# CoTrain: Efficient Scheduling for Large-Model Training upon GPU and CPU in Parallel

Presented by: Saverio Scumaci and Carlos Venegas

# Introduction

- Deep Learning (DL) models have seen significant growth in terms of both the number of parameters and training datasets
  - ChatGPT by OpenAI has reached 175 billion parameters and trained on a dataset of many terabytes
- Heavily rely on GPUs with high computing intensity

# Problem

- Training these large-scale models requires high computational power, primarily relying on GPUs with high computing intensity
    - Mainstream GPUs have limited on-chip memory capacity, making training on large-scale models expensive and challenging

- Efforts have been made to reduce GPU memory requirements
    - Utilize CPU memory and computing resources, such as heterogeneous DL training
    - DeepSpeed's ZeRO-Offload is a recent solution

# Proposed Solution: CoTrain

- Runtime scheduling framework for training DL models
  - Maximizes GPU utilization while offloading specific tasks to the CPU
- Modules: Initialization, Training Engine, Task Scheduler, and Data Allocator,

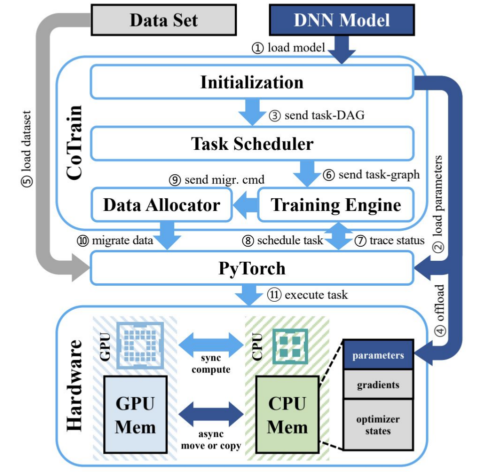



Figure 4: System architecture of CoTrain.

# CoTrain Continued

- Integrated with PyTorch (built on top of)
  - Open-source machine learning framework that provides a flexible and dynamic platform for developing and training deep learning models

# Important Vocabulary

- Gradient - magnitude of the changes that need to be made to the model's parameters during the training process to minimize the loss function
- Loss Function: A function used to measure the error or discrepancy between model predictions and actual target values.
- "k" - decides the allocation of parameter-update tasks, with the first "k" layers assigned to the GPU and the rest to the CPU.

# Initialization

- Initializes the deep learning model by analyzing and registering all layer-level tasks
- Create task Directed Acyclic Graph (DAG)
- Define unique Layer IDs (LIDs) for each layer to manage data positions in CPU memory
- Recording parameters' computational complexity

# Training Engine

- Executes the entire deep learning model training process
  - the forward and backward stages, as well as parameter updates

- Schedules and controls the tasks for each training layer, and managing the asynchronous offloading and synchronization of data between the GPU and CPU memory

- Forward
- Backward
- PARAM-UPDATE
- Separation
- Data Copy
- Data Move

**Algorithm 1** CoTrain

**Training Engine:**

**function** FORWARD()
**for** *layer* $l = 0, ..., n$ **do**
  $index \leftarrow Dict.find(w^{(l+k)}.LID)$
  **call** DATA COPY$(index, w^{(l+k)})$
  $activation\ a^{(t)} \leftarrow \sum_{k=1}^{l_r}(a^{(l-1)}, w^{(l)})$
  $release\ w^{(l)}$
**end for**
**end function**
**function** BACKWARD()
**for** *layer* $l = n, ..., 0$ **do**
  $index \leftarrow Dict.find(w^{(l-k)}.LID)$
  **call** DATA COPY$(index, w^{(l-k)})$
  $gradient\ g^{(l)} \leftarrow \sum_{k=1}^{l_r} \partial\ell(g^{(l+1)}, w^{(l)})$
  $release\ w^{(l)}$
  $index_g \leftarrow Dict\_Grad.find(w^{(l)}.LID)$
  **call** DATA MOVE$(index_g, g^{(l)})$
**end for**
**end function**
**function** PARAM-UPDATE()
**for** *layer* $l = n, ..., 0$ **do**
  **if** $w^{(l+1)}.device = CPU$ **then**
    $in\,CPU:\ w^{(l+1)} \leftarrow w^{(l)} - \eta(g^{(l)} + \partial\Omega(w^{(l)}))$
  **else if** $w^{(l+1)}.device = GPU$ **then**
    $in\,GPU:\ w^{(l+1)} \leftarrow w^{(l)} - \eta(g^{(l)} + \partial\Omega(w^{(l)}))$
    $mem - addr \leftarrow Dict.find(w^{(l)}.LID)$
    **call** DATA COPY$(index, w^{(l)}.data)$
  **end if**
**end for**
**end function**

**Task Scheduler:**

**function** SEPARATION$(time_{BWD}, time_{UPDATE})$
  $separation\ distance\ k \leftarrow \dfrac{time_{BWD} + time_{UPDATE}}{2\,time_{UPDATE}}$
  $task\ graph:\ tg \leftarrow scheduler(k)$
**end function**

**Data Allocator:**

**function** DATA COPY$(src, tgt)$
  $tgt.data \leftarrow tgt.empty\_tensor(src.size, tgt.device)$
  $tgt.data \leftarrow src.copy(src.data)$
**end function**
**function** DATA MOVE$(src, tgt)$
  $src.data \leftarrow src.data.pin\_memory()$
  $src.data \leftarrow src.data.to(tgt.device)$
**end function**

# Task Scheduler

- Partitions and schedules tasks during the training process
- Ensures that tasks are allocated to the appropriate computing resources

# Data Allocator

- Efficiently manages the migration of data between GPU memory and CPU memory
- Ensures that data transfer tasks are executed asynchronously

# Dataflow of Cotrain - GPU CPU Communication

1. GPU sends Layer ID of n-1 to GPU
2. Layer ID converted to CPU Memory Address
3. Data Allocator sends n-1 params to GPU
4. GPU sends n+1 params to Data Allocator
5. Data Allocator sends n+1 params

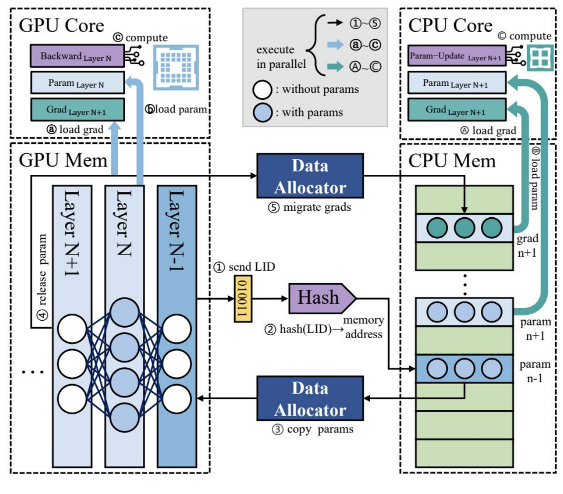


Figure 5: Dataflow of CoTrain

# Dataflow of CoTrain - CPU GPU Computations

Meanwhile, locally…

GPU:

A. Output gradient of n+1 loaded
B. Parameters of n loaded
C. Compute backward gradient

CPU:

A. Gradient received from GPU loaded
B. Parameters of n+1 loaded
C. Run Param Update stage

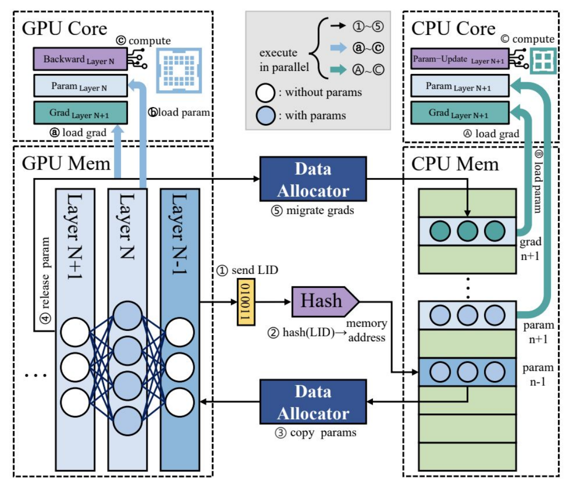


Figure 5: Dataflow of CoTrain

# Parallelization: Requirements and Challenges

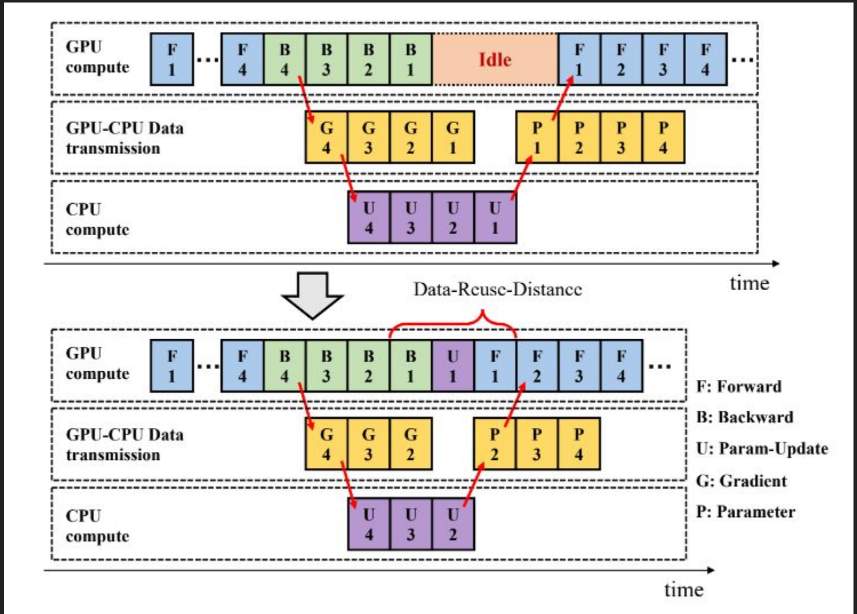
- Backward requires 3 things
  - Data is independent
  - ‘.. the training stage is separable…’
  - Resources are available
- Parallelization may break requirement 1

# Solution to Parallelization Problems

- GPU creates gradients for CPU
- CPU uses a priority queue for gradients
- CPU waits until gradients enter queue
- Uses cuda.stream() for communication

# Cause for Concern

- GPU and CPU wait on each other
- Reusable computations lost



# Workaround

- Computations are saved when possible

# Structure of CoTrain

- Contains 3 Threads
  - Training
  - Transfer
  - Param-Update

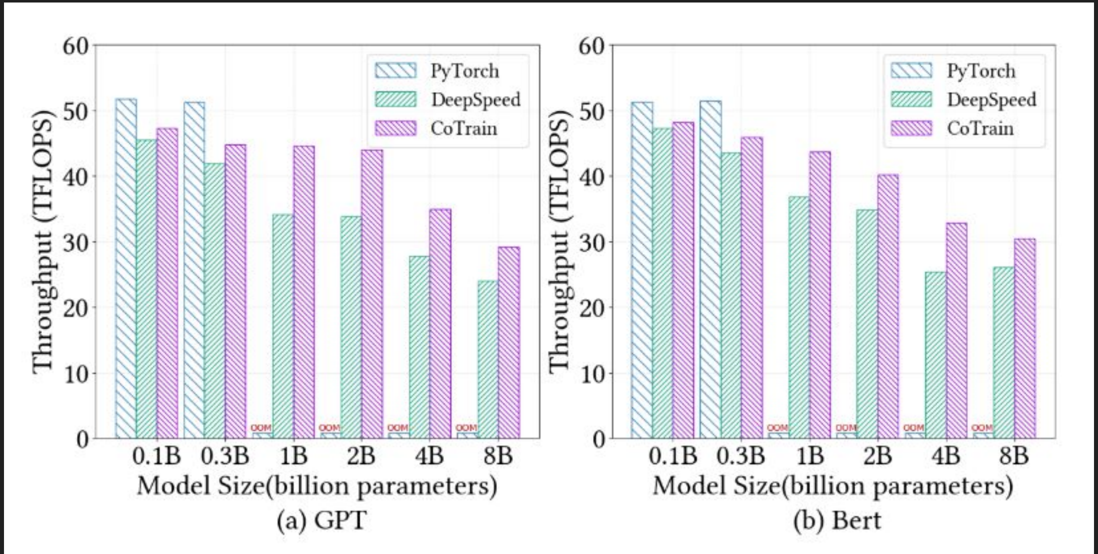
# Evaluation: CPU, GPU, etc.

| Device | Type |
|---|---|
| GPU | NVIDIA TITAN RTX |
| GPU Mem | 24GB HBM2 |
| CPU | Intel Xeon CPU E5-2683(14-core,2.00GHz) |
| CPU cache | L1-32K, L2-3.5M, L3-35M |
| CPU Mem | 128GB 2133MHz DDR4 |
| PCIe | PCIe 3.0 |

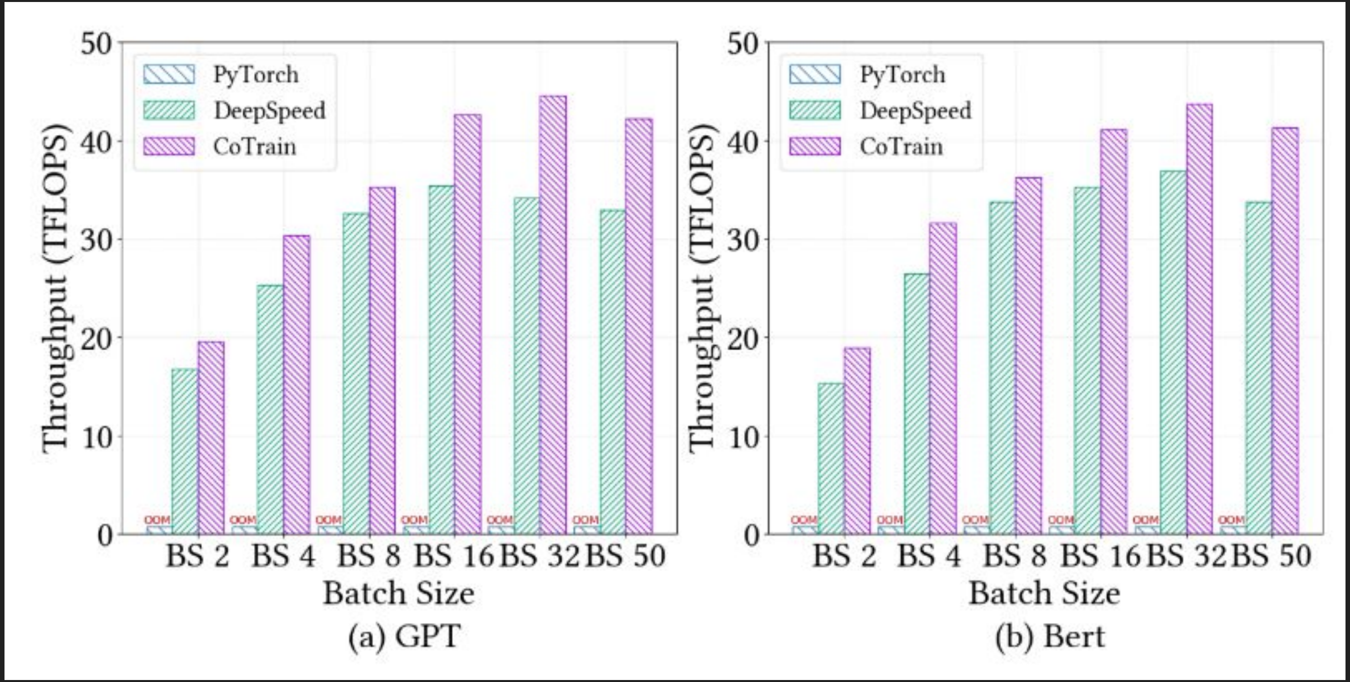
# More Specifications

- Used with ChatGPT and Bert
- Used Stanford Question Answering Dataset
- Compared to PyTorch and DeepSpeed

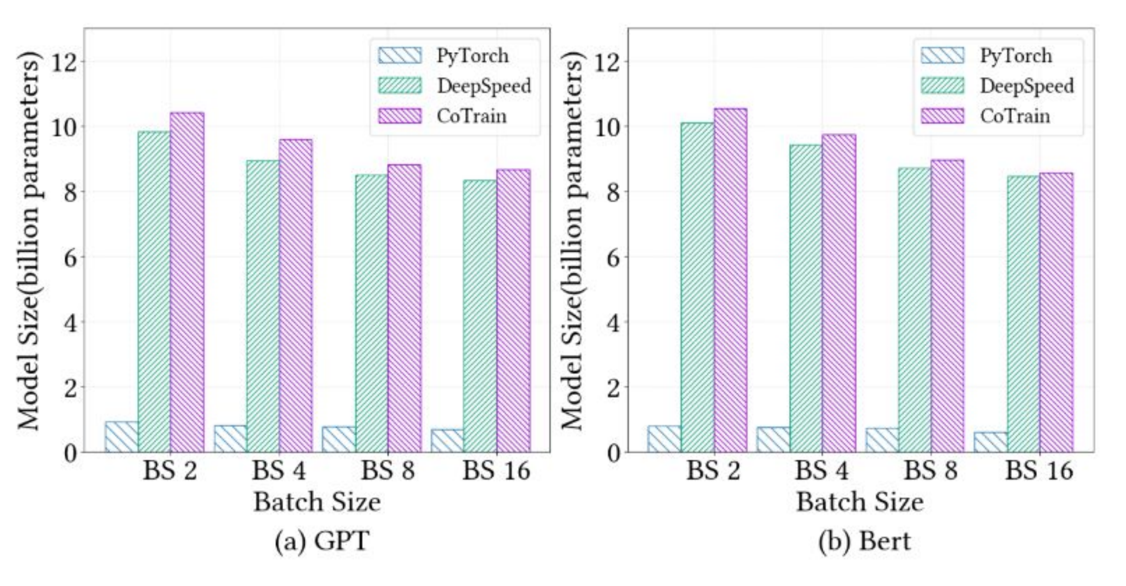
# Figure 7: Training Throughput



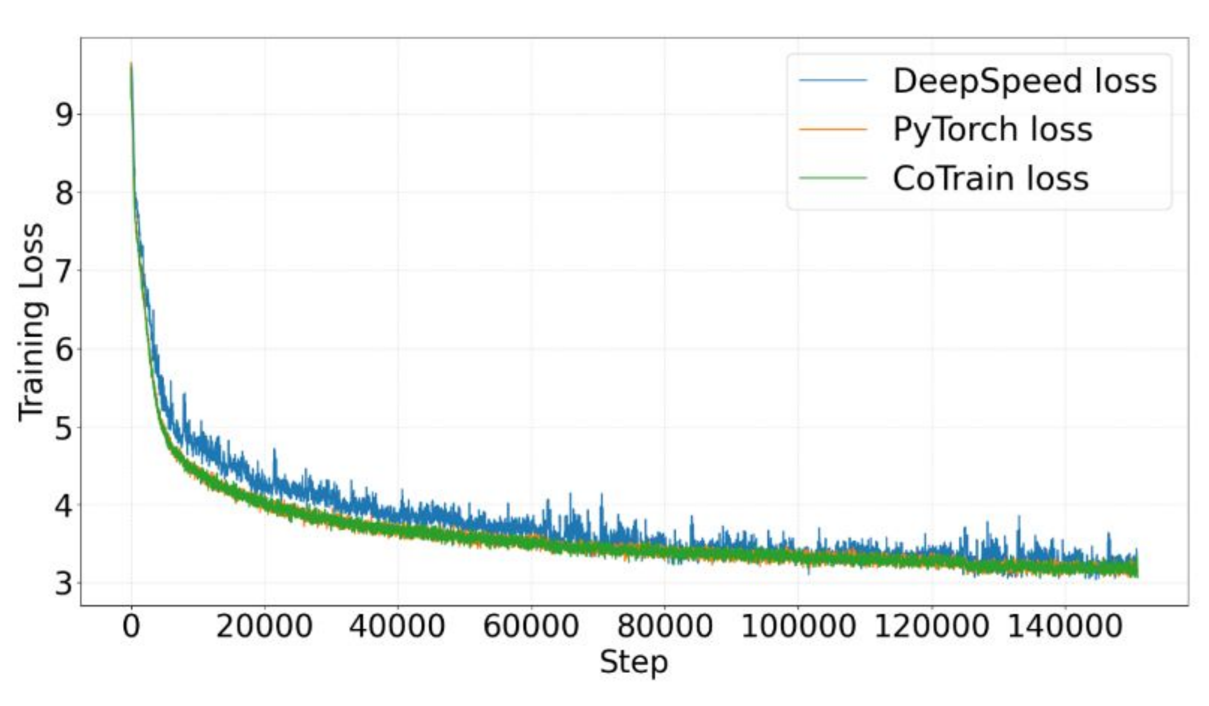
# Figure 8: Throughput of PyTorch, DeepSpeed and Cotrain in Various Batch Sizes



# Figure 9: The Max Model Size for Different Batch Size



# Figure 10: Model Convergence



# Figure 11: The Idle Time in the Whole Step Time