Objective: Explore DeepSpeed features (ZeRO stages, mixed precision, model parallelism, offload) and push the limits for training large models (Pythia family).

Environment:

- **Hardware**: 2 × T4 GPUs.
- **Software**: PyTorch 2.0, DeepSpeed 0.16.7
- **Model**: EleutherAI/pythia-160m, EleutherAI/pythia-410m for Question 5

1 Question 1: ZeRO Optimization Stages

1.1 Concept & Configuration Setting

Concept Details: DeepSpeed's ZeRO is a memory optimization technology that partitions optimizer states, gradients, and model parameters across GPUs during distributed training, reducing per-device memory requirements.

Stage	What's Sharded	Memory Savings	Communication Performance
1	Optimizer states only	Moderate	Minimal overhead
2	+ Gradients	Higher	Moderate overhead
3	+ Model parameters	Highest	Highest overhead

Table 1: ZeRO: Sharding Strategy, **Expected** Memory Savings and Performance Trade-offs

Key Config in ds_config.json: In the ds_config.json file, set the zero_optimization.stage parameter to 1, 2, 3 while keeping other parameters fixed. Evaluate ZeRO Stages 1-3 using the pythia-160m model on two GPUs with P2P communication disabled and the fixed batch size and learning rate. All other batch sizes and gradient accumulation are set to "auto" so DeepSpeed dynamically balances throughput and memory.

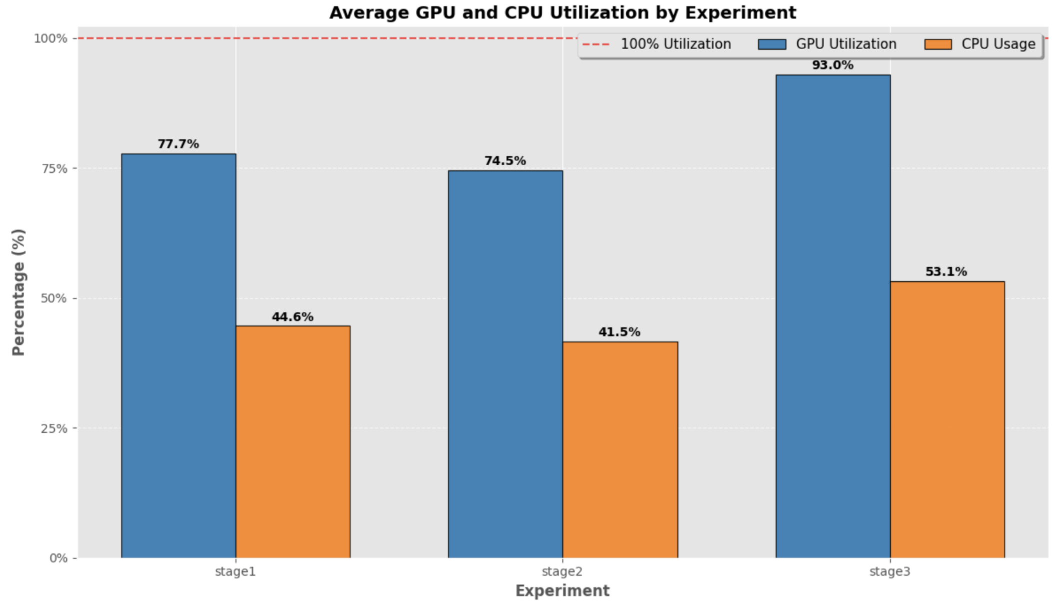
```
{
    "zero_optimization": {
        "stage": 1
    },
    "fp16": {
        "enabled": true
    },
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "gradient_accumulation_steps": "auto"
}
```

1.2 Experimental Result

Result Observation:This table and bar chart summarize how each ZeRO stage impacts cpu and gpu memory utilization and overall training speed. Stage 2 has extra communication overhead for only marginal memory savings, while Stage 3 trades higher memory usage for the best throughput.

Stage	GPU Memory (MB)	GPU Util. (%)	CPU Util. (%)	Step Time (s)	Speed (steps/s)
1	6,940.2	77.7	44.6	0.89	1.12
2	6,966.9	74.5	41.5	0.93	1.08
3	8,433.3	93.0	53.1	0.804	1.24

Table 2: Resource utilization and throughput across ZeRO optimization stages



1.3 Analysis & Insights

- **Partial Sharding Inefficiency**: Stage 2's gradient sharding adds overhead without significant memory savings, making it inefficient for small models on limited GPUs.
- **Memory–Throughput Tradeoff**: Full parameter sharding of Stage 3 increases memory usage but provides the highest throughput, perform well when speed is the priority.
- **Rely on Model Scale & Hardware**: Higher ZeRO stages may not provide benefits for small models or limited GPUs. So empirical benchmarking is key.
- **Suggestion**:For small and medium-sized models, use ZeRO 1 for simplicity and efficiency, and use ZeRO 3 only if extra memory is available. For large models, higher ZeRO stages offer memory savings—combine ZeRO 3 with CPU offloading for maximum capacity.

2 Question 2: Mixed Precision Training

2.1 Concept & Configuration Setting

Concept Details: Mixed-precision (FP16/BF16) instead of FP32 in deep learning models, particularly with modern GPUs, is mainly due to it can reduce memory and speed up kernels on modern GPUs.

Format	Dynamic Range	Precision	Memory Usage	Speed (vs FP32)
FP32	Wide	High	High	Baseline
FP16	Narrower	Lower	½ of FP32	1.5–2× faster
BF16	Wider (vs FP16)	Lower	½ of FP32	1.5–2× faster

Table 3: Comparison: Dynamic Range, Precision, **Expected** Memory Usage and Speed

- **FP32**: The 32-bit floating-point format (FP32) offers high precision, particularly for representing both small and large numbers. However, this precision comes at the cost of increased memory usage and computational power.
- **FP16**: The 16-bit floating-point format (FP16) reduces precision compared to FP32 but is often sufficient for many deep learning applications, particularly when training large models at a faster pace.

- **BF16**: The bfloat16 format (BF16) also uses 16 bits, but it differs from FP16 by allocating more exponent bits (8 for BF16 vs. 5 for FP16). This provides a wider dynamic range, making BF16 more suitable for tasks that require handling a broader range of values. However, BF16 sacrifices some precision in the fractional part, which can affect fine-grained computations.

Key Config in ds_config.json: To switch precision training and compare, enable the corresponding section in your DeepSpeed config. Other settings such as optimizer and batch size held constant across runs, zero_optimization.stage=1.

Precision	fp16.enabled	bf16.enabled
FP32 (Baseline)	false	false
FP16	true	false
BF16	false	true

Table 4: DeepSpeed Mixed-precision training Configuration

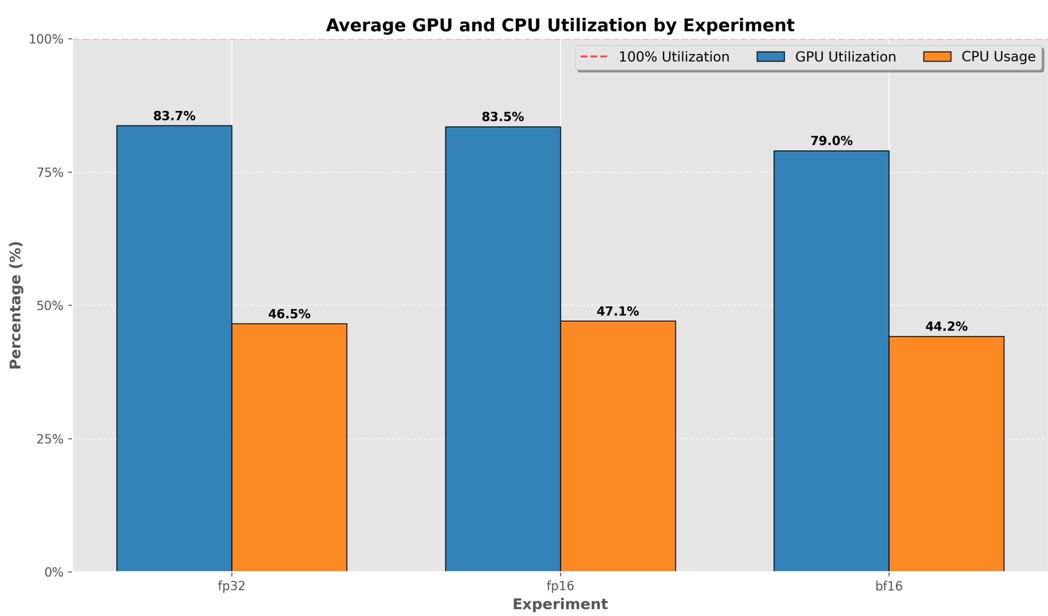
2.2 Experimental Result

Result Observation:Across identical model, batch size, and hardware:

- **FP32** serves as the stability baseline with mid-tier memory use and throughput.
- **FP16** offers the largest GPU memory reduction (35% vs. FP32) and the highest throughput (×2.7).
- **BF16**: delivers modest memory savings (10%) but falls slightly behind FP32 in speed on this GPU.

Precision	GPU Memory (MB)	GPU Util. (%)	CPU Util. (%)	Step Time (s)	Speed (steps/s)
FP32	6,954.5	83.7	46.5	0.89	1.12
FP16	4,528.8	83.5	47.1	0.34	2.98
BF16	6,241.6	79.0	44.2	1.06	0.94

Table 5: Resource utilization and throughput across different precision configurations



2.3 Analysis & Insights

- **Memory vs. Speed Trade-off**: FP16 reduces memory by 35% and delivers a 166% speedup over FP32. BF16 reduces memory by 10% but incurs a 16 slowdown relative to FP32.
- **Utilization Stability**: GPU utilization stays above 79% across all modes, indicating efficient compute use. CPU utilization remains around 44–47%, showing no notable host-side bottleneck.
- **Suggestion**: Use FP16 mixed-precision for best memory savings and throughput on modern NVIDIA GPUs. Reserve BF16 for hardware (e.g., recent AMD/Intel or custom accelerators) with robust BF16 units, or when FP16 stability issues arise.

3 Question 3: Model Parallelism

3.1 Concept & Configuration Setting

Concept Details:

Aspect	Data Parallelism	Model Parallelism
What is parallelized?	Training data (each GPU processes different data)	The model (each GPU holds part of the model)
Memory usage	Multiple copies of the model are stored across GPUs	Model is split across GPUs to avoid memory overflow
When to use	When the model fits in GPU memory, but the dataset is large	When the model is too large to fit in a single GPU's memory
Communication	GPUs communicate gradient updates after each pass	GPUs exchange intermediate results during forward/backward pass

Table 6: Comparison of Data Parallelism and Model Parallelism

Key Config in ds_config.json: Enable ZeRO stage 3 for maximal memory savings, FP16 for mixed precision, and pipeline-parallelism with 2 stages and a micro-batch size of 1. All other batch sizes and gradient accumulation are set to "auto" so DeepSpeed dynamically balances throughput and memory.

- **pipeline**: Enables stage-wise layer partitioning.
- **stages**: Number of pipeline splits. Set to 2, meaning the model will be split across two devices.
- **micro_batch_size**: This defines how small the batch is for each device.
- **intermediate_batch_size**: Controls activation buffering ("auto" will let DeepSpeed pick based on model's memory and GPU constraints).

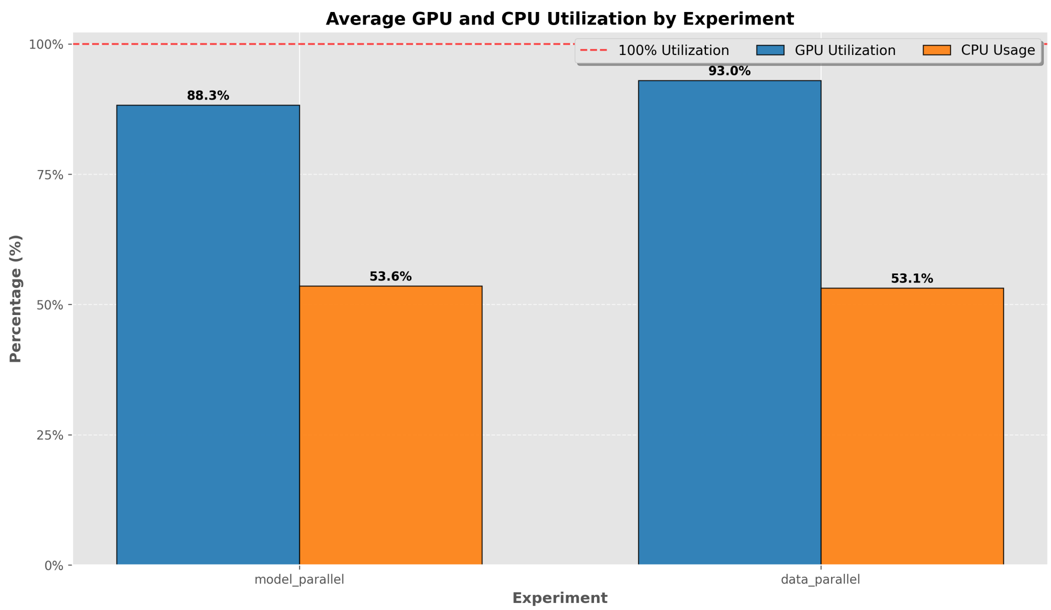
```
{
  "zero_optimization": {
    "stage": 3
  },
  "fp16": {
    "enabled": true
  },
  "pipeline": {
    "enabled": true,
    "stages": 2,
    "micro_batch_size": 1,
    "intermediate_batch_size": "auto"
  },
  "train_batch_size": "auto",
  "train_micro_batch_size_per_gpu": "auto",
  "gradient_accumulation_steps": "auto"
}
```

3.2 Experimental Result

Result Observation: Implementing 2-stage pipeline parallelism with ZeRO 3 and FP16 reduces GPU memory by 31%, more than doubles iteration throughput, and keeps CPU utilization virtually unchanged, compared to data parallelism on the same two GPUs.

Stage	GPU Memory (MB)	GPU Util. (%)	CPU Util. (%)	Step Time (s)	Speed (steps/s)
Model Parallelism	5,793.1	88.3	53.6	0.295	3.39
Data Parallelism	8,433.3	93.0	53.2	0.806	1.24

Table 7: Resource utilization and throughput for different parallelism strategies



3.3 Analysis & Insights

- **GPU Utilization Trade-off**: Utilization dips slightly (93% → 88%) due to pipeline fill/drain phases, but the net throughput gain outweighs this.
- **Memory Efficiency**: Splitting the model across two GPUs cuts per-GPU memory use by 31%, enabling much larger networks or higher per-GPU batch sizes.
- **Throughput Gain**: Despite pipeline bubbles, the smaller micro-batches and off-loaded parameters more than double effective iteration speed (1.24 → 3.39 it/s).
- **Summary**: DeepSpeed's pipeline parallelism allows training large models across GPUs, trading slight communication overhead for significant memory savings.

4 Question 4: Offload Techniques

4.1 Concept & Configuration Setting

Concept Details: Offload optimizer states and/or model parameters into host RAM. DeepSpeed provides two ZeRO offload modes to relieve GPU memory pressure: CPU Offload and NVMe Offload. Only try CPU Offload for experiment.

Key Config in ds_config.json: Enable ZeRO stage 3 offloading of optimizer states and/or parameters to CPU (or NVMe) with optional pinning for higher throughput

- **offload_optimizer**: Keep the optimizer's internal states (e.g., Adam moments) in CPU RAM instead of GPU.

- **offload_param**: Keep the model's weights in CPU RAM and only load them onto GPU when needed.

```
{
  "zero_optimization": {
    "stage": 3,
    "offload_optimizer": { "device": "cpu", "pin_memory": true },
    "offload_param":     { "device": "cpu", "pin_memory": true }
  },
  "fp16": { "enabled": true },
  "train_batch_size": "auto",
  "train_micro_batch_size_per_gpu": "auto",
  "gradient_accumulation_steps": "auto"
}
```

Aspect	Offload to CPU	No Offload
What is offloaded?	Optimizer states & model parameters	Nothing
Memory location	CPU host RAM (pinned)	GPU device memory
When to use	When GPU memory is a bottleneck	When ample GPU memory is available
Trade-off	↓ GPU memory use, ↑ CPU involvement, ↓ throughput	↑ GPU memory use, ↑ throughput, minimal CPU impact

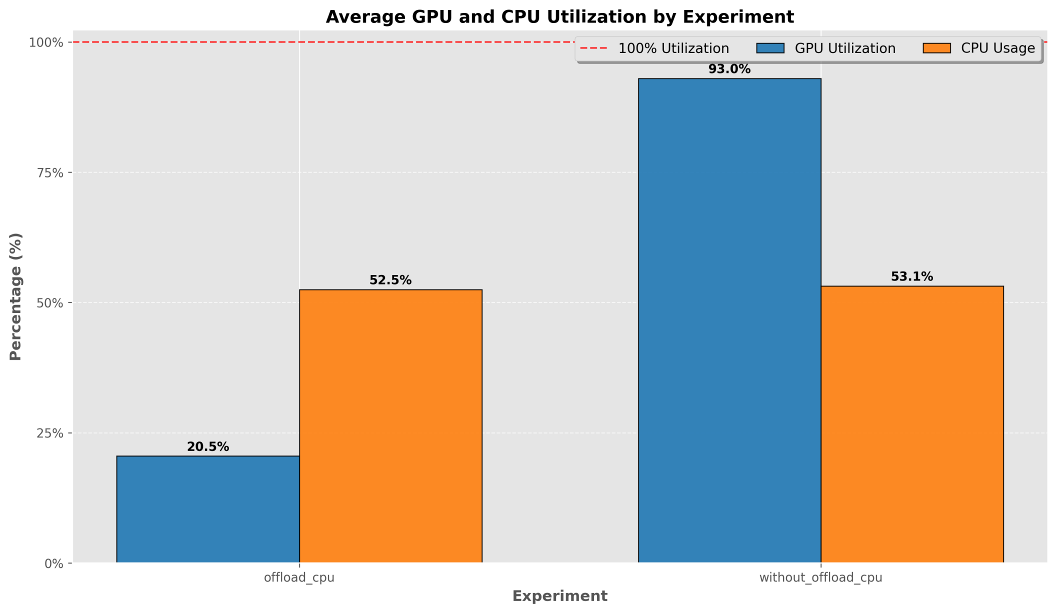
Table 8: Comparison of Offload to CPU and No Offload

4.2 Experimental Result

Result Observation: Offloading to CPU cuts GPU memory usage by over 56% but reduces iteration throughput by 37%, while CPU utilization rises modestly.

Offload Mode	GPU Memory (MB)	GPU Util. (%)	CPU Util. (%)	Step Time (s)	Speed (steps/s)
Offload to CPU	3,714.0	20.5	52.5	1.2426	0.80
No Offload	8,433.3	93.0	53.2	1.2420	1.24

Table 9: Resource utilization and throughput with/without cpu offload method



4.3 Analysis & Insights

- **Device Utilization**: GPU utilization plunges (93% → 21%) when offloading, reflecting idle GPU cycles waiting on CPU, while CPU utilization stays similar (53%), indicating headroom but also coordination overhead.
- **GPU Memory Savings**: Offloading parameters and optimizer states to CPU reduces per-GPU memory from 8.4 GB to 3.7 GB (–56%), enabling larger models or batch sizes on the same hardware.
- **Throughput Trade-off**: Despite vastly lower GPU memory pressure, iteration speed drops from 1.24 it/s to 0.80 it/s (–35%) as CPU–GPU data transfers and CPU processing introduce latency.
- **Summary**: DeepSpeed offload strategies reduce GPU memory usage by transferring optimizer state and model parameters to the CPU. This reduces GPU use and step rate but enables training extremely large models on GPUs with limited memory when some speed tradeoff is acceptable.

5 Question 5: Training Larger Models

5.1 Concept & Configuration Setting

Concept Details: Select larger Pythia model: 160M → 410M. Apply ZeRO-3 + CPU offload + BF16 + model parallelism.

- **ZeRO 3** with both optimizer- and parameter-offload to CPU to fit a larger model in limited GPU RAM.
- **bf16**: precision to cut memory footprint 2× vs FP32.
- **2-stage pipeline parallelism** to split the model across two GPUs.
- **Auto batch-size/accumulation** lets DeepSpeed choose the largest stable micro-batch and grad-acc steps.

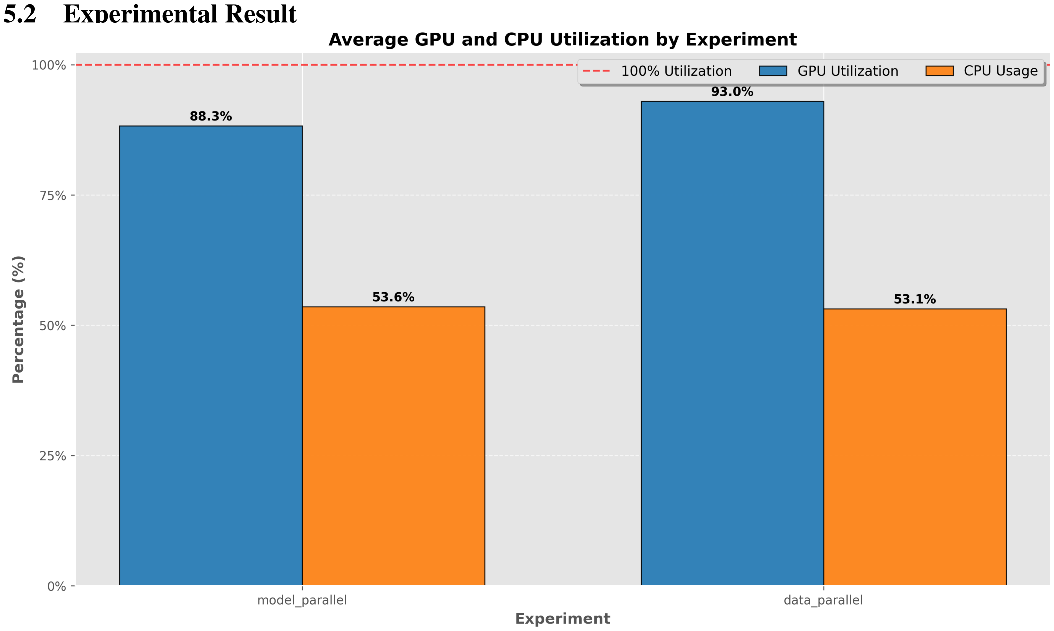
Key Config in ds_config.json:

```json
{
  "zero_optimization": {
    "stage": 3,
    "offload_optimizer": { "device": "cpu", "pin_memory": true  },
    "offload_param":     { "device": "cpu",  "pin_memory": true }
  },
  "bf16":     { "enabled": true },
  "pipeline": {
      "enabled": true,
      "stages": 2
  },
  "train_batch_size": "auto",
  "gradient_accumulation_steps": "auto"
}
```

Metric	GPU Memory (MB)	GPU Util. (%)	CPU Util. (%)	Time/Iter (s)	Est. Time (h:m)
Training Progress	8,133.5	55.4	53.1	3.46	2:01

Table 10: Training resource usage and estimated completion time (2,108 iterations) on pythia-160m

5.2 Experimental Result



5.3 Analysis & Insights

- **GPU Memory Savings**: Offloading both optimizer state and parameters to CPU combined with bfloat16 precision keeps per-GPU memory at 8.1 GB, making a 410 M-parameter model feasible on a 16 GB card.
- **Throughput Trade-off**: Each iteration takes 3.46 s (0.29 it/s). Offload-induced data movement and CPU processing introduce latency compared to an all-GPU run, but the trade-off is acceptable for scaling.
- **Scalability**: The model size grew 2.6× (160 M→410 M), yet GPU memory only rose 2.3×—a sub-linear increase thanks to ZeRO 3 + offload + bf16.