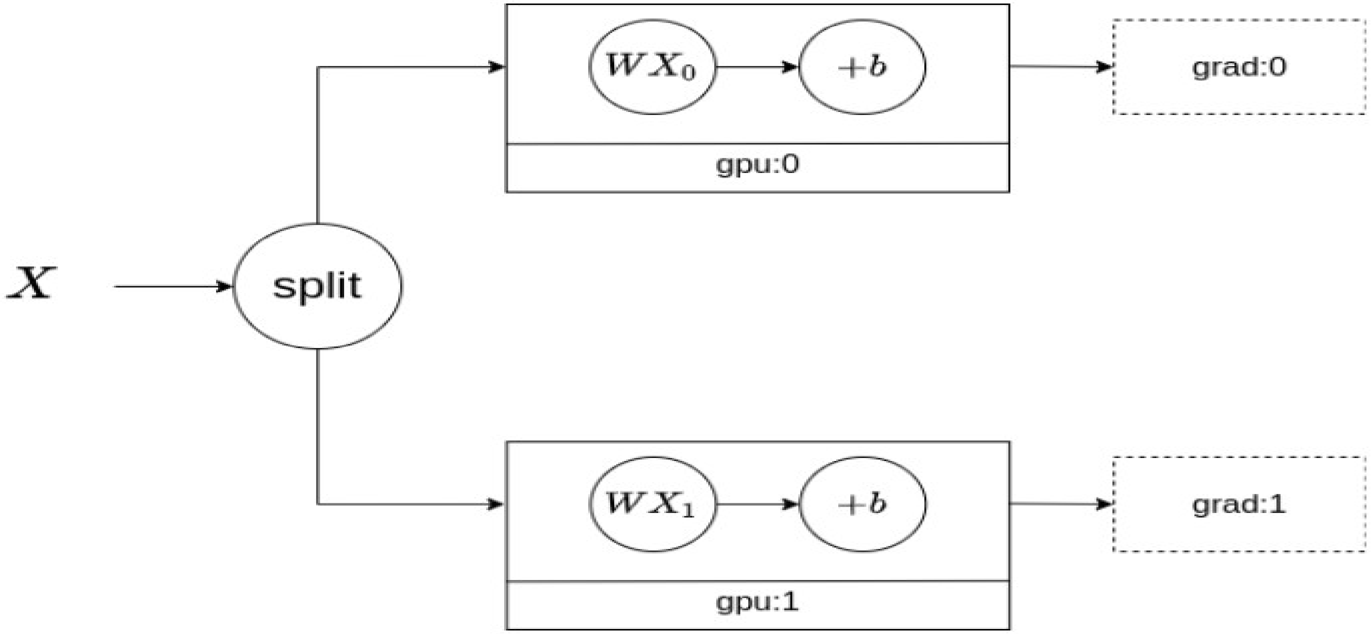
Batch size of 2 will take 5 steps to complete a single epoch.

The Batch Size images must fit in a single GPU memory and the model weights and their gradients must also fit there.

Model.fit

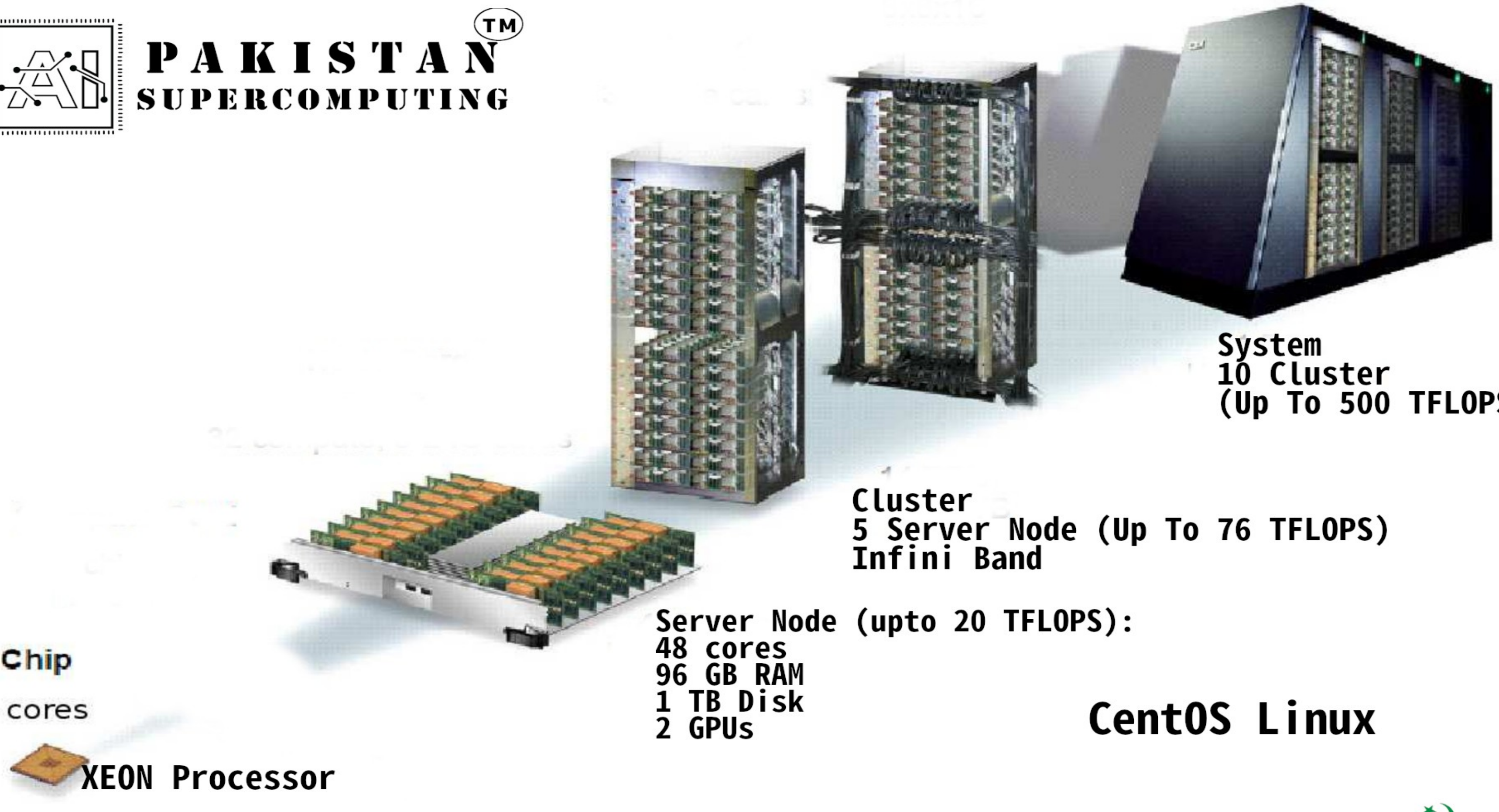
While calling model.fit(...) TF automatically occupy and use one GPU device.

TF “MirroredStrategy”, distributes the training process over multiple GPU devices on the same machine.



```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import modelsimport numpy as np
import argparse
import time
import syssys.path.append('/gpfs/projects/nct00/nct00002/cifar-utils')
from cifar import load_cifarparser = argparse.ArgumentParser()
parser.add_argument(' -- epochs', type=int, default=5)
parser.add_argument(' -- batch_size', type=int, default=2048)
parser.add_argument(' -- n_gpus', type=int, default=1)args = parser.parse_args()
batch_size = args.batch_size
epochs = args.epochs
n_gpus = args.n_gpustrain_ds, test_ds = load_cifar(batch_size)
device_type = 'GPU'
devices = tf.config.experimental.list_physical_devices(device_type)
devices_names = [d.name.split("e:")[1] for d in devices]
strategy = tf.distribute.MirroredStrategy( devices=devices_names[:n_gpus])
with strategy.scope():
model = tf.keras.applications.resnet_v2.ResNet50V2(
include_top=True, weights=None,
input_shape=(128, 128, 3), classes=10)
opt = tf.keras.optimizers.SGD(0.01*n_gpus)
model.compile(loss='sparse_categorical_crossentropy',
optimizer=opt, metrics=['accuracy'])
model.fit(train_ds, epochs=epochs, verbose=2)
```

Intro: Supercomputing System



System
10 Cluster
(Up To 500 TFLOPS)

Cluster
5 Server Node (Up To 76 TFLOPS)
Infini Band

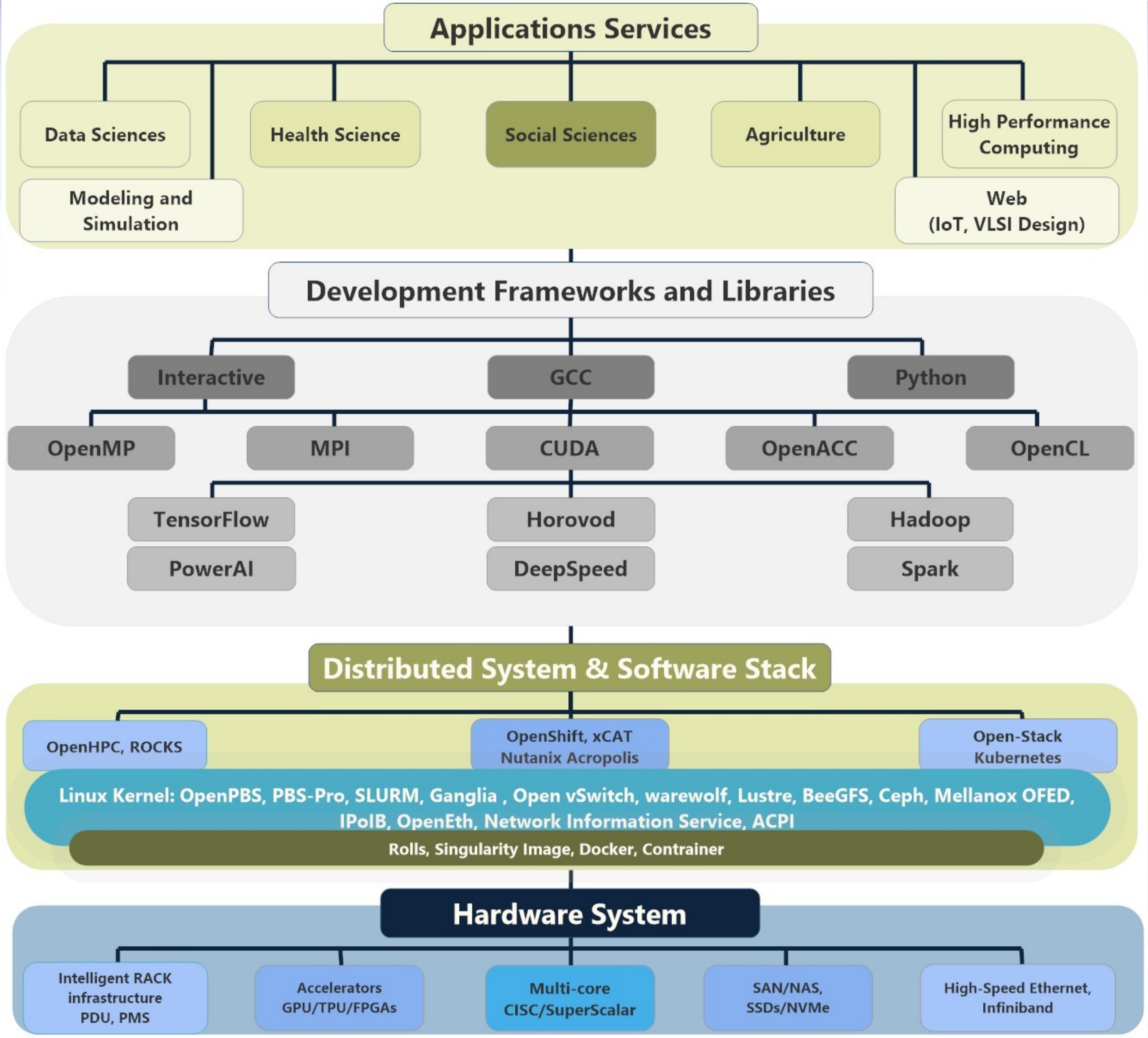
Server Node (upto 20 TFLOPS):
48 cores
96 GB RAM
1 TB Disk
2 GPUs

CentOS Linux

Chip
4 cores

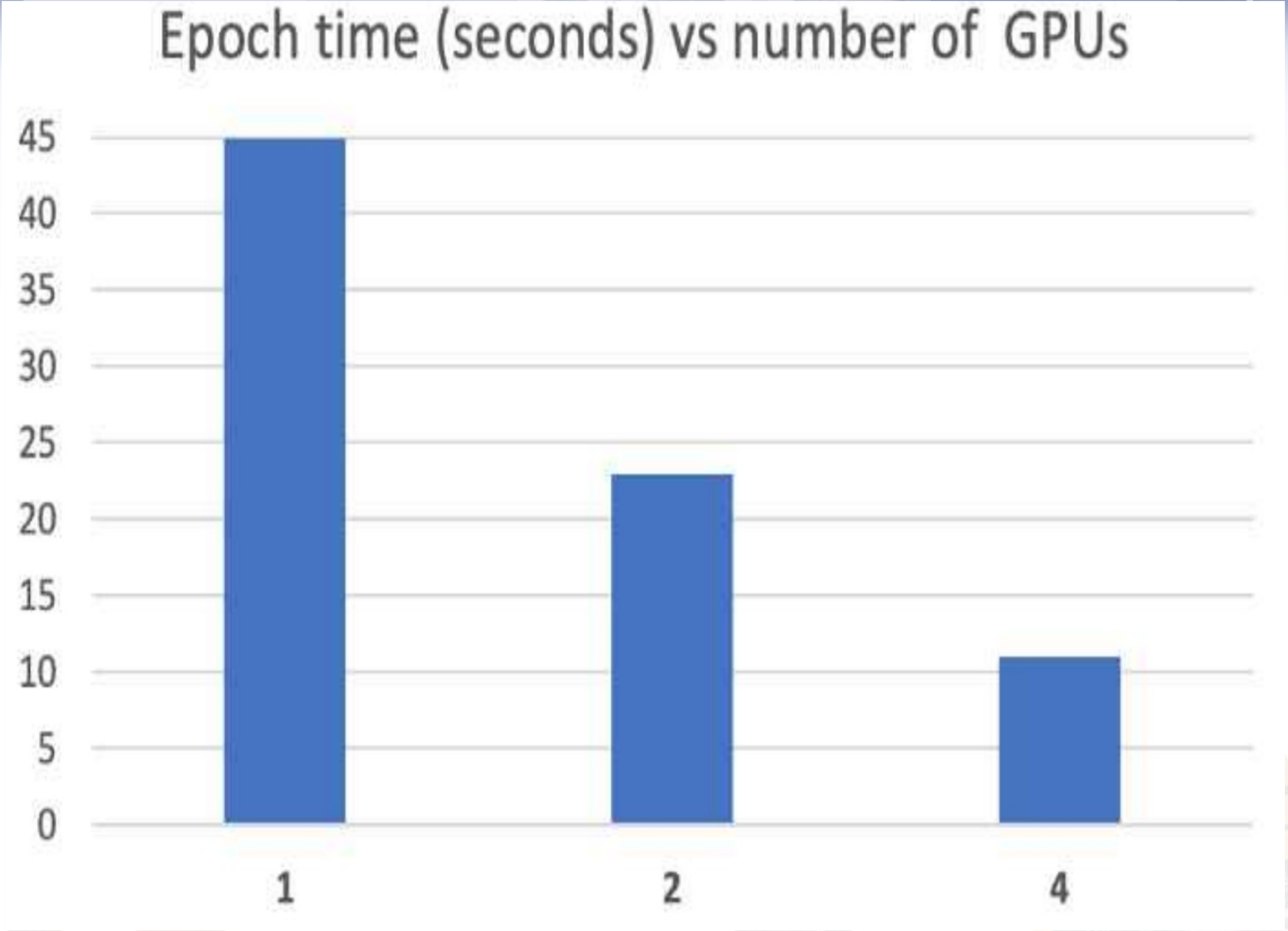
XEON Processor

Barcelona Supercomputing Center
Centro Nacional de Supercomputación


Applications Services
Data Sciences
Modeling and Simulation
Health Science
Social Sciences
Agriculture
High Performance Computing
Web (IoT, VLSI Design)
Development Frameworks and Libraries
Interactive
GCC
Python
OpenMP
MPI
CUDA
OpenACC
OpenCL
TensorFlow
PowerAI
Horovod
DeepSpeed
Hadoop
Spark
Distributed System & Software Stack
OpenHPC, ROCKS
OpenShift, xCAT Nutanix Acropolis
Open-Stack Kubernetes
Linux Kernel: OpenPBS, PBS-Pro, SLURM, Ganglia , Open vSwitch, warewolf, Lustre, BeeGFS, Ceph, Mellanox OFED, IPoIB, OpenEth, Network Information Service, ACPI
Rolls, Singularity Image, Docker, Contrainer
Hardware System
Intelligent RACK infrastructure PDU, PMS
Accelerators GPU/TPU/FPGAs
Multi-core CISC/SuperScalar
SAN/NAS, SSDs/NVMe
High-Speed Ethernet, Infiniband

Supercomputing Execution: SLURM

```
#!/bin/bash
#SBATCH --job-name=”ResNet50"
#SBATCH --D .
#SBATCH --output=ResNet50_%j.output
#SBATCH --error=ResNet50_%j.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=160
#SBATCH --gres=gpu:4
#SBATCH --time=00:20:00module purge;
module load gcc/8.3.0 cuda/10.2 cudnn/7.6.4 nccl/2.4.8 tensorrt/6.0.1
  openmpi/4.0.1 atlas/3.10.3 scalapack/2.0.2 fftw/3.3.8 szip/2.1.1 ffmpeg/4.2.1
  opencv/4.1.1 python/3.7.4_ML

python ResNet50.py -- epochs 5 -- batch_size 256 -- n_gpus 1
python ResNet50.py -- epochs 5 -- batch_size 512 -- n_gpus 2
python ResNet50.py -- epochs 5 -- batch_size 1024 -- n_gpus 4
```


Epoch time (seconds) vs number of GPUs
45
40
35
30
25
20
15
10
5
0
1
2
4

Conclusion

Complex and data-intensive AI applications are penetrating society, much like electricity. Supercomputing and parallel processing are the only solutions to support them.
With the synergy of the following roles:

- **Scientists to identify problems**
- **AI scientists**
- **HPC engineers**
- **Software developers**
- **Business experts.**

Supercomputing and Parallel Programming

Spring School Namal University Mianwali (21-23 April 2024)

- Understand Supercomputing Architecture and Programming Models
- Parallel Programming Approaches
- Multi-Core Programming
- Multi-Node Programming
- Offload Programming
- Distributed AI

Center of Excellence: Supercomputing for AI & Big Data

Distributed Artificial Intelligence

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center