SA-MIRI Course Schedule (29/10 Tentative)

day	Wednesday	day	Friday
29/10	10. Introduction to Parallel Training of Neural Networks Presentation Pc Presentation Pd Presentation Pe	31/10	Midterm period (no class)
05/11	~~(Midterm period) PRESENTATIONS – I~~ Midterm period (no class)	07/11	Pg
12/11	11. Practical Guide to Efficient Training with PyTorch 12. Parallelizing Model Training with Distributed Data Parallel Presentation Pf	14/11	Ph
19/11	Ph Presentation Pg	21/11	Pi
26/11	Honoris Causa Dr. Oriol Vinyals (no class)	28/11	Pi
03/12	13. Introduction to Large Language Models 14. End-to-End Large Language Models Workflow Presentation Ph	05/12	Pj
10/12	15. Exploring Optimization and Scaling of LLMs Presentation Pi	12/12	Pk*
17/12	Pk* 16. Looking Forward: Supercomputing and AI Futures Presentation Pj	19/12	Pl * “attendance not required (for students returning home)”

* No presentation, submission only.

SA-MIRI Course Schedule (Tentative)

Practical Session	acronim	Book Chapter included
Pa	GETTING STARTED	task 2.1 – Log into MareNostrum 5 task 2.2 – Change your password task 2.3 – (Optional) Enable passwordless ssh authentication task 2.4 – Transfer files using scp task 2.5 – (Optional) Mount the MN5 filesystem on your laptop task 3.1 – Compare icx and gcc compiler optimizations task 3.2 – Reflecting on slurm job prioritization task 3.3 – Submit your first slurm job
Pb	CONTAINERS	task 3.4 – Install docker in your platform task 3.5 – Download docker image task 3.6 – Run docker image task 3.7 – Stop a docker container task 3.8 – Run docker with port mapping task 3.9 – Start the jupyter notebook server task 3.10 – Create and run a test notebook
Pc	MPI	task 4.1 – Compile and run your first mpi program task 4.2 – Observe node distribution using hostnames task 4.3 – Point-to-point communication task 4.4 – Write and run the sequential program that estimate π task 4.5 – Write and run the parallel mpi code to estimate the value of π task 4.6 – Analysis using Gustafson's law to estimate the value of π task 4.7 – Experimenting with scatter and gather
Pd	CUDA	task 5.1 – Your first hello world in cuda task 5.2 – Dimensionality of a thread block and grid task 5.3 – Investigating parallel execution with multiple threads task 5.4 – Element-wise vector addition using cuda task 5.5 – Parallel matrix multiplication with cuda task 5.6 – Running cuda jobs with slurm task 5.7 – Profiling matrix multiplication on the gpu task 5.8 – Compute-bound vs memory-bound
Pe	CUDA-aware MPI	task 6.1 – Reflecting on cuda's execution model task 6.2 – Precision trade-offs: true or false? task 6.3 – Submit and validate the first performance run task 6.4 – Understand the metrics collection task 6.5 – Explore the effect of optimized compilation flags task 6.6 – Evaluate the impact of the cub library task 6.7 – Benchmarking the impact of gpu count and problem size
Pf	DL COLABS	task 7.1 – Set up your google colab environment task 7.2 – Execute the provided notebook step-by-step task 7.3 – Improve the accuracy of your model task 8.1 – Improving a basic cnn model task 8.2 – Exporting the python script task 8.3 – Running your first natural network on a login node task 8.4 – Submitting your first gpu dl training with slurm task 9.1 – Comparative implementation in pytorch and tensorflow

Practical Session	acronim	Book Chapter included
Pg	SCALING TF	task 10.1 – Task setup and file structure task 10.2 – Code review and understanding task 10.3 – Training on cpu (baseline performance) task 10.4 – Gpu execution time and cpu vs gpu comparison task 10.6 – Analyze the impact of gpu parallelism task 10.7 – Parallelization of resnet152 task 10.8 – Comparing parallel training performance across model sizes task 10.9 – Resnet101v2 scalability analysis task 10.10 – Interpreting results with Amdahl's and Gustafson's laws task 10.11 – Applying Amdahl's and Gustafson's laws to Resnet101v2
Ph	OPTIMIZE PT	task 11.1 – Find the maximum viable batch size task 11.2 – Investigating dataloader bottleneck with a lightweight model task 11.3 – Optimizing dataloaders with appropriate num_workers task 11.4 – Confirming dataloader efficiency with vit task 11.5 – Enable mixed precision with vit + micro-224 task 11.6 – Reproducing the effect of torch.compile() task 11.7 – Report your conclusions task 11.8 – Minor code tweaks for a final throughput boost
Pi	SCALING PT	task 12.1 – Reproducing distributed training results on mn5 task 12.2 – Analyze and compare scaling efficiency task 12.3 – Investigate diminishing returns in training time task 12.4 – Find the sweet spot for your use case
Pj	END-TO-END HF	task 14.1 – Obtain your hugging face access token task 14.2 – Download and run the hello world in google colab task 14.3 – Download the model and dataset locally using huggingface-cli task 14.4 – Transfer the model and dataset to MareNostrum 5 task 14.5 – Run the inference and fine-tuning script on mn5 task 14.6 – Compare execution in colab vs mn5
Pk	OPT & SCAL LLM	task 15.1 – Baseline experiment using facebook/opt-1.3b task 15.2 – Finding the out-of-memory limit task 15.3 – Mixed precision training task 15.4 – Model precision task 15.5 – Increasing batch size with model precision task 15.6 – Enabling flash attention task 15.7 – Increasing batch size with flash attention task 15.8 – Using the liger kernel task 15.9 – Augmenting batch size due to liger kernels task 15.10 – Scaling on multiple gpus task 15.11 – Final reflections
Pl	FUTURE	task 16.1 – Mapping the pendulum shifts of the triad task 16.2 – Exploring the new giants of compute task 16.3 – Powering the future of ai task 16.4 – Charting the role of quantum in future supercomputing task 16.5 – From data farms to ai preserves task 16.6 – Embodiment as a source of learning

(SA-MIRI V 1.0 11/25)

10. Introduction to Parallel Training of Neural Networks

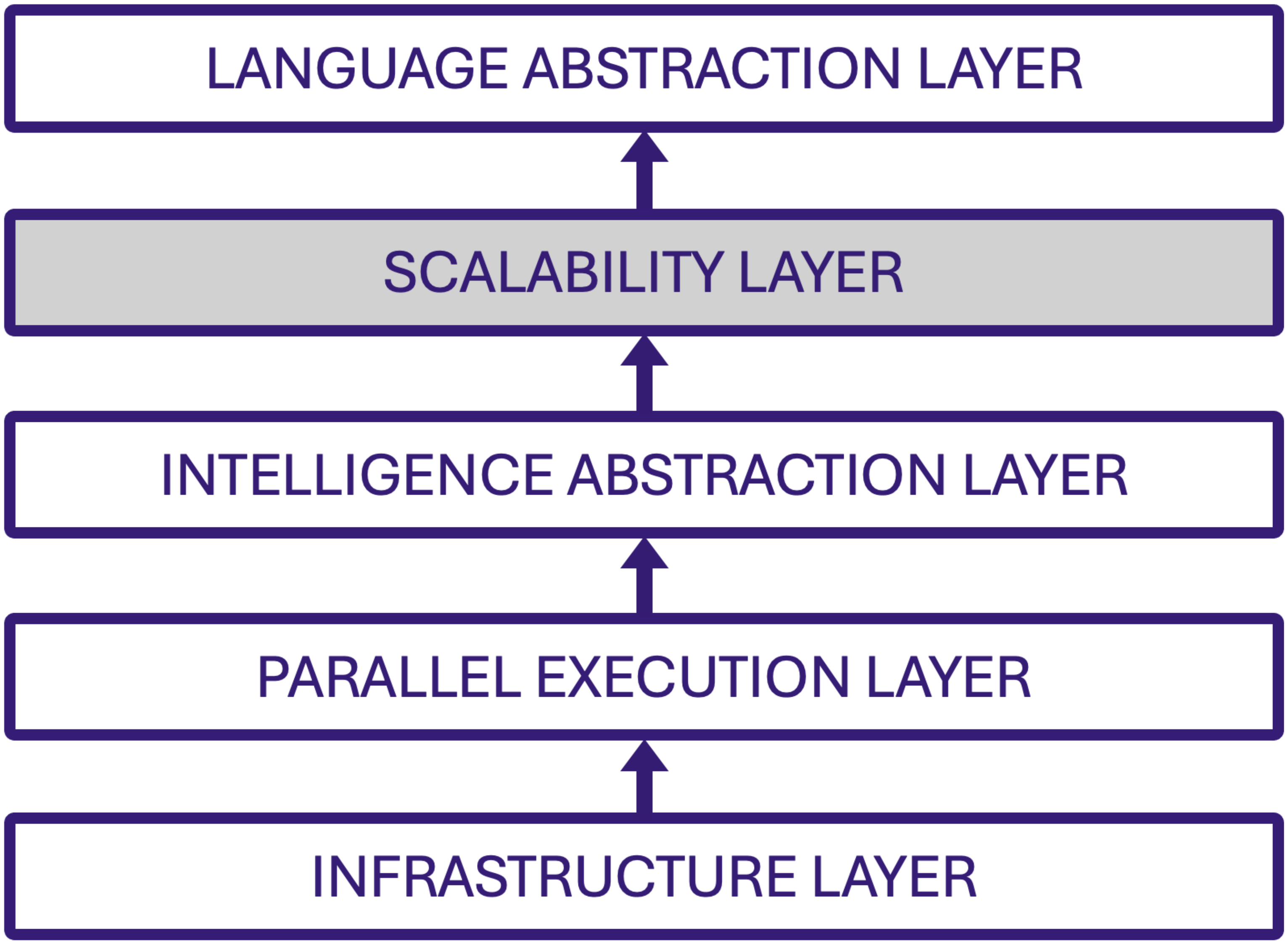
Supercomputing for Artificial Intelligence

Foundations, Architectures, and Scaling Deep Learning Workloads

Jordi TORRES.AI

Content

Part IV: The Scalability Layer



Part IV: The Scalability Layer

- **Content:**
  - This part focuses on the practical challenges and strategies for scaling neural network training efficiently across the powerful infrastructure of a modern supercomputer.
  - It represents a key transition point in the course:
    After learning single-GPU training fundamentals, students now confront the realities of scaling workloads across multiple GPUs—and eventually, multiple nodes.
  - From local optimization → to distributed scalability.

Why Parallel Training?

- **Modern models are too large/slow for a single GPU.**

- **On this course we focus:**
  - frameworks that make distributed training practical on data parallelism across multiple GPUs

User Abstraction Layer
PyTorch, TensorFlow

Software Layer
NCCL, CUDA-aware MPI

Hardware Layer
NVLINK, PCIe, Infiniband (RDMA) GPUDirect

- **Key idea: Scaling = compute + communication.**

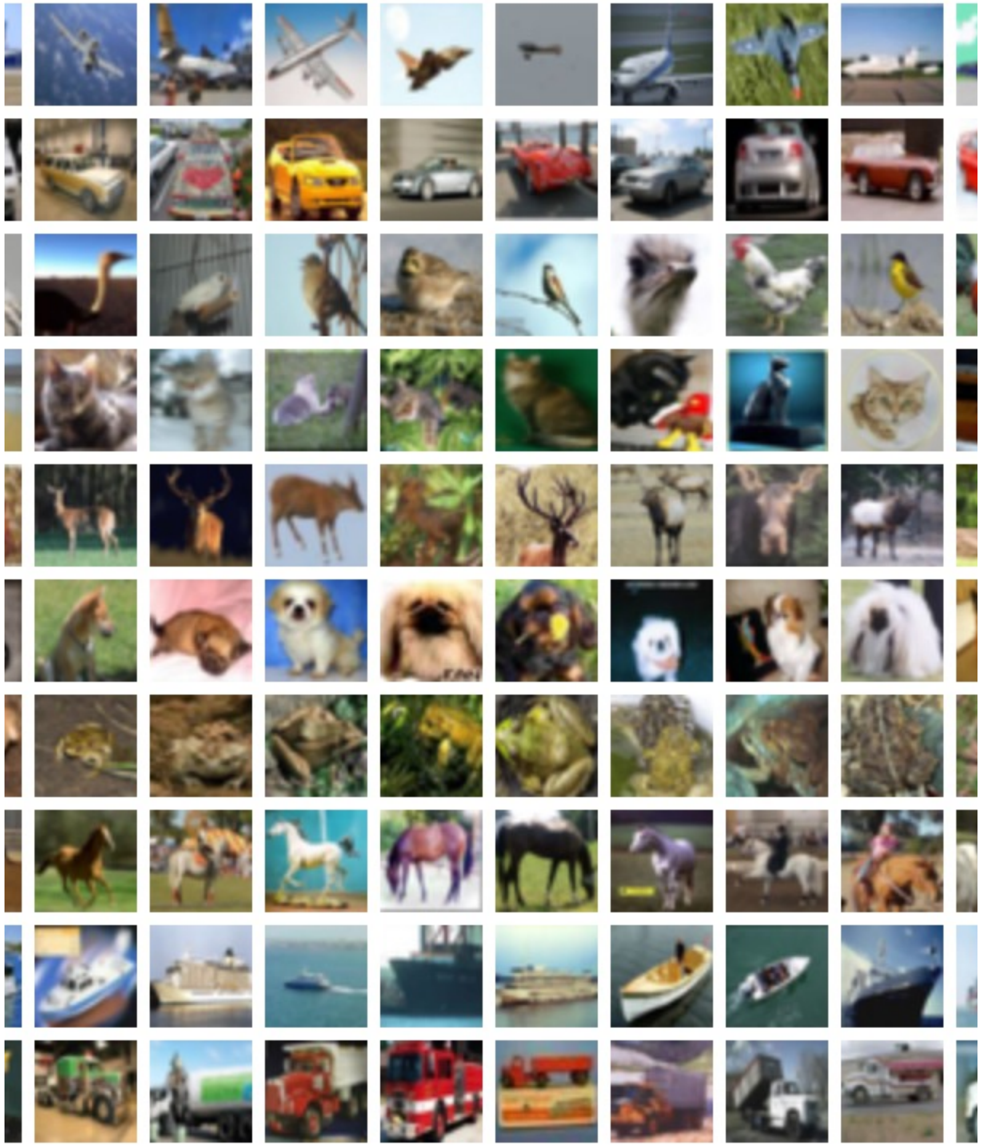
Comparing CPU and GPU Performance: A Practical Training Case

Goal of This Section

- **What we'll do**
  - Train ResNet50V2 on CIFAR-10 with TensorFlow.
  - Measure CPU vs GPU time per epoch.
  - Set a baseline for later multi-GPU runs on MN5.
- **Why TensorFlow here?**
  - Pedagogical warm-up; concepts are framework-agnostic (we'll use PyTorch later).

CIFAR-10 Dataset

- 60,000 color images (32 × 32 pixels), 10 classes.
- Each class contains 6,000 images.
- Training set: 50,000 images (5 batches × 10,000).
- Test set: 10,000 images (1,000 per class)



Source: A. Krizhevsky, V. Nair, and G. Hinton (2009), *Learning Multiple Layers of Features from Tiny Images.*

Dataset Details and Preprocessing

- **Directory structure**

```
├── cifar-10-batches-py
    ├── batches.meta
    ├── data_batch_1
    ├── data_batch_2
    ├── data_batch_3
    ├── data_batch_4
    ├── data_batch_5
    └── test_batch
```

- **Resizing for heavier workload**
  - All images resized from 32 × 32 → 128 × 128.
  - This increases computational cost to highlight CPU vs GPU performance differences.
  - May slightly reduce accuracy (acceptable here).
- **Data loading**
  - Custom function `load_data()` (in `cifar-utils/cifar.py`) Handles loading, resizing, and formatting into TensorFlow datasets.

Model Architecture: ResNet

- **Why ResNet?**
  - Residual Networks are robust for image recognition tasks.
- **Model used**

```
tf.keras.applications.ResNet50V2(
    include_top=True,
    weights="imagenet",
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
)
```

  - Parameter highlights
    - include_top=True: include classification head
    - weights=None: random initialization (no pretraining)

      Since the goal is performance comparison, not accuracy, pretrained weights are disabled.
    - input_shape=(128,128,3)
    - classes=10: CIFAR-10 categories
    - classifier_activation='softmax'

File Setup on MareNostrum 5

- **Directory to copy from GitHub (*)**

```
/Chapter10
├── ResNet50_seq.py
├── ResNet50_seq.CPU.slurm
├── ResNet50_seq.GPU.slurm
├── ResNet50.py
├── ResNet50.slurm
└── cifar-utils/
    └── cifar.py
```

(*)

```
cd <dir>
git clone https://github.com/jorditorresBCN/HPC4AIbook.git
cd HPC4AIbook
ls
```

File Setup on MareNostrum 5

- **Dataset location**
  - CIFAR-10 is pre-loaded in a shared MN5 path — do not copy it.
  - `gpfs/projects/nct_345/cifar-utils/cifar-10-batches-py`

- **Modify `cifar.py`**

```
def load_cifar(batch_size,path='/gpfs/projects/nct_345/cifar-utils/cifar-10-batches-py’):
    train_images = np.empty((50000, 3, 32, 32), dtype='uint8')
    train_labels = np.empty((50000,), dtype='uint8')

    for i in range(1, 6):
    fpath = os.path.join(path, 'data_batch_' + str(i))
    (train_images[(i - 1) * 10000: i * 10000, :, :, :],
    train_labels[(i - 1) * 10000: i * 10000]) = load_batch(fpath)

    fpath = os.path.join(path, 'test_batch')
    test_images, test_labels = load_batch(fpath)
```

Baseline Code: Sequential Training

- **Script: ResNet50_seq.py**
  - Core structure

```
train_ds, test_ds = load_cifar(batch_size)
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)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=opt, metrics=['accuracy'])
model.fit(train_ds, epochs=epochs, verbose=2)
```

  - Key points
    - Sequential execution (no data parallelism).
    - Large batch size (2048) for stable throughput measurement.
    - Focus on execution time, not accuracy.

Task 10.1

- **Task 10.1: Task Setup and File Structure**
  - Objective: Ensure all necessary files are correctly placed and accessible on MN5.
  - Checklist
    - Verify $HOME/Chapter10 folder structure.
    - Confirm Singularity image path and container availability.

    → **`gpfs/projects/nct_345/MN5-NGC-TensorFlow-23.03.sif`**
    - Check that `load_cifar()` runs without errors.

Task 10.2

- **Task 10.2: Code Review and Understanding**
  - Goal: Familiarize yourself with the logic of `ResNet50_seq.py`.
  - Focus points
    - How the dataset is loaded.
    - Model creation and configuration.
    - Compilation and optimizer setup.
    - Execution flow of model.fit().
  - Tip:
    - Ask your instructor for clarification if any part of the workflow is unclear.

Training on CPU (Baseline)

- **SLURM script: `ResNet50_seq.CPU.slurm`**

```
#!/bin/bash
#SBATCH --chdir .
#SBATCH --job-name=ResNet50_seq_CPU
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --cpus-per-task 20
#SBATCH --time 01:15:00
#SBATCH --account <account>
#SBATCH --qos acc_debug

module purge
module load singularity

SINGULARITY_CONTAINER=/gpfs/<path>/MN5-NGC-TensorFlow-23.03.sif
singularity exec $SINGULARITY_CONTAINER python ResNet50_seq.py --epochs 1 --batch_size 256
```

Container: `MN5-NGC-TensorFlow-23.03.sif` (from Section 3.4).

Task 10.3

- **Task 10.3: CPU Baseline Measurement**
  - Your mission
    - Review and adjust your SLURM script if necessary.
    - Execute the job and examine .out logs.
    - Record the epoch time (**focus on 2nd epoch**).
  - Expected result ≈ 215–220 seconds per epoch.

Running on a Single GPU

- **SLURM script: ResNet50_seq.GPU.slurm**

```
...
#SBATCH --gres=gpu:1
...
singularity exec --nv $SINGULARITY_CONTAINER python ResNet50_seq.py --epochs 1 --batch_size 256
```

The `--nv` flag enables GPU access within the container.

Task 10.4

- **Task 10.4: GPU Execution Time and Comparison**
  - Goal
    - Measure execution time on GPU.
    - Compare results with CPU baseline.
    - Discuss observed differences.
  - Expected outcome
    - ~10 × speedup from CPU to GPU.
    - Lower latency per step (121 ms → 73 ms).
    - Focus on the second epoch (after warm-up).

Analysis and Discussion

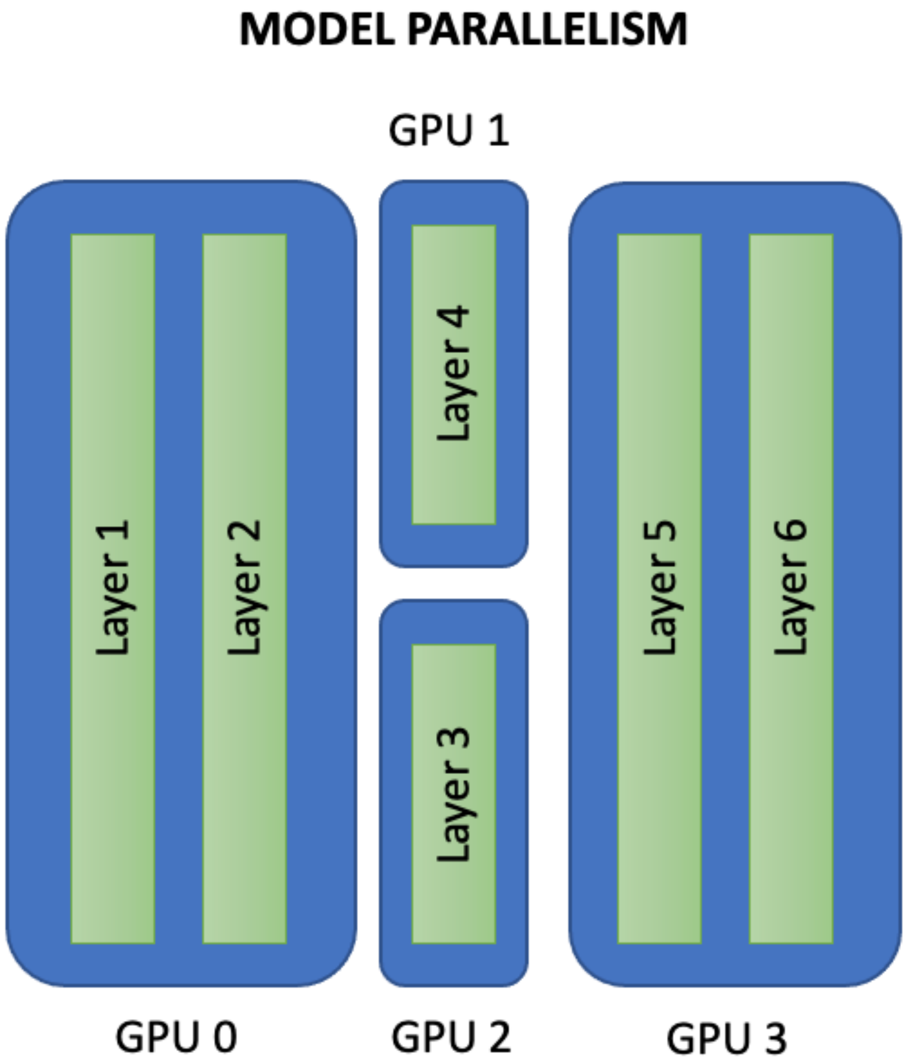
- **Training on GPU is an order of magnitude faster than on CPU.**
  - GPUs exploit massive parallelism (thousands of CUDA cores).
- **Notes**
  - CIFAR-10 provides a simple yet effective benchmark.
  - ResNet50V2 offers a realistic workload for GPU benchmarking.
  - Accuracy remains low (few epochs, no pretraining).
  - Warm-up in epoch 1 introduces extra overhead → ignore it in comparisons.
  - **This sequential setup establishes the baseline for multi-GPU training in the next tasks.**

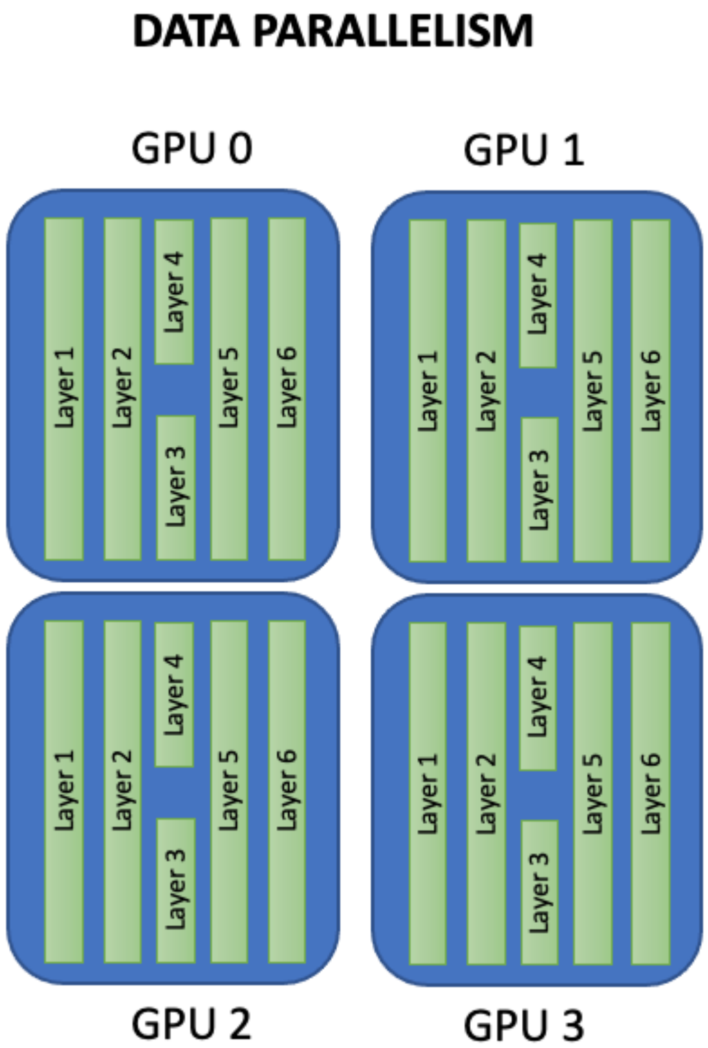
Accelerate Training with Parallelism in TensorFlow

Why Parallel Training (HPC context)

- Deep learning training is compute- and memory-intensive.
- Multi-GPU data parallelism lets us train bigger models/datasets and finish sooner.
- Key idea: split the work across GPUs while keeping model replicas in sync.

Two Families of Parallelism





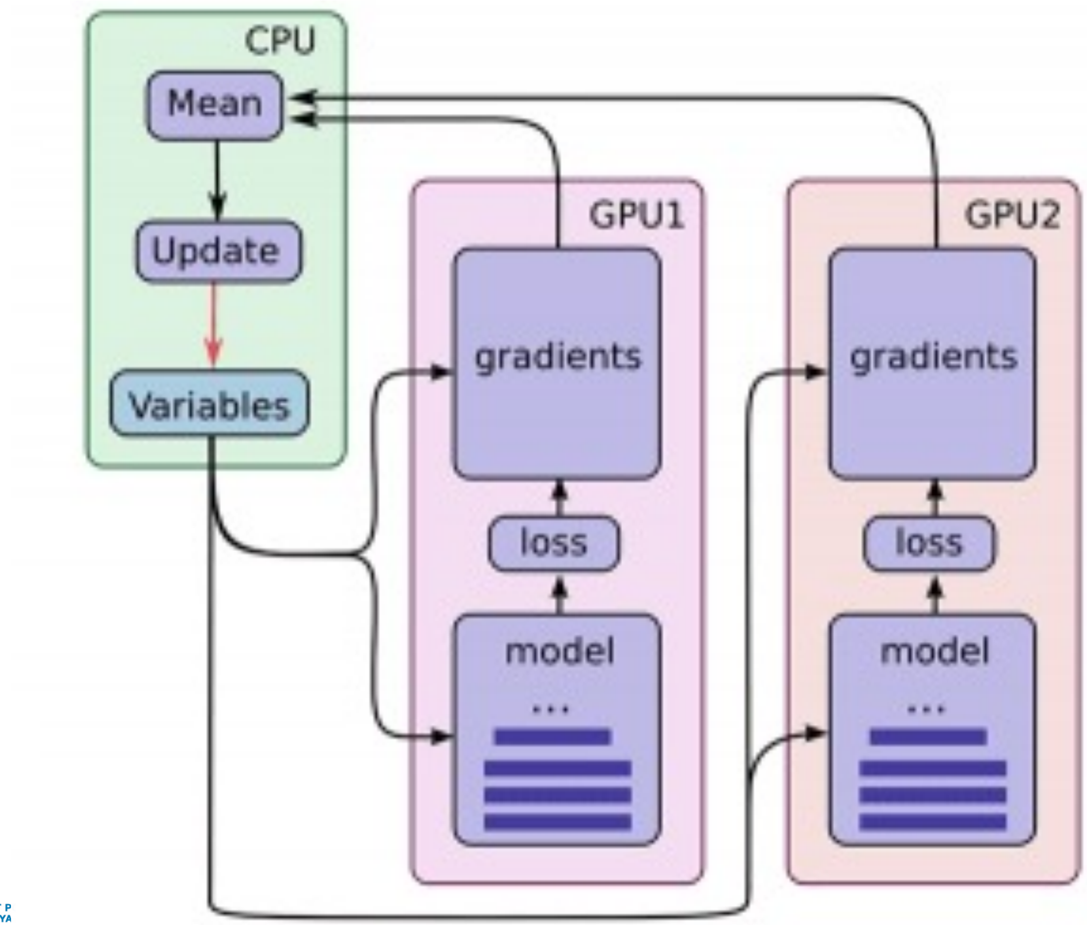
Two Families of Parallelism

- **Model parallelism:**
  – split the *model* across devices (layers/ops on different GPUs).
- **Data parallelism:**
  – replicate the model; each GPU trains on different data shards; synchronize gradients each step.
- **We focus on data parallelism**
  → simplest, most common on a single node.

- **Beyond Model & Data Parallelism (for context)**
  – Gradient Accumulation → large *effective* batches via micro-batches.
  – Optimizer State Sharding (ZeRO) → shard params/gradients/optimizer states.
  – Tensor / Pipeline / Sequence Parallelism → used by LLM stacks (DeepSpeed, Megatron-LM).

MirroredStrategy in One Picture

- **Creates one replica per GPU on a single host.**
  - Performs synchronous all-reduce of gradients each step.
  - Keeps mirrored variables consistent on all devices.



source: https://jhui.github.io/2017/03/07/TensorFlow-GPU/

MirroredStrategy API

```
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

```
with mirrored_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(learning_rate)

     model.compile(loss='sparse_categorical_crossentropy',
                   optimizer=opt, metrics=['accuracy'])
```

```
dataset = load_data(batch_size)
model.fit(dataset, epochs=5, verbose=2)
```

MirroredStrategy

- **What Actually Happens Each Step**
  - Replicate model vars (kept in sync).
  - Each GPU: forward + backward on its local mini-batch.
  - All-Reduce gradients → update shared variables → broadcast.

- **Takeaway:**
  - compute is local; sync cost grows with number of GPUs.

Batch Size & Learning Rate Rules

- **Aim for the largest batch that fits (avoid OOM).**
- **With MirroredStrategy, the script's batch_size is global:**
  - 1 GPU → 2048
  - 2 GPUs → 4096 (per-GPU = 2048)
  - 4 GPUs → 8192 (per-GPU = 2048)
- **Learning Rate scaling rule (linear):**

```
learning_rate = learning_rate_base * number_of_gpus
opt = tf.keras.optimizers.SGD(learning_rate)
```

- **Memory tips:**
  – H100 64 GB fits the values above; reduce on smaller GPUs.

Measuring Performance

- **Use epoch time (seconds) from `fit()` as a simple timing proxy.**
  - Ignore epoch 1 (initialization & warm-up).

- **Throughput metrics:**
  - **Global**: images/s across all GPUs.
  - **Per-GPU**: global / #GPUs (efficiency signal).

Experiment Template (MN5)

- **Script:**

```
ResNet50.py (args: --epochs E --batch_size B --n_gpus G)
```

- **SLURM batch** (single job, 3 runs for clarity)**:**

```
singularity exec --nv $SIF python ResNet50.py --epochs 5 --batch_size 2048 --n_gpus 1
singularity exec --nv $SIF python ResNet50.py --epochs 5 --batch_size 4096 --n_gpus 2
singularity exec --nv $SIF python ResNet50.py --epochs 5 --batch_size 8192 --n_gpus 4
```

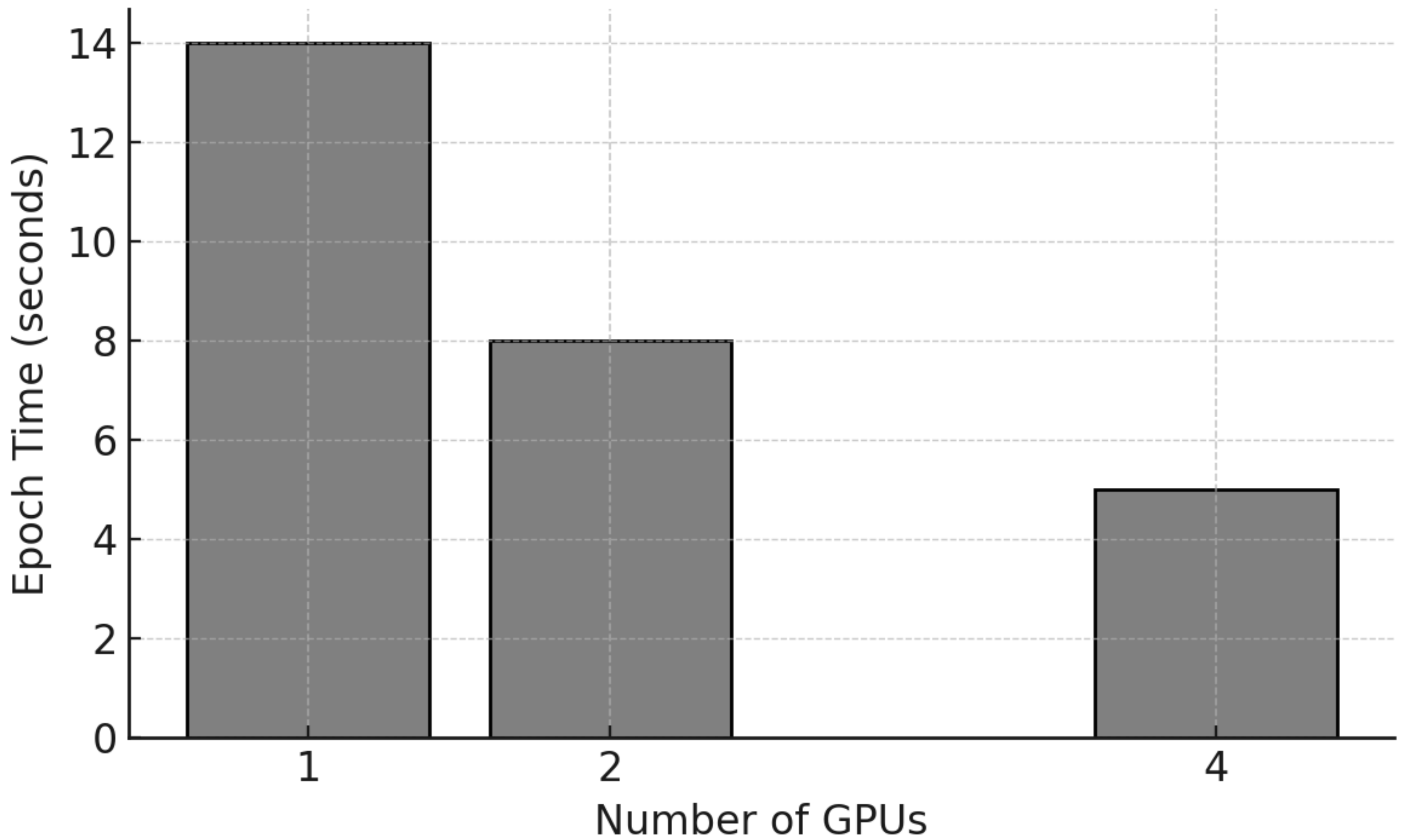
- **Important notes:**
  - In production, submit separate jobs per config to avoid idle GPUs.
  - We care about performance, not accuracy, in this warm-up.

Results: Epoch Time (ResNet50, CIFAR-10)

```
 Model ResNet50 1 GPUs
-----------------------
Batch_size: 2048
Num_replicas: 1

Epoch 1/2
25/25 - 29s - loss: 2.2484 - accuracy: 0.1629 - 29s/epoch - 1s/step
Epoch 2/2
25/25 - 14s - loss: 2.1255 - accuracy: 0.2195 - 14s/epoch - 544ms/step
 Model ResNet50 2 GPUs
-----------------------
Batch_size: 4096
Num_replicas: 2

Epoch 1/2
13/13 - 31s - loss: 2.2603 - accuracy: 0.1494 - 31s/epoch - 2s/step
Epoch 2/2
13/13 - 8s - loss: 2.1533 - accuracy: 0.2064 - 8s/epoch - 623ms/step
 Model ResNet50 4 GPUs
-----------------------
Batch_size: 8192
Num_replicas: 4

Epoch 1/2
7/7 - 47s - loss: 2.2741 - accuracy: 0.1514 - 47s/epoch - 7s/step
Epoch 2/2
7/7 - 5s - loss: 2.1481 - accuracy: 0.2160 - 5s/epoch - 768ms/step
```

Results: Epoch Time

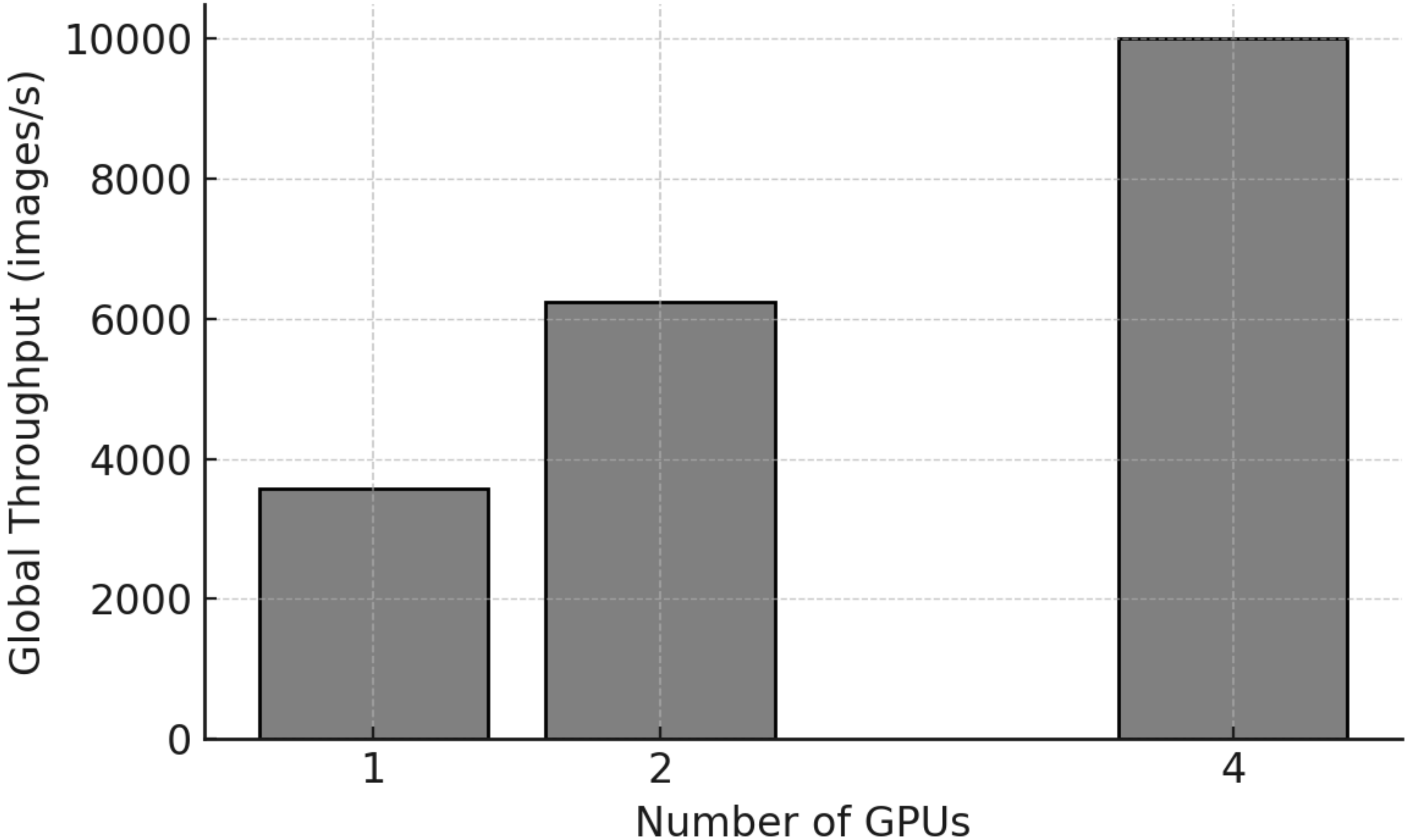
- **(ResNet50, CIFAR-10)**



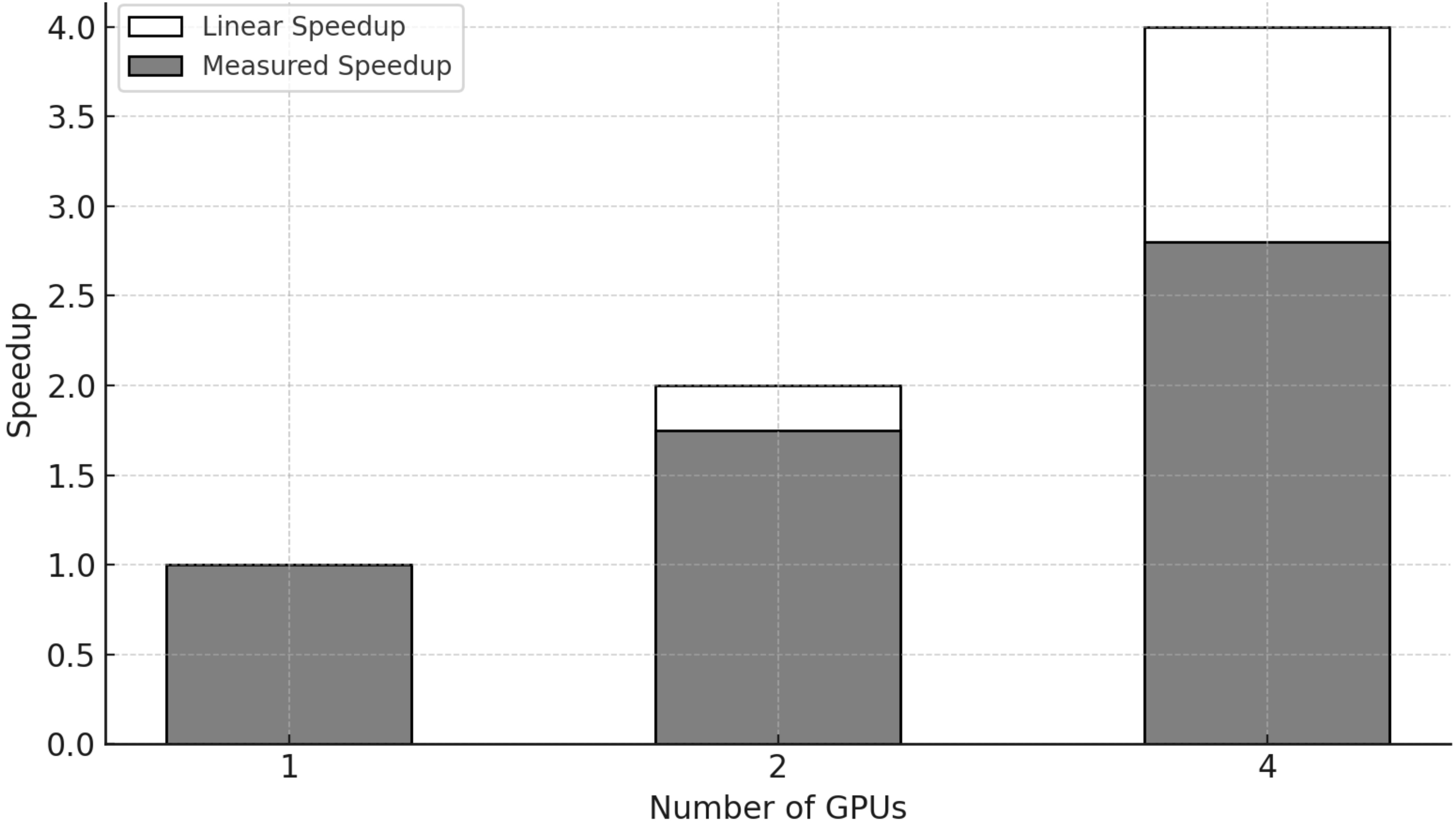
→ sub-linear improvement (good but not perfect scaling).

Global Throughput

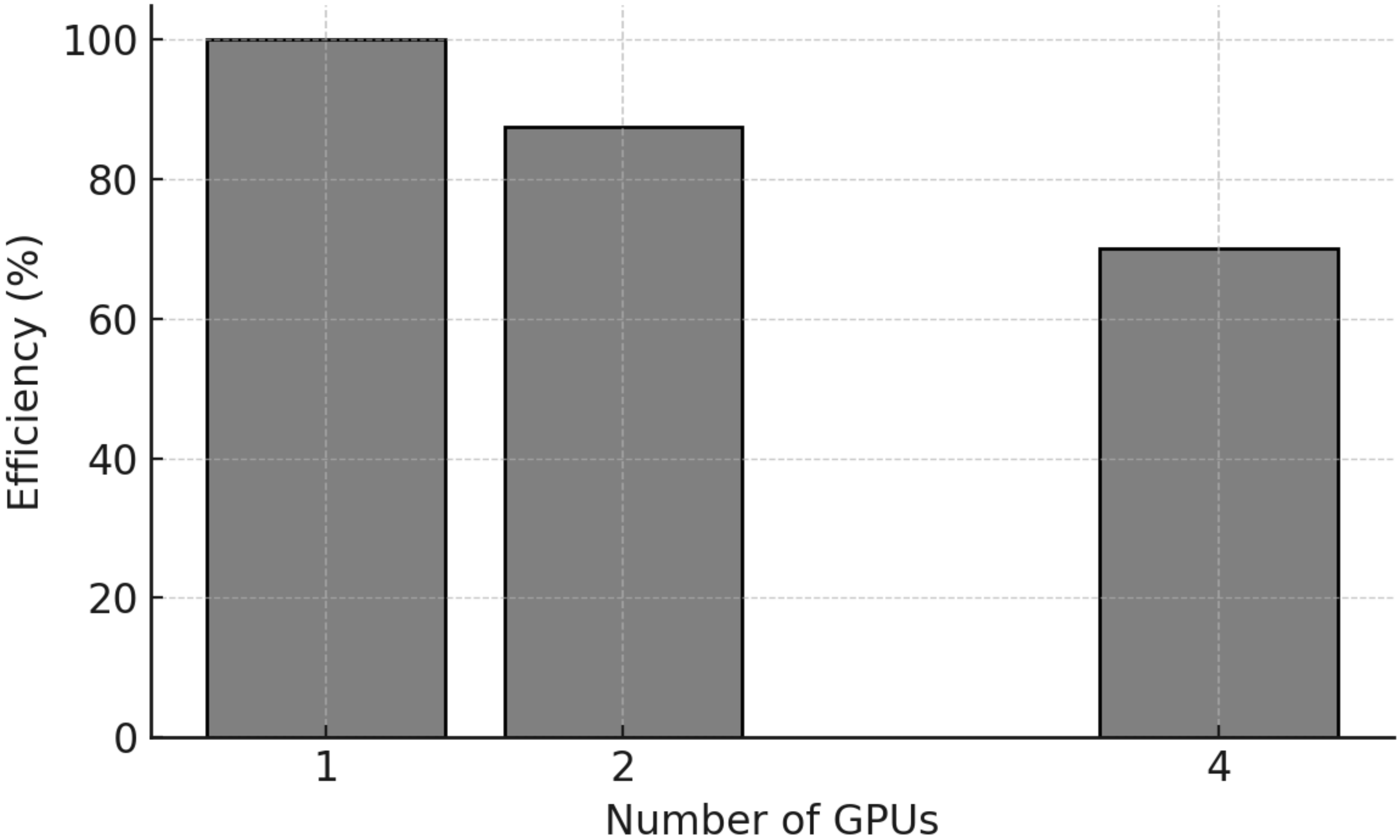
- Compute with 50,000 training images.
- More GPUs → higher images/s, but not linear.



Speedup


Linear Speedup
Measured Speedup
Speedup
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
4
Number of GPUs

Efficiency



Task 10.6

- ~~Task 10.5~~
- **Task 10.6 Analyze the Impact of GPU Parallelism**
  - Goal: reproduce the 1/2/4-GPU runs; report *epoch-2 time*, global throughput, speedup, and efficiency.
  - Deliverables
    - Plot epoch time vs GPUs.
    - Plot global throughput vs GPUs.
    - Plot speedup + ideal line; plot efficiency.
  - Short analysis:
    - How does throughput evolve with GPUs?
    - Is speedup close to linear? If not, explain using the overheads above.

Pro tip: keep batch per GPU constant; double-check LR scaling.

Impact of Model Size on Parallel Training

Why Model Size Matters

- **Deeper models**

  ⇒ more params & FLOPs ⇒ higher compute & memory needs.
- **Larger compute per GPU can hide communication**

  → often better efficiency.

- **Case study:**
  - Compare ResNet50V2 vs ResNet152V2 on CIFAR-10.
  - Runs on 1, 2, 4 GPUs (single node, MirroredStrategy).
  - Focus: performance & scaling

ResNet152V2 Setup (MN5)

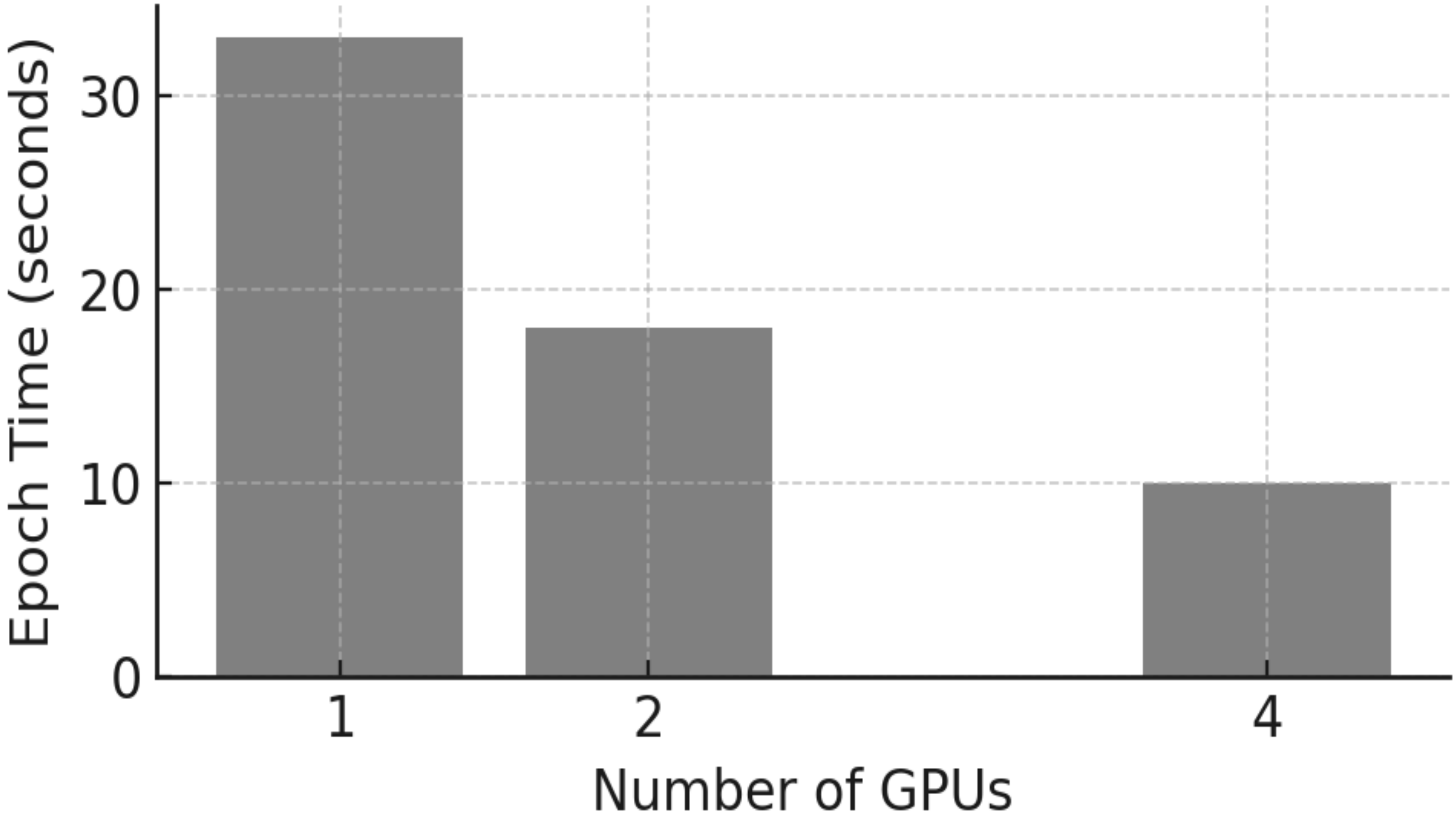
- **H100 SXM5 64 GB recommended per-GPU batches:**
  - **1 GPU:** 1024
  - **2 GPUs:** 2048
  - **4 GPUs:** 4096

  - Smaller-memory GPUs (A100 40 GB / RTX 4090 24 GB): reduce by ×2–×4.
  - Same script & SLURM template; only model and batch/LR change.

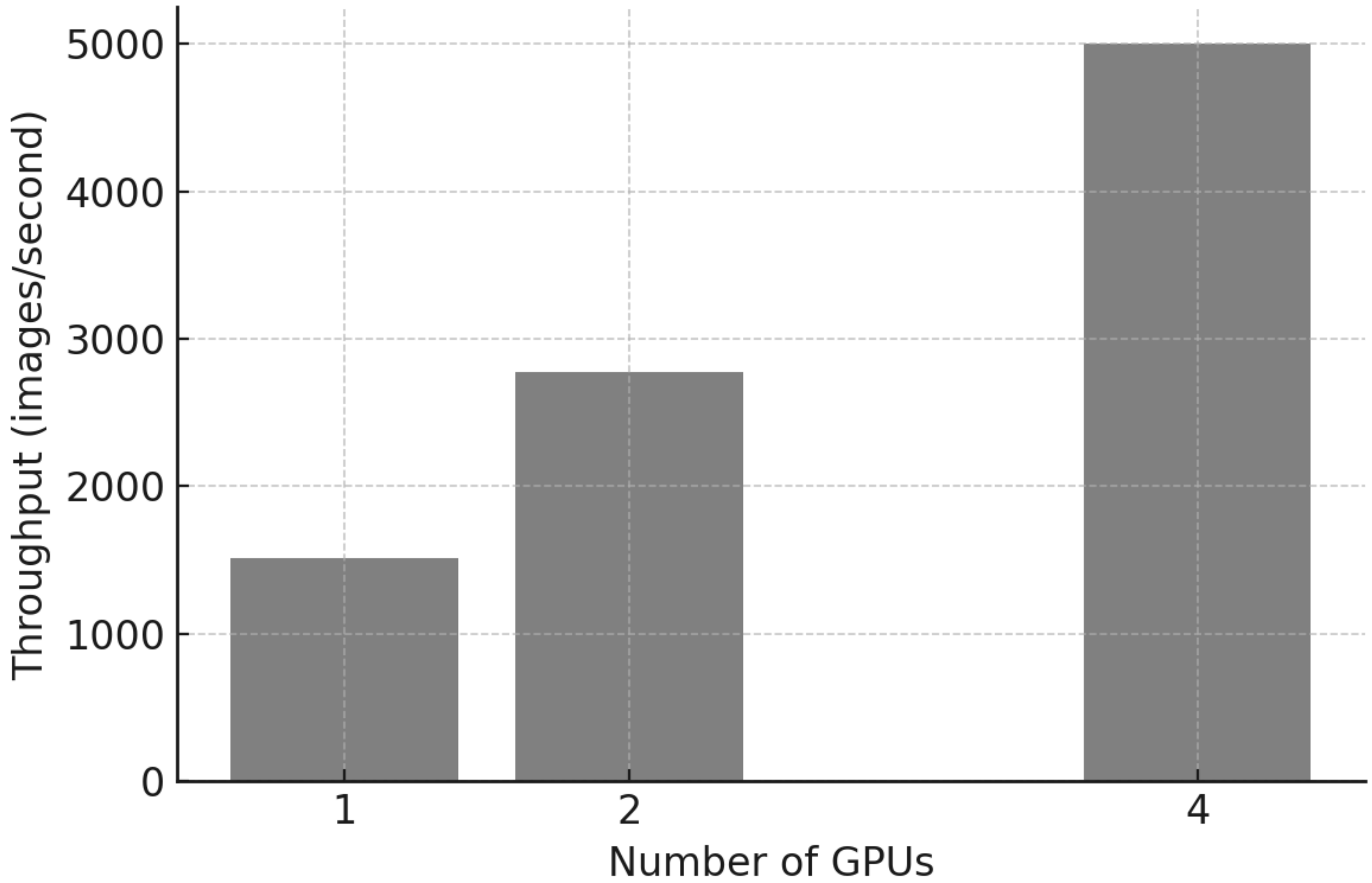
ResNet152V2: Raw Results

```
Model ResNet152 1 GPUs
-----------------------
Batch_size: 1024
Num_replicas: 1

Epoch 1/2
49/49 - 63s - loss: 2.2454 - accuracy: 0.1583 - 63s/epoch - 1s/step
Epoch 2/2
49/49 - 33s - loss: 2.1264 - accuracy: 0.2140 - 33s/epoch - 668ms/step
 Model ResNet152 2 GPUs
-----------------------
Batch_size: 2048
Num_replicas: 2

Epoch 1/2
25/25 - 76s - loss: 2.2705 - accuracy: 0.1471 - 76s/epoch - 3s/step
Epoch 2/2
25/25 - 18s - loss: 2.1383 - accuracy: 0.2072 - 18s/epoch - 718ms/step
 Model ResNet152 4 GPUs
-----------------------
Batch_size: 4096
Num_replicas: 4

Epoch 1/2
13/13 - 126s - loss: 2.3283 - accuracy: 0.1462 - 126s/epoch - 10s/step
Epoch 2/2
13/13 - 10s - loss: 2.2006 - accuracy: 0.1844 - 10s/epoch - 773ms/step
```

ResNet152V2: Results

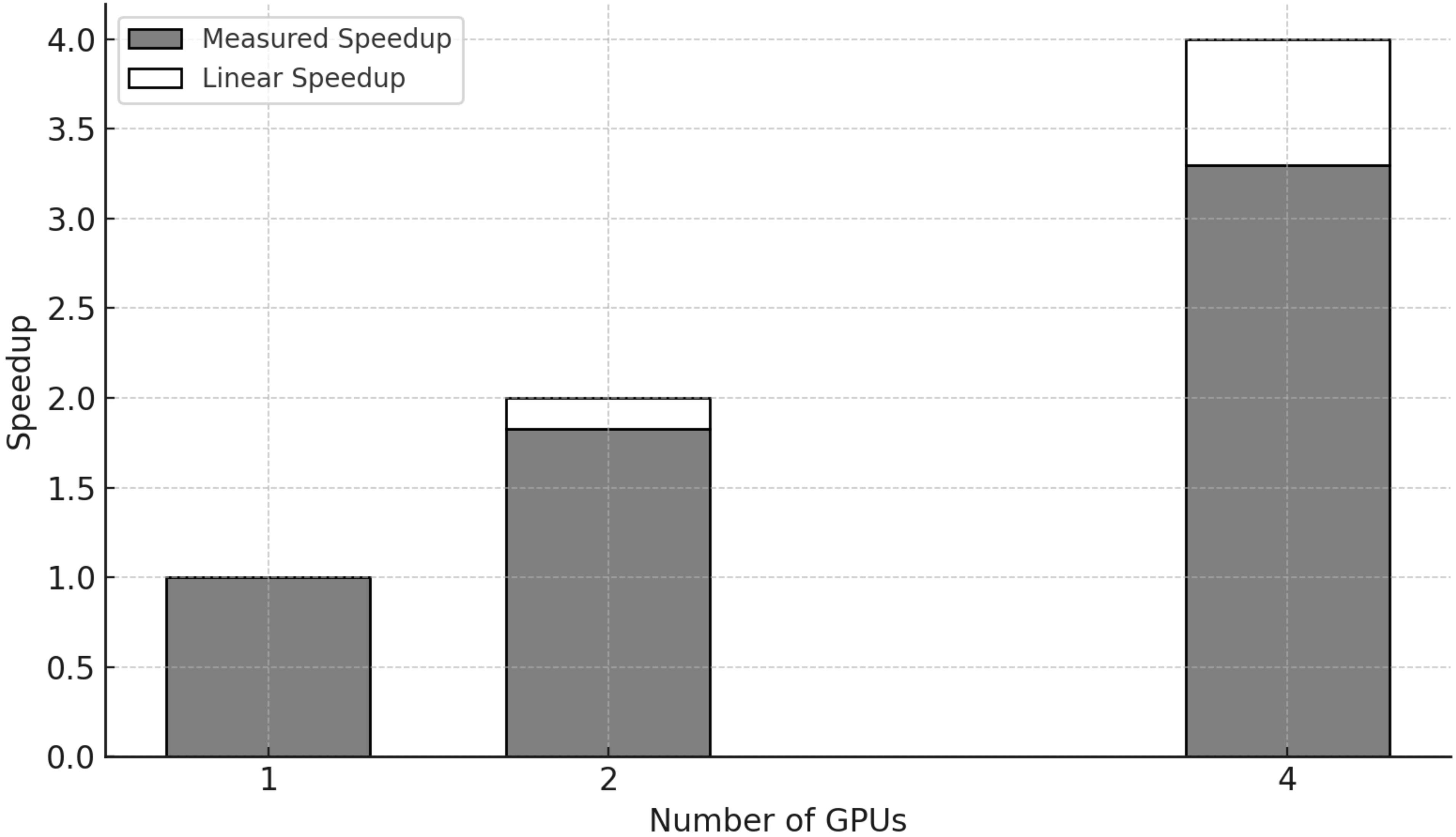


ResNet152V2: Throughput



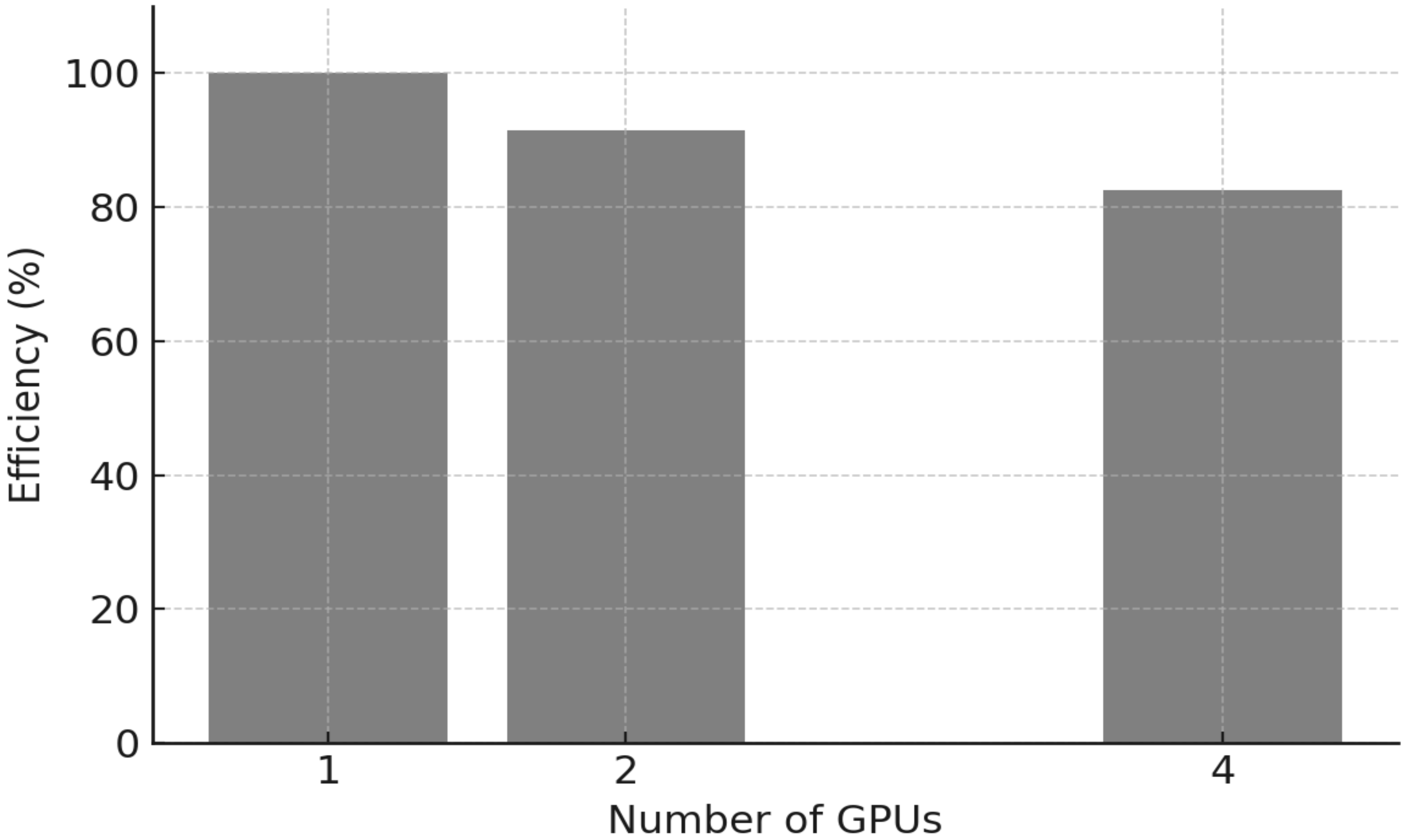
→ Global throughput (images/s) grows with #GPUs, not perfectly linear.

Speedup



Measured Speedup
Linear Speedup
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Speedup
1
2
4
Number of GPUs

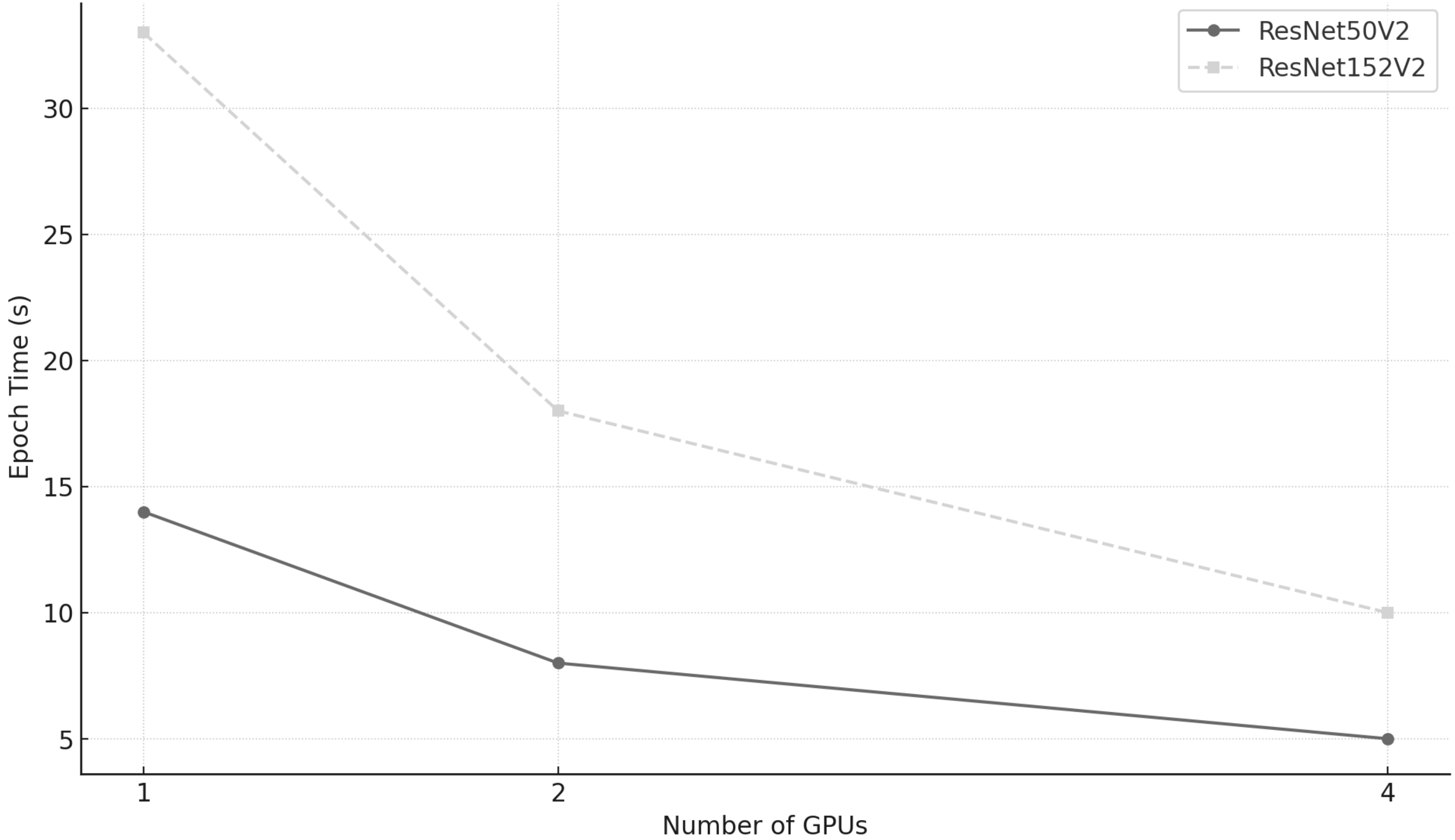
Efficiency



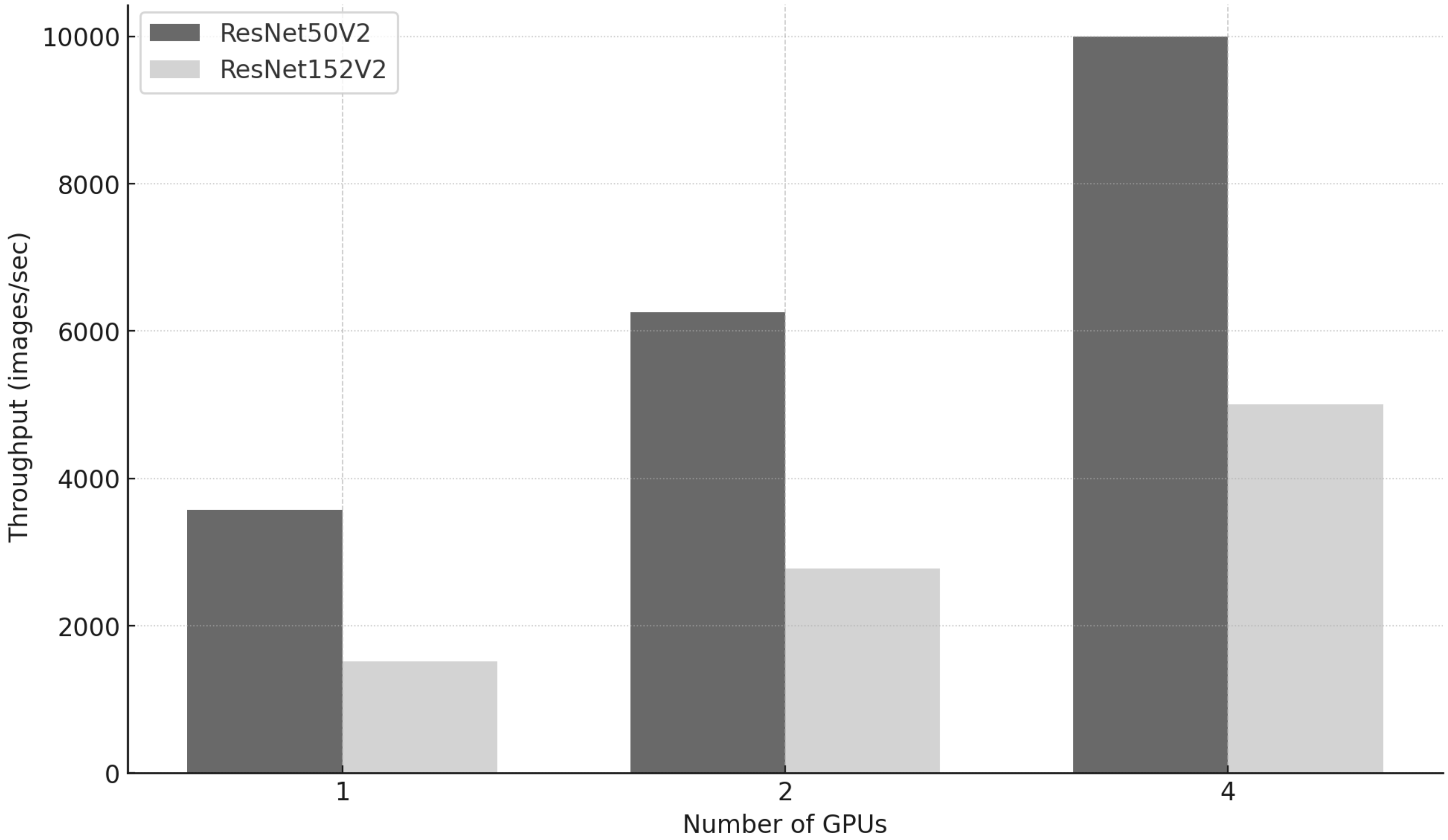
Interpreting ResNet152V2 Scaling

- Gradient all-reduce latency grows with replicas.
- Input pipeline pressure as step time shrinks.
- Fixed costs (I/O & init) more visible at small epoch times.
  → better efficiency than ResNet50 at 4 GPUs.

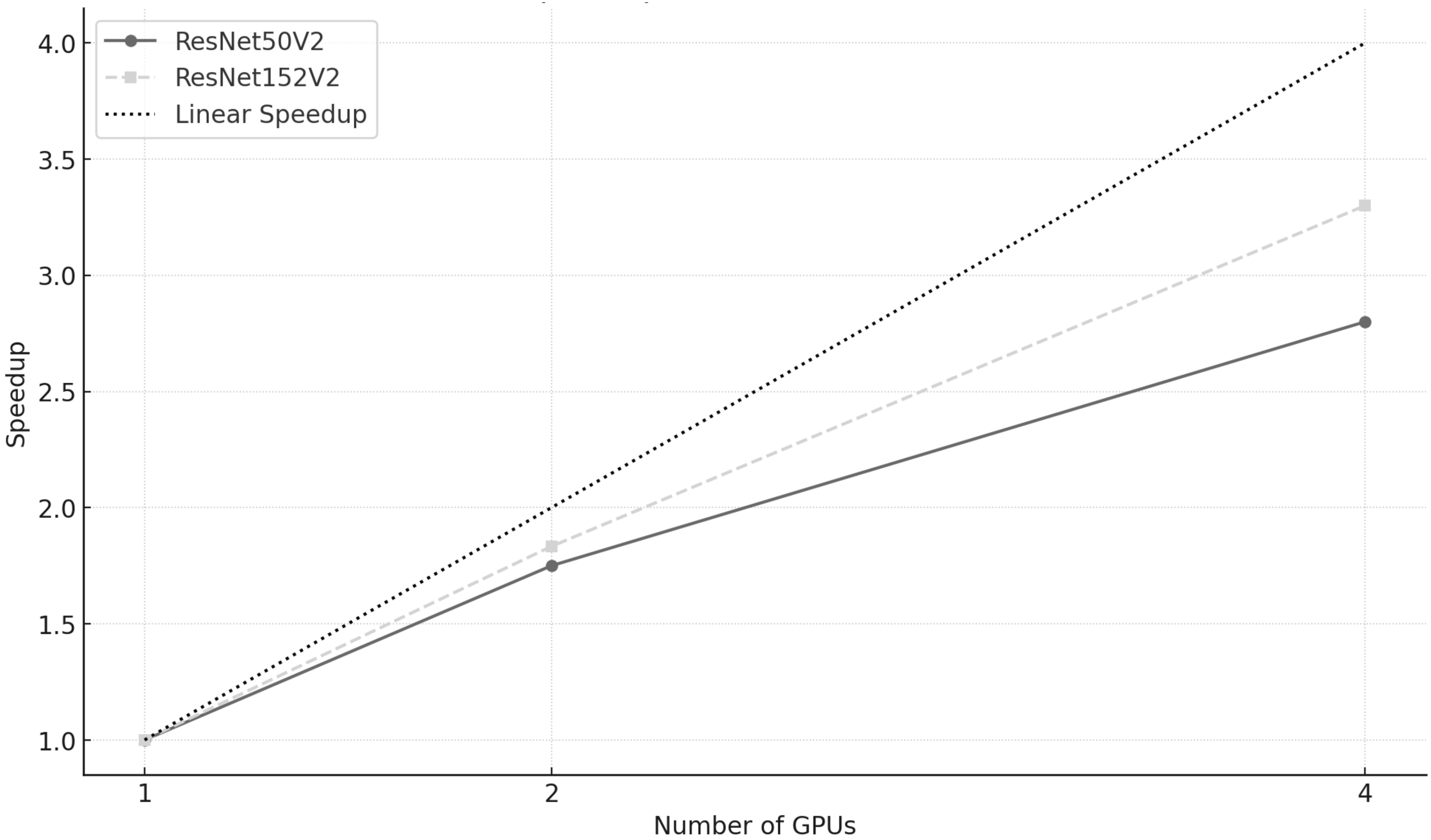
Epoch Time: ResNet50V2 vs ResNet152V2



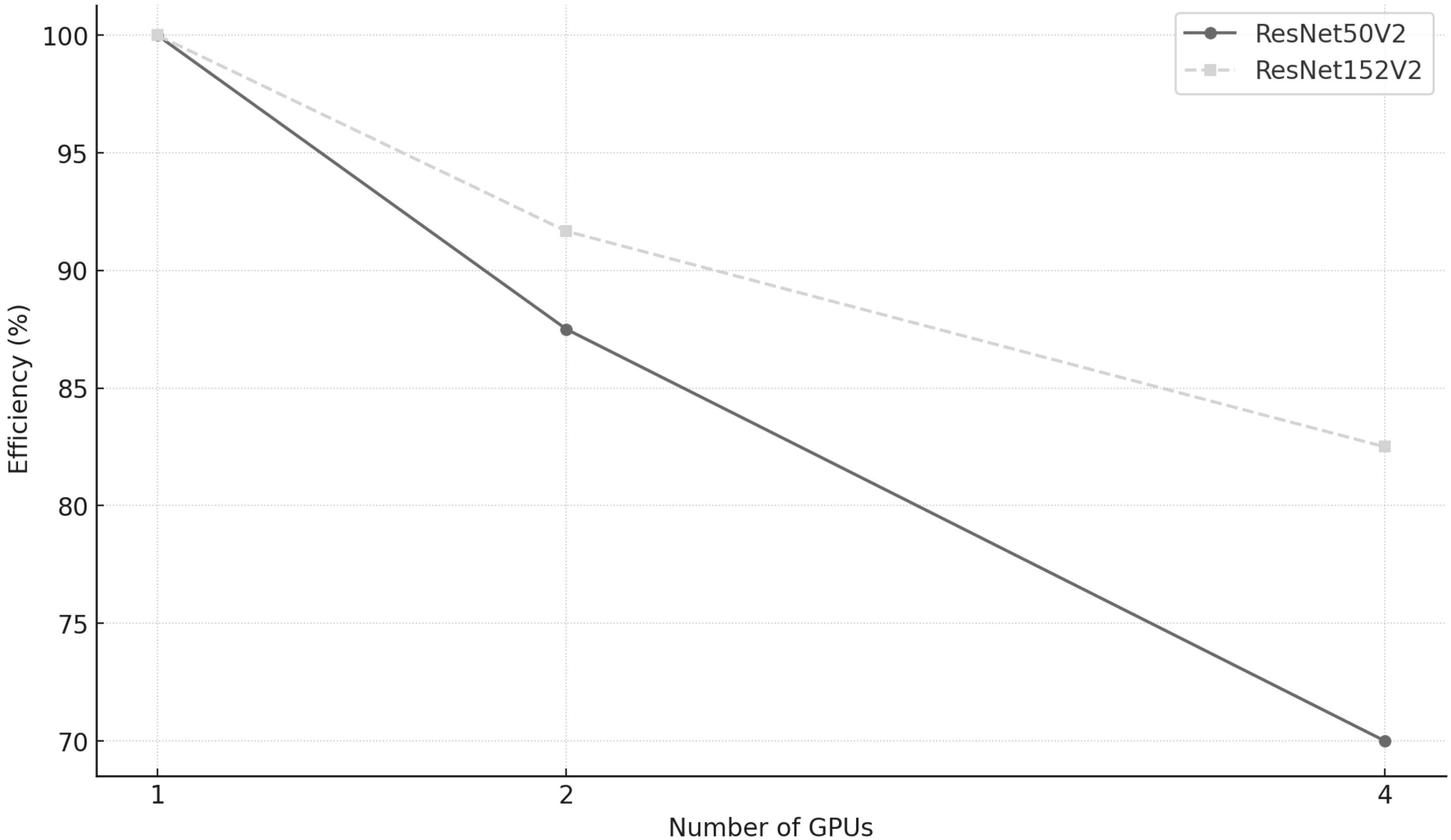
Throughput: ResNet50V2 vs ResNet152V2



Speedup: ResNet50V2 vs ResNet152V2



Efficiency: ResNet50V2 vs ResNet152V2



Takeaways (Model Size & Scaling)

- **Larger models scale better**
  - (more math per sync → higher efficiency).
- **Absolute speed**
  - ResNet50V2 faster & higher throughput.
- **Relative gains**
  - ResNet152V2 benefits more from extra GPUs.
- **Memory footprint**
  - ~60M params (R152) vs ~25M (R50) → consider BF16/mixed precision + activation checkpointing to increase per-GPU batch.

Task 10.7

- **Task 10.7: Parallelize ResNet152V2**
  - Do:
    - New Python + SLURM to run 1/2/4 GPUs.
    - Keep per-GPU batch constant; scale global batch + LR with GPUs.
    - Report: epoch-2 time, global throughput, speedup, efficiency.
    - Analysis: explain scaling vs ResNet50V2.
  - Deliverables: Data + 4 plots (time, throughput, speedup vs ideal, efficiency) + commentary.

Task 10.8

- **Task 10.8: Compare Model Sizes**
  - Produce combined plots for ResNet50V2 vs ResNet152V2:
    - Epoch time, Throughput, Speedup (+ideal), Efficiency.
  - Answer:
    - Why is R152 slower per epoch?
    - Why lower throughput even with more GPUs?
    - How do speedup/efficiency differ?
    - Any added sync overhead from deeper nets?

Task 10.9

- **Task 10.9: Add ResNet101V2**
  - Repeat for ResNet101V2 (1/2/4 GPUs).
  - Plot all three models side by side (same four charts).
  - Discuss:
    - Does ResNet101 scale like ResNet50 or ResNet152?
    - Where do you see diminishing returns or overheads?

Pg: Introduction to Parallel Training

- **Tasks included:**
  - task 10.1 to task 10.9

- **Deliverable & Evaluation:**
  - Upload a single PDF (per group) to the racó@FIB intranet, containing **as many slides as you need per task to clearly express what is requested in each one**.

- **In class (evaluation day):**
  - One group (chosen at random) will give a clear, concise, and straight-to-the-point presentation.
    However, **do not worry about timing — it does not need to follow a strict "elevator pitch" style** as in previous labs.
    The goal here is pedagogical, allowing time to discuss results and reflect on what has been learned.

Lab presentation

- Remember: Good practical experience for students!

and ... a way to stimulate homework accomplishment

- 1 group **will be randomly chosen**
  - We'll sum 4 numbers from randomly chosen students and use the '%' function with the total number of students to find the winner in the list.

```
>>> nums_to_add = ...+...+...+...
>>> winner= nums_to_add % num_students +1
>>> print (winner)
```