# Sistema de ficheiros otimizado para o treino de modelos de Deep Learning

Augusto Campos
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal, PG57510

Carlos Silva
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal, PG57518

Tiago Silva
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal, PG57617

## Abstract

Este trabalho apresenta o *desenho* e a *implementação* de um **sistema de ficheiros distribuído** orientado para cargas de trabalho de **Deep Learning**. Desenvolvido com recurso à plataforma **FUSE** e com suporte à interface **POSIX**, o sistema garante compatibilidade com ferramentas como o *PyTorch* ou o *TensorFlow*. A arquitetura adotada segue um modelo *cliente-servidor*, onde o cliente interage com múltiplos servidores responsáveis pela gestão e armazenamento dos dados. Para responder aos requisitos de *desempenho* e *resiliência* típicos do treino de modelos de Deep Learning, o sistema integra mecanismos de *replicação de dados* para *tolerância a faltas* e uma *cache* no lado do cliente para otimização das operações de leitura. A *avaliação experimental* comprova os benefícios destas funcionalidades na melhoria da *robustez* e *eficiência* do sistema em cenários reais/sintéticos de treino.

## Keywords

Sistema de ficheiros distribuído, FUSE, POSIX, Deep Learning, Tolerância a faltas, Cache, Desempenho, TensorFlow

## 1 Introdução

O rápido crescimento da utilização de modelos de **Deep Learning** tem levado a um aumento significativo nos requisitos computacionais e de armazenamento associados ao seu treino. Estas cargas de trabalho, frequentemente *intensivas em dados*, requerem **sistemas de ficheiros** capazes de fornecer acesso rápido, eficiente e confiável a grandes volumes de informação. No entanto, os sistemas de ficheiros tradicionais nem sempre estão preparados para responder às particularidades deste tipo de utilização, nomeadamente no que diz respeito à *redundância*, *latência de acesso* e *resiliência a falhas*.

Neste contexto, surge a necessidade de desenvolver *soluções de armazenamento especializadas* para este tipo de carga de trabalho. Entre os desafios que se colocam, destaca-se a capacidade de garantir **tolerância a faltas**, mantendo a *disponibilidade* e *integridade* dos dados mesmo perante falhas de servidores, e a **otimização do desempenho**, reduzindo *tempos de leitura* que impactam diretamente a duração do treino. Adicionalmente, é essencial manter a **compatibilidade com interfaces padrão como POSIX**, de forma a assegurar a integração com **frameworks** amplamente utilizadas, como o *PyTorch* ou o *TensorFlow*.

Este trabalho propõe o desenvolvimento de um **sistema de ficheiros distribuído**, implementado com base na plataforma **FUSE** e compatível com a interface **POSIX**, especificamente orientado para cargas de trabalho de **Deep Learning**. O sistema segue uma arquitetura *cliente-servidor* e incorpora duas funcionalidades centrais: **(i)** um mecanismo de **replicação** de dados, com o objetivo de garantir **tolerância a faltas**, e **(ii)** uma **cache** no lado do cliente, concebida para **otimizar o desempenho** em operações de leitura, especialmente relevantes no contexto de treino de modelos.

Os resultados obtidos através da *avaliação experimental* — conduzida com *cargas sintéticas* que simulavam os padrões de acesso de treino com *TensorFlow* — demonstram que o sistema desenvolvido consegue manter-se funcional perante *falhas de servidores* e apresentar melhorias notáveis em termos de **latência de leitura**, quando comparado com abordagens sem cache. Estas evidências validam as *decisões de desenho* e mostram o **potencial da solução proposta** como base para sistemas de armazenamento adaptados a ambientes de **Inteligência Artificial**.

## 2 Desenho e Arquitetura da Solução

A solução proposta segue uma **arquitetura cliente-servidor**, suportada pela biblioteca **FUSE (Filesystem in Userspace)**, que permite a implementação de um **sistema de ficheiros totalmente compatível com a interface POSIX** sem a necessidade de desenvolvimento em *kernel space*. O sistema é composto por três componentes principais: a *aplicação cliente* (tipicamente um processo de treino de modelos de *Deep Learning*), o *módulo de FUSE* que implementa a lógica do sistema de ficheiros, e um *conjunto de servidores de armazenamento* onde os dados estão efetivamente guardados.

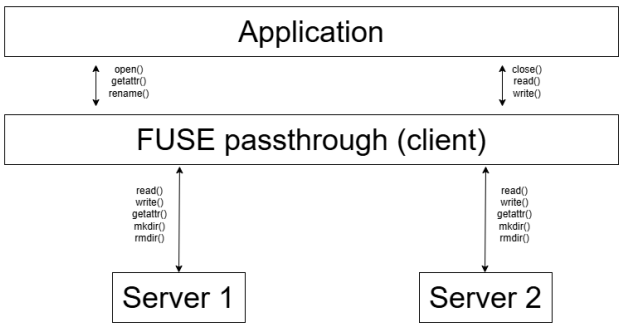


**Figure 1: Arquitetura geral do sistema**

### 2.1 Arquitetura geral

A Figura 1 representa a **arquitetura geral do sistema**. A aplicação interage com o sistema de ficheiros através de chamadas **POSIX standard**, como `open()`, `read()`, `write()` e `close()`, que são interceptadas pelo módulo **FUSE** no espaço do utilizador. Este módulo atua como *cliente*, encaminhando as operações para os *servidores de armazenamento*.

A **comunicação entre o cliente e os servidores** é realizada através do protocolo **TCP**, estabelecendo uma ligação ponto-a-ponto para cada operação. Cada pedido enviado pelo cliente bloqueia a execução até que seja recebida uma resposta correspondente, ou seja, **toda a comunicação é síncrona**. Esta abordagem *simplifica o tratamento da lógica de consistência* e *facilita a deteção de falhas na comunicação*, ainda que introduza alguma *latência* em cenários de *alta concorrência*.

Para garantir **tolerância a faltas**, os dados são **replicados entre múltiplos servidores**. Cada operação de **escrita** (`write()`), **criação de diretórios** (`mkdir()`), **remoção** (`rmdir()`), ou **alteração de atributos** (`getattr()`) é *propagada para todos os servidores*, assegurando que o conteúdo é mantido consistente. Já as operações de **leitura** (`read()`) podem ser satisfeitas a partir de *qualquer servidor disponível*, aumentando assim a **disponibilidade e escalabilidade do sistema**.

Além disso, o **cliente** é responsável por gerir a *seleção do servidor a utilizar*. A estratégia aplicada nesta solução é uma política de seleção do tipo *Round Robin*, adequada ao contexto do protótipo. No entanto, num cenário de produção, *métricas dinâmicas* como a *latência de comunicação* ou a *carga dos servidores* devem ser consideradas para **melhorar a performance global**.

## 2.2 Escrita e leitura por chunks

De modo a tornar a comunicação entre cliente e servidores mais **eficiente e robusta**, as operações de **escrita e leitura** foram desenhadas para funcionar com *chunks de dados de tamanho fixo*, dividindo grandes ficheiros em blocos processados sequencialmente. Isto é consequência da *definição de mensagem* que foi construída, onde o *buffer* apenas tem **4KB**. Todos os pedidos tanto de **escrita** como de **leitura** são transformados em *múltiplos pedidos sucessivos* quando o seu tamanho é superior a 4KB. A escolha deste valor revelou-se adequada para a máquina na qual se testou, no entanto, é algo *passível de ser configurado e modificado* para o uso noutros ambientes. A **divisão em chunks**, embora garanta a *correção do sistema de ficheiros*, **aumenta o overhead em comunicação**, porque aumenta o número de mensagens que são trocadas entre cliente e servidor.

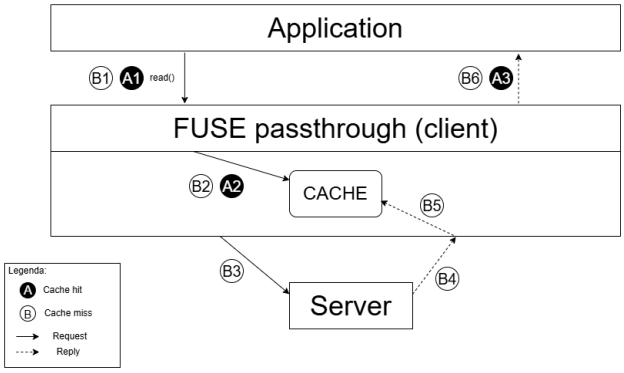


**Figure 2: Travessia de um pedido de leitura**

## 2.3 Otimização de performance - Cache

Para **otimizar o desempenho**, particularmente em **operações de leitura intensiva** — comuns durante o treino de modelos de **Deep Learning** — foi implementado um mecanismo de **cache local** no lado do cliente, como ilustrado na Figura 2.

Este mecanismo atua da seguinte forma:

Quando a aplicação emite uma chamada `read()` (A1), o cliente verifica inicialmente a **cache local** (A2).

Se o conteúdo estiver em **cache** (*cache hit*), os dados são *imediatamente retornados* à aplicação (A3), evitando a *comunicação com os servidores* (B1-B6).

No caso de um *cache miss* (B2), a operação é *encaminhada para um dos servidores* (B3). O conteúdo retornado (B4) é *armazenado em cache* (B5) antes de ser entregue à aplicação (B6).

Este design permite **reduzir significativamente a latência de leitura** em acessos repetidos a ficheiros, além de **diminuir a carga sobre os servidores de armazenamento**.

Este mecanismo traz **benefícios quando o treino envolve múltiplas épocas** e os mesmos ficheiros são acedidos em cada época. **Políticas de *eviction* tradicionais** não são tão adequadas para este cenário. Queremos que pelo menos uma parte do *dataset* seja carregada para a cache e lá permaneça durante as várias épocas. Intuitivamente, chegou-se à conclusão que optar por uma política como a *Least Recently Used* (LRU) iria fazer com que os dados saíssem constantemente da cache antes de serem reutilizados.

Dada a situação, idealizou-se uma política de *eviction* que consistiria em *manter a cache stale até que se detetasse que um novo modelo estivesse a ser treinado*. Após o sistema se aperceber que o modelo que está a ser treinado é novo, simplesmente seria feito um *reset à cache* com o intuito de conseguir guardar uma parte do dataset em cache.

No protótipo, a **cache fica stale** a partir do momento em que atinge o *limite pré-definido*, pelo que, de maneira a simular a política idealizada, é necessário **reiniciar o programa que está a correr o módulo FUSE** para dar o *reset à cache* no momento certo.

## 2.4 Implementação do Protótipo

A **implementação do protótipo** foi realizada em linguagem **C**, recorrendo à biblioteca **FUSE (Filesystem in Userspace)**, que permite desenvolver sistemas de ficheiros no *espaço de utilizador*. O módulo desenvolvido baseia-se no exemplo `passthrough` fornecido pela própria FUSE, *estendendo-o com funcionalidades adicionais*, nomeadamente **comunicação com servidores remotos**, **caching local** e **mecanismos de sincronização**. Os servidores remotos foram simulados na implementação e teste na mesma máquina do cliente. Cada servidor escreve numa pasta dedicada e foram usados apenas 2 servidores. Seguiram-se algumas boas práticas para a extensão para N servidores e não se considera que a solução esteja reduzida a apenas 2 servidores.

Várias **operações do sistema de ficheiros** foram sobrepostas, como `getattr`, `mkdir`, `rmdir`, `read`, `write`, entre outras. No caso da **leitura** (`xmp_read`), é verificado primeiro se os dados já se encontram na **cache**. Caso não estejam, é feito um **pedido remoto aos servidores**, seguido do **armazenamento local dos dados** para reutilizações futuras. A **escrita** (`xmp_write`) *invalida entradas*

*de cache correspondentes*, garantindo **consistência entre cliente e servidor**.

O protótipo implementa, ainda, **operações básicas de sistema de ficheiros**, como **criação e remoção de ficheiros e pastas** (`xmp_create`, `xmp_unlink`, `xmp_mkdir`, `xmp_rmdir`), bem como a **manipulação de permissões**, **ligações simbólicas** e **atributos**. Foram mantidas várias chamadas nativas do sistema operativo, mas sempre que necessário, estas foram *complementadas com chamadas remotas* para **sincronização entre cliente e servidor**.

## 3 Benchmarking

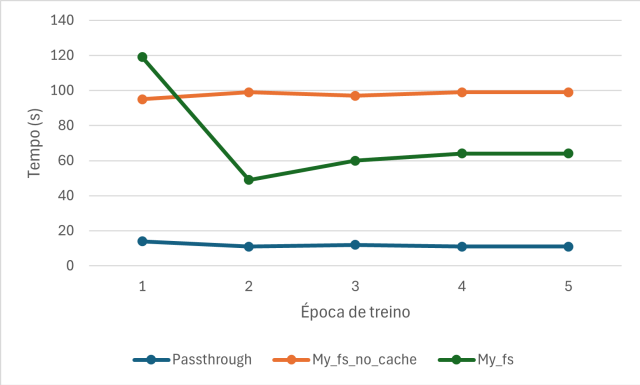


**Figure 3: Tempo de treino por época em segundos**

### 3.1 Objetivos da Avaliação

A avaliação experimental tem como objetivo principal analisar o desempenho do sistema de ficheiros distribuído proposto, com cache (*My_fs*), em comparação com duas alternativas: um sistema de ficheiros tradicional (*Passthrough*) e uma versão sem cache do sistema proposto (*My_fs_no_cache*). As questões orientadoras da avaliação foram:

- O sistema *My_fs* consegue melhorar o *throughput* de treino em relação às abordagens comparadas?
- A utilização da cache permite reduzir significativamente o tempo por época?
- A solução proposta tira partido eficiente dos recursos computacionais disponíveis?

### 3.2 Metodologia de Testes

Os testes foram realizados utilizando a ferramenta `dlio_benchmark`, com o *workload* `resnet50`, que simula um processo de treino de modelos de *deep learning* com características de acesso a dados representativas de cenários reais. Este *workload* emula a leitura de imagens em ficheiros no formato TFRecord e realiza operações computacionais típicas de redes convolucionais profundas, como a ResNet-50.

A configuração experimental foi orientada para isolar o impacto do sistema de ficheiros, minimizando o paralelismo e o número de ficheiros de treino utilizados. Especificamente, foram utilizados apenas 10 ficheiros de treino, com leitura sequencial feita por um único fio (*thread*), e com acesso aos dados fornecido através do sistema de ficheiros alvo.

Foram avaliadas três configurações distintas:

- **Passthrough:** sistema de ficheiros tradicional, sem qualquer camada de cache ou otimização.
- **My_fs_no_cache:** versão do sistema de ficheiros distribuído proposta, mas com a cache desativada.
- **My_fs:** sistema de ficheiros distribuído completo, incluindo a camada de cache de 500Mb implementada para otimização de acessos repetidos.

Cada configuração foi avaliada ao longo de cinco épocas de treino, permitindo observar o impacto acumulado da cache e da estratégia de distribuição dos dados. O principal objetivo desta metodologia foi comparar o desempenho relativo entre as abordagens, em condições controladas e reproduzíveis.

As medições incluíram métricas como: taxa de utilização da unidade aceleradora (`train_au_percentage`), *throughput* de treino em amostras por segundo (`train_throughput_samples_per_second`) e I/O médio em MB/s (`train_io_mean_MB_per_second`). As medições foram feitas ao longo de 5 épocas, em três configurações distintas: `Passthrough`, `My_fs_no_cache` e `My_fs`.

O sistema de testes era composto por um processador **Intel 12th Gen Core i7-1255U** com **2 núcleos físicos**, **12 MB de cache** e aproximadamente **2 GB de memória RAM** disponível. Nenhuma GPU real foi utilizada; o acelerador foi simulado para efeitos de benchmark.

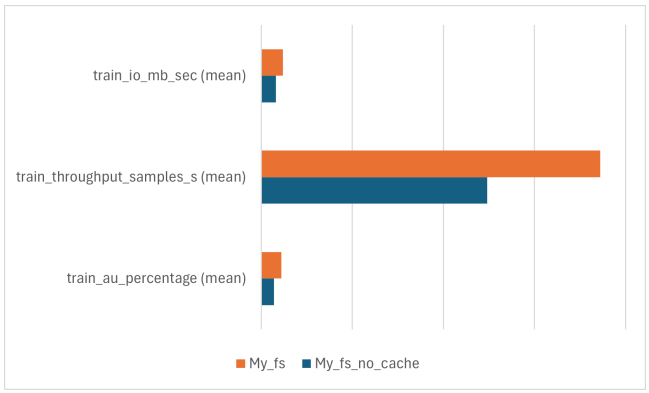


**Figure 4: Variadas métricas de eficiência no treino**

### 3.3 Apresentação e Discussão dos Resultados

Os resultados obtidos evidenciam o impacto positivo da introdução de cache no sistema distribuído `My_fs`, embora o desempenho global do sistema `Passthrough`, que representa um sistema de ficheiros local tradicional, tenha sido consistentemente superior em todas as métricas avaliadas.

- **Taxa de utilização da aceleradora:** O sistema `Passthrough` obteve a melhor taxa de utilização da GPU, demonstrando um pipeline de treino mais eficiente e com menor tempo de espera por dados. Ainda assim, a introdução de cache em `My_fs` permitiu melhorar a utilização face à versão sem

cache (`My_fs_no_cache`), indicando que a cache reduz o impacto da latência de acesso em sistemas distribuídos.

- **Throughput de treino:** Em termos de amostras processadas por segundo, o sistema `Passthrough` alcançou o throughput mais elevado. Contudo, a comparação entre `My_fs` e `My_fs_no_cache` mostra um ganho de aproximadamente 50% com a introdução da cache, refletindo uma melhoria significativa dentro do contexto do sistema distribuído.
- **Taxa de I/O:** O sistema local `Passthrough` apresentou uma taxa de leitura média muito superior, como seria expectável devido à ausência de camadas adicionais de abstração e comunicação. A cache em `My_fs` permitiu, ainda assim, aumentar em 50% a taxa de I/O face à versão sem cache, validando o seu impacto positivo na performance de leitura em cenários distribuídos.
- **Tempo por época:** A solução `Passthrough` foi claramente a mais rápida, com tempos de execução significativamente inferiores aos restantes sistemas. A comparação entre `My_fs` e `My_fs_no_cache` revela que a cache reduz o tempo médio por época de forma consistente e a Figura 3 mostra um padrão que é explicável. O treino é mais lento na versão com cache na primeira época, visto que a cache está vazia e tem o *overhead* de ter que ser preenchida à medida que as leituras vão sendo feitas. Nas seguintes épocas, a cache já tem a capacidade de responder a parte dos pedidos, pelo que se pode notar uma redução de 50% do tempo da primeira época. Nas outras configurações não são visíveis grandes variações de tempo entre épocas.

As Figuras 3 e 4 ilustram graficamente estas comparações. A Figura 4 apresenta apenas as configurações `My_fs` e `My_fs_no_cache`, uma vez que a superioridade de `Passthrough` comprometeria a escala visual dos restantes. No entanto, a sua exclusão gráfica não deve ser interpretada como inferioridade, mas sim como uma escolha de apresentação focada na análise da evolução interna do sistema distribuído.

Importa reforçar que o objetivo da avaliação não foi maximizar o desempenho absoluto, mas sim validar os ganhos proporcionados pela introdução de uma camada de cache em sistemas de ficheiros distribuídos. Neste contexto, os resultados mostram que a cache tem um papel relevante na aproximação do desempenho a soluções locais, mantendo a escalabilidade e flexibilidade típicas de sistemas distribuídos.

## 3.4 Conclusão da Avaliação

A avaliação experimental confirmou que a inclusão de uma camada de cache no sistema de ficheiros distribuído (`My_fs`) traz benefícios mensuráveis em termos de utilização do sistema, tempo de treino e taxa de leitura. No entanto, ambas as soluções distribuídas ficam muito abaixo das métricas do sistema tradicional (*Passthrough*). Embora o ambiente de teste seja limitado em termos de hardware, os resultados indicam que a abordagem proposta é eficaz e pode ser particularmente vantajosa em contextos com acesso a recursos computacionais mais potentes.

# 4 Trabalho Relacionado

A necessidade de sistemas de armazenamento otimizados para cargas de trabalho de Deep Learning tem sido amplamente reconhecida no mundo da investigação e dos artigos científicos. Vários trabalhos exploram soluções distribuídas com tolerância a faltas e otimizações de I/O, centrando-se na minimização da latência e maximização do throughput durante o treino de modelos.

Soluções como o Ceph e o HDFS têm sido empregues em ambientes de Machine Learning distribuído, oferecendo escalabilidade e replicação de dados, embora nem sempre garantam desempenho adequado para workloads com forte componente de leitura aleatória, como os de treino com datasets de imagens. Em [3], é discutida a importância do formato e acesso eficiente aos dados em pipelines de treino com TensorFlow, o que motiva soluções mais especializadas.

Do lado académico, o trabalho de Gao et al. (2020) com Petrel [2] propõe um sistema de armazenamento distribuído para AI que combina cache agressiva e interfaces otimizadas para workloads de leitura intensiva. Outros projetos, como o SAND [4], focam-se em integração de cache multinível, mostrando melhorias expressivas no tempo de treino em clusters.

No contexto da camada de sistema de ficheiros, FUSE tem sido utilizada como plataforma de prototipagem para novas abordagens de armazenamento, dada a sua flexibilidade e compatibilidade POSIX, conforme explorado em trabalhos como [1].

O presente trabalho distingue-se por combinar uma abordagem leve baseada em FUSE com um mecanismo de cache local e replicação simplificada, orientado especificamente para ambientes com recursos limitados e sem dependência de infraestrutura de larga escala.

# 5 Conclusão e Trabalho Futuro

Este trabalho apresentou o desenvolvimento de um sistema de ficheiros distribuído, compatível com a interface POSIX e orientado para cargas de trabalho típicas de treino de modelos de Deep Learning. A solução proposta, implementada sobre a biblioteca FUSE, integra dois mecanismos centrais: a replicação de dados, para garantir tolerância a faltas, e uma cache local no cliente, para otimizar o desempenho em leituras repetidas.

A avaliação experimental demonstrou que a introdução de uma camada de cache permite ganhos significativos no tempo de treino e na eficiência do sistema, aproximando o desempenho da solução distribuída a sistemas locais em cenários com reutilização intensiva de dados. Embora o sistema `Passthrough` tenha superado o sistema distribuído em termos absolutos, os resultados validam as decisões de design tomadas e evidenciam o potencial da solução para evoluir em ambientes reais com maior capacidade computacional.

## 5.1 Trabalho Futuro

Apesar dos resultados promissores, existem várias direções interessantes para evolução futura deste trabalho:

- **Política de cache adaptativa:** A política atual de cache é rudimentar e depende de reinícios manuais para simular comportamentos desejados. Uma evolução natural passa pela implementação de uma política de substituição adaptativa, sensível ao padrão de acesso e à deteção automática de novos ciclos de treino.

- **Replicação inteligente:** A replicação implementada atualmente é síncrona e uniforme para todos os servidores. Poderá ser vantajoso explorar estratégias de replicação assíncrona, diferencial ou com codificação de redundância (erasure coding), permitindo ganhos em latência e uso eficiente do espaço de armazenamento.
- **Gestão de falhas mais robusta:** A tolerância a faltas pode ser aprofundada com mecanismos mais completos de deteção de falhas, recuperação automática de nós e reconfiguração dinâmica do conjunto de servidores.
- **Integração com métricas de rede:** A seleção do servidor é feita atualmente por *round robin*, ignorando variações de latência ou carga. Uma extensão relevante seria o uso de métricas dinâmicas para selecionar, em tempo real, o servidor mais responsivo, otimizando o tempo de acesso.
- **Suporte para paralelismo e múltiplos clientes:** O protótipo foi testado em cenários com um único cliente e acesso sequencial. A avaliação futura deverá considerar múltiplos clientes concorrentes e acessos paralelos, replicando mais fielmente ambientes reais de treino distribuído.

Acredita-se que, com estas evoluções, o sistema proposto poderá tornar-se uma alternativa leve e eficaz para ambientes de treino de modelos de AI, oferecendo um equilíbrio entre desempenho, tolerância a faltas e simplicidade de integração.

## References


[1] John R. Douceur, William J. Bolosky, et al. 2011. System Benchmarks for User-Space File Systems using FUSE. In *Proceedings of the USENIX Annual Technical Conference*. 1–12.

[2] Yifeng Gao, Chengcheng Xu, Xiaoyi Shi, et al. 2020. Petrel: A distributed file system for training deep learning models. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2020), 113–129.

[3] TensorFlow Team. 2022. Data Input Pipeline Performance Guide. https://www.tensorflow.org/guide/data_performance. Accessed: 2025-06-02.

[4] Yu Zhang, Qi Lu, Song Jiang, et al. 2021. SAND: Towards High-Performance Deep Learning using Multilevel I/O Caching. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2021), 1–14.


## A Guia de Benchmarking

Este anexo descreve os passos seguidos para realizar o benchmarking do sistema de ficheiros desenvolvido, utilizando a *MLPerf™ Storage Benchmark Suite*.

### A.1 Objetivo

O principal objetivo deste guia é configurar uma framework de benchmark, compreender os workloads utilizados e aplicá-los para avaliar o desempenho de diferentes implementações de sistemas de ficheiros. Em particular, foi utilizada a MLPerf™ Storage Benchmark Suite para caracterizar o desempenho do sistema de ficheiros FUSE perante workloads de machine learning.

O código do benchmark está disponível em: https://github.com/mlcommons/storage

Documentação adicional pode ser consultada em: https://dlio-benchmark.readthedocs.io/en/latest/#

## B Setup

(1) Preparar um ambiente de execução adequado e garantir a disponibilidade de uma implementação do sistema de ficheiros FUSE compatível com o benchmark.

(2) Instalar os pacotes necessários:

```
sudo apt-get install mpich
sudo apt install python3-pip
sudo apt install python3-venv
```

(3) Clonar o repositório do benchmark e atualizar o repositório `dlio_benchmark` para a versão mais recente:

```
git clone -b v1.0 --recurse-submodules https://github.com/mlc
cd storage/dlio_benchmark/
git checkout main
```

(4) Criar e ativar um ambiente virtual Python, compilar o benchmark e regressar à pasta raiz:

```
python3 -m venv myenv
source myenv/bin/activate
pip install .[dlio_profiler]
cd ..
```

## C Execução

Foi utilizado um workload que simula operações de I/O de armazenamento realizadas pelo TensorFlow durante o treino do modelo ResNet50 com uma GPU NVIDIA H100. O ficheiro de configuração usado foi `resnet50_h100.yaml`, localizado na pasta `storageconf/workload/`.

(1) Criar o dataset no ponto de montagem do sistema de ficheiros FUSE (e.g., `/mnt/fs`). Foram criados 10 ficheiros `tfrecord` de treino:

```
./benchmark.sh datagen --hosts 127.0.0.1 --workload resnet50
--num-parallel 1 --param dataset.num_files_train=10 \
--param dataset.data_folder=/mnt/fs/resnet_data
```

(2) Executar o workload de treino sobre o dataset, com um único *reader thread*:

```
./benchmark.sh run --hosts 127.0.0.1 --workload resnet50 --ac
--num-accelerators 1 --results-dir /mnt/fs/resnet_h100 \
--param dataset.num_files_train=10 \
--param dataset.data_folder=/mnt/fs/resnet_data \
--param reader.read_threads=1
```

(3) Avaliar os dados da pasta selecionada (–results-dir).