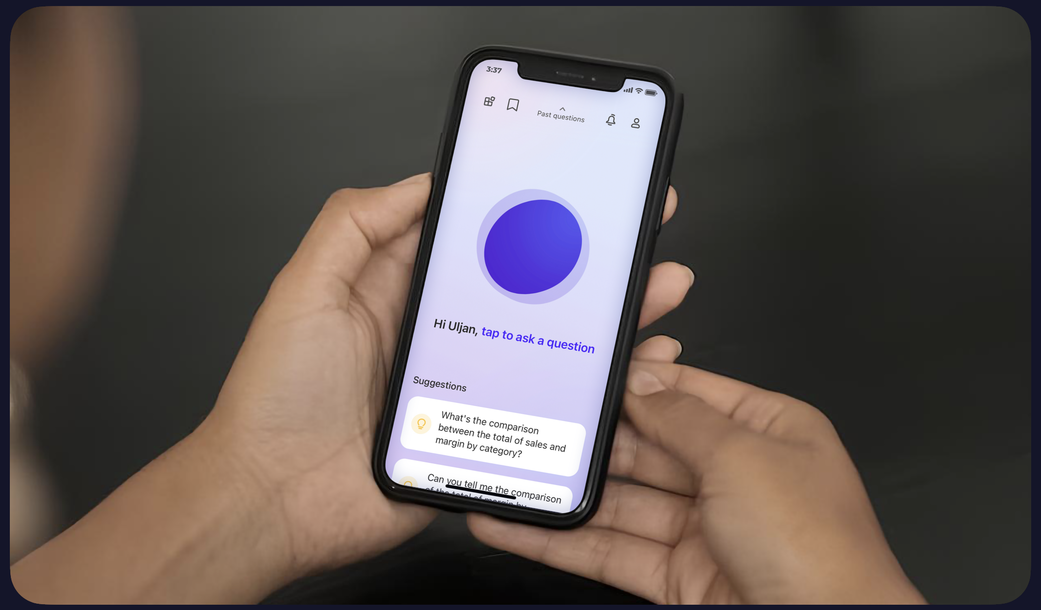
iGenius

User Best Practices and Results on Leonardo

Paolo Albano

Our mission



Humanizing data, democratizing knowledge.

Our goal is to enable anyone to explore company data and ask business questions in natural language. We are committed to making data analysis accessible, secure and private, supporting businesses in highly regulated sectors.

Our vision

Becoming the leading force in AI for businesses.

We believe in AI solutions that prioritize people over data, putting technology at the service of humanity. We aim to create a world where data becomes second nature to everyone, fostering a decision-making culture that can finally be inclusive and intuitive.

We help companies transform data into decision intelligence

Crystal

The revolution of the user interface

Crystal is an AI-powered Decision Intelligence tool for analyzing your business data in natural language. Interacting with Crystal is as easy as talking to your team!

Italia

Our first LLM, 100% Open Source

Italia 9B is a foundational LLM with 9 billion parameters, a 4,096-token context window, and a 50,000-token vocabulary, trained entirely from scratch in Italian on trillions of tokens from diverse data sources, including public, synthetic, and domain-specific content.

This exclusive Italian training enables Italia 9B to capture linguistic and cultural nuances with exceptional precision, without relying on English translations.

Italia 9B is developed in collaboration with Cineca and released under the MIT license.

Why use HPC: Model Complexity

How much memory does it take?

Memory for model parameters

- Number of parameters: 9 Billions
- Memory for parameters
- Params Memory = N x bytes per parameter

(FP32 param (32-bit floating point): 4 bytes per parameter)

Total parameter Memory = 4 bytes x 9 × 10^9 = 36GB

Memory for optimizer (AdamW)

Total Optimizer ≈ 3 x Parameter Memory

(same amount of model parameters for gradient, first and second moments)

Batch Memory

Based on batch size and number of tokens per batch (context length)

Batch Memory = batch size × tokens per batch × embedding size × bytes per embedding

More memory as overhead

- Memory for network communication
- Possible gradient accumulation
- Pytorch buffers

Total Memory ≈ Parameters Memory + Optimizer Memory + Batch Memory + Overhead

Why use HPC: Dataset Complexity

Today v0.1

Composition	90% Italian data, 10% English data
Dataset size	Around 1.2 Trillion of tokens
Disk size	Around 4TB

Tomorrow CPT

Composition	50% Italian data, 50% English data
Dataset size	Around 3 Trillion of tokens

Why use HPC: Porting from torchrun to slurm

Manually launch on multiple nodes

(example with 2 nodes and 4gpu each node)

On master node

```
torchrun \
    --nproc_per_node=4 \
    --nnodes=2 \
    --node_rank=0 \
    --master_addr="hostname-master" \
    --master_port=1234 \
    script.py
```

Other nodes

```
torchrun \
    --nproc_per_node=4 \
    --nnodes=2 \
    --node_rank=1 \
    --master_addr="hostname-master" \
    --master_port=1234 \
    script.py
```

Move everything to slurm

```
#!/bin/bash

#SBATCH --job-name=torchrun_to_slurm          # Job Name
#SBATCH --nodes=2                             # Number of nodes
#SBATCH --ntasks-per-node=4                   # 1 for each gpu
#SBATCH --gres=gpu:4
#SBATCH --cpus-per-task=8
#SBATCH --time=02:00:00                       # Limit execution time
#SBATCH --partition=partition_name            # Nome della partizione (es. gpu)
#SBATCH --output=output_%j.txt                # Output log
#SBATCH --error=error_%j.txt                  # Error log

module load python/3.11.6
module load cuda/12.1

# first node as master
MASTER_ADDR=$(scontrol show hostname $SLURM_NODELIST | head -n 1)
MASTER_PORT=1234

NODE_RANK=$SLURM_NODEID

NNODES=$SLURM_NNODES

NPROC_PER_NODE=4

torchrun \
    --nproc_per_node=$NPROC_PER_NODE \
    --nnodes=$NNODES \
    --node_rank=$NODE_RANK \
    --master_addr=$MASTER_ADDR \
    --master_port=$MASTER_PORT \
    script.py
```

Why use HPC: HPC vs. cloud vendor

Pros

Dedicated Resources

No shared resources when get exclusive nodes, which can decrease performance

Customization

Greater control over the environment, software stack, and hardware optimizations tailored to specific needs.

Ease of Job Scaling

Facilitates running the same job across different nodes, useful in the early stages of a project

Simplified Porting

The process of porting deep learning model training is greatly simplified with frameworks like Pytorch

Faster GPU Provisioning

Provisioning nodes with GPUs can be quicker than with cloud vendors, especially due to regional limitations

Cons

Onboarding Phase

A brief onboarding phase is necessary to understand how to use the HPC effectively, including job submission, queue management, job dependencies, and monitoring

Not all HPC are LLM ready

Efficient LLM training requires not only powerful and modern GPUs (e.g., support for FlashAttention), but also fast communication between GPUs (Infiniband) and high-speed storage to optimize data throughput. Training jobs for an LLM last several days, which introduces infrastructural complexities.

Resource Optimization: GPU Load Optimization

To achieve optimal GPU utilization between 70% and 90%, it's essential to conduct various test trainings to identify the hyperparameter configurations that yield the best performance. Below this threshold, GPU utilization is considered underused, while exceeding this range can lead to potential issues that may slow down the training process:

Increased Latency

One or more GPUs may experience higher latency, creating bottlenecks that can hinder overall performance.

Garbage Collection

Triggering the PyTorch garbage collector more frequently can introduce delays in the training loop.

Out of Memory (OOM) Risks

Operating above the optimal GPU utilization increases the risk of encountering OOM errors, which can halt the training process. By carefully tuning hyperparameters and monitoring GPU performance, we can strike a balance that maximizes efficiency while minimizing these risks.

We use **Weight & Biases** (W&B) for tracking training jobs, logging metrics, and managing experiments.

However, since Leonardo's nodes are not connected to the internet, W&B cannot be used as it continuously sends information to W&B's online platform.

To work around this, W&B can be used in offline mode and later synced manually.

add as env variable:
export WANDB_MODE=offline

The job will create a folder wandb that can be sync manually with:
wandb sync --sync-all

A monitoring tool is essential for these evaluations. In our case, we chose **Weight & Biases.**

Resource Optimization: Checkpoint Strategy

Optimizing checkpointing during the training of a Large Language Model (LLM) is crucial to balance the time spent saving checkpoints and the ability to resume training without significant loss in case of interruptions.

- During the initial tests, based on the duration of the init phase, you can decide to **save every n hours of training** (usually a good compromise is to save every 3-5 times the duration of the init phase) or an equivalent number of steps.
- **Don’t waste disk space:**
  - Reduce the Number of Checkpoints kept
  - Metric-based checkpoints
- **Optimized Distributed Checkpointing:** Some acceleration frameworks (like DeepSeed) provide tools to manage distributed checkpointing, avoiding redundant saving on each device.
- **Use High-Performance File Systems for Checkpointing:** on Leonardo, we observed a performance increase by switching to the fast partition, reducing the inizialization time from 1 hour to approximately 20 minutes
- Remember to submit the same job in as dependency to **automate the resume process**. This way, if the job fails or is interrupted, it can automatically restart from the last checkpoint without manual intervention.

Resource Optimization: Pre-Tokenization

Why Pre-Tokenize?

Speed-up Training	Pre-tokenizing the dataset before training can significantly reduce training time by eliminating the need for tokenization during the training loop.
Reduced Latency	By pre-processing data, we minimize delays during training iterations, leading to a smoother training experience.
Simplified Pipeline	Streamlines the data loading process, making it easier to manage large datasets.
Tokenization CPU-Intensive Process	Since tokenization is CPU-intensive, we can create a dedicated job for this task, avoiding the use of GPU nodes and freeing up valuable resources for other processes.
Easier Optimization and Partitioning	Pre-tokenizing the dataset allows for easier optimization and partitioning based on token counts. Only after tokenization do we know the precise number of tokens in our dataset, enabling more informed decisions for data management.

Resource Optimization: Training Parallelism

Training parallelism refers to techniques that distribute the training workload across multiple processors (CPUs/GPUs) to speed up the training process and handle larger models and datasets.

Data Parallelism

- Distributes input data across multiple GPUs.
- Each GPU processes a portion of the data independently.
- Can be effectively used on relatively small models to enhance training speed without requiring extensive resources.

Model Parallelism

- Splits the model itself across multiple devices.
- Useful for very large models that do not fit into a single GPU's memory.

What we used

FSDP hybrid: is a training strategy that combines data parallelism and model parallelism to optimize resource utilization and memory efficiency. Apply FULL_SHARD within a node, and replicate parameters across nodes, reduced communication volume that could be a bottleneck.

Resource Optimization: Performance Optimization

Mixed Precision Training

Mixed precision training utilizes both 16-bit (BF16) and 32-bit (FP32) floating-point types during model training. This approach leverages the speed of BF16, particularly optimized for NVIDIA GPUs, while maintaining the numerical stability of FP32 for certain calculations.

- Faster Computations
- Reduced Memory Usage (see Overhead)

Gradient Accumulation

The optimizer step is applied only after a specified number of mini-batches have been processed.

- Enables effective training with larger effective batch sizes without exceeding memory limits, improving model convergence
- Balances the workload across multiple iterations, ensuring that GPUs remain efficiently utilized without the risk of memory overflow.

Thank you