Máster en Inteligencia Artificial UMU

Infraestructuras para Deep Learning

Práctica 1: Programación de plataformas de cómputo para DL

1 Objetivos

1. Aprendizaje de librerías y frameworks modernos para aceleración de cargas de trabajo de Deep Learning en CPU y aceleradores.
2. Implementación y evaluación de técnicas de programación paralela para realizar procesos de inferencia y entrenamiento de modelos de Deep Learning.

2 Entornos disponibles

De forma similar a como se ha procedido con otras asignaturas del máster, para poder realizar esta práctica se utilizará fundamentalmente el clúster ATLAS de la UMU. No obstante, para aquellas tareas que requieran del uso de una CPU y/o de una GPU, es decir, no requieran de un entorno multi-GPU, se podría utilizar la máquina del estudiante.

2.1 Máquina del estudiante

En este caso, se tendría que disponer de una máquina con Linux y tener instalado el sistema de virtualización Docker. Además, para aquellos que dispongan de GPU propia, tendrá que ser del fabricante NVIDIA.

En la carpeta de Recursos/Práctica1 del Aula Virtual hay un fichero Dockerfile para poder realizar la construcción de la imagen y poder crear el contenedor. Es importante tener en cuenta que tras construir la imagen con el comando:

```
[mi_maquina]$ sudo docker build -f Dockerfile -t accelerate:idl .
```

La imagen creada ocupará unos 30 GBytes de espacio en disco y, dependiendo de la potencia de la máquina, su creación puede requerir de bastantes minutos (una media hora si la máquina no es muy potente).

Tras crear la imagen, para poder lanzar el contenedor de manera interactiva se ejecutará el comando:

```
[mi_maquina]$ sudo docker container run --gpus all -it ID_IMAGEN
```

Donde ID_IMAGEN será el identificador de la imagen previamente construida con el Dockerfile y cuyo identificador se podrá consultar con el comando:

```
[mi_maquina]$ sudo docker images
```

Es recomendable entrar de manera interactiva al contenedor para poder testear el entorno de ejecución y su correcto funcionamiento.

2.2 Clúster ATLAS

Tal y como pudisteis trabajar en el primer cuatrimestre con otras asignaturas, esta solución está basada en el uso de servidores dinámicos proporcionados por el clúster ATLAS de la Universidad de Murcia. La novedad es el poder acceder a la cola03 que tiene nodos multi-GPU. Por tanto, en el enunciado de esta práctica no se repetirá cómo tenéis que acceder al clúster para lanzar experimentos. De hecho, la configuración correcta de los scripts necesarios para la realización de los experimentos de la práctica para envío de trabajos no interactivos en los que SLURM se encarga de procesarlos, formará parte de los entregables de esta práctica. Una guía del uso del clúster ser puede consultar aquí.

Lo que sí es necesario comentar es para qué se usará cada una de las colas:

- **Cola01**: Recomendable para cómputo con solo CPU.
- **Cola02**: Necesario para computar con GPU.
- **Cola03**: Necesario para computar con nodos multi-GPU.

En lugar de imágenes Docker, en el clúster las imágenes se gestionarán a través de *Apptainer* (antiguo *Singularity*). En esta asignatura, todas las imágenes ".sif" de los contenedores Apptainer que sean necesarios para llevar a cabo las tareas de esta práctica, se encontrarán disponibles en el clúster en la carpeta con ruta absoluta: `/software/singularity/Informatica/mia-idl-apptainer/`.

Todos podréis acceder a esta carpeta tras conectarnos al clúster mediante el comando siguiente (donde sustituiremos `usuario` por nuestro *login* correspondiente):

```
[mi_maquina]$ ssh usuario@atlas-scc.atica.um.es
```

En esta carpeta existe una imagen genérica que contiene todo el software necesario para ejecutar experimentos con el software Accelerate de Hugging Face, que será descrito más abajo. Esta imagen se llama "mia_idl_1.0.sif". No obstante, excepcionalmente se podrá permitir ampliar y/o crear una nueva imagen de Apptainer a petición del grupo de prácticas en caso de que se quiera explorar funcionalidad no incluida en esta imagen genérica. En este caso, se tendrá que comprobar con el profesor su viabilidad y, en caso de serlo, el grupo de prácticas proporcionará un fichero Dockerfile con los requisitos de instalación al profesor mediante un mensaje a través del aula virtual. Este Dockerfile servirá de base para construir la imagen Apptainer que finalmente estará disponible en la carpeta mencionada anteriormente para su uso por parte del grupo de prácticas que lo haya solicitado.

3 Accelerate

Accelerate es una librería de Hugging Face sobre PyTorch que permite adaptar fácilmente procedimientos de entrenamiento e inferencia de nuestros modelos de Deep Learning, para que se ejecuten más rápidamente y de manera eficiente aprovechando las características hardware de plataformas de cómputo tanto en entornos centralizados (e.g., uno solo nodo de cómputo con GPU) como distribuidos (e.g., entorno multi-GPU); un tutorial de uso de Accelerate se puede encontrar aquí. Así, partiendo del siguiente ejemplo de código PyTorch para entrenar un modelo de red neuronal arbitrario ("model") sobre GPU:

```
# 1. Mover el modelo a la GPU para acelerar el entrenamiento
device = "cuda"
model.to(device)
# 2. Iterar sobre los lotes de datos del DataLoader
for batch in training_dataloader:
    # 3. Reiniciar los gradientes del optimizador (se acumulan por defecto en PyTorch)
    optimizer.zero_grad()
    # 4. Obtener los datos de entrada y sus etiquetas
    inputs, targets = batch
    # 5. Mover los datos a la GPU para que coincidan con el dispositivo del modelo
    inputs = inputs.to(device)
    targets = targets.to(device)
    # 6. Hacer la predicción con el modelo
    outputs = model(inputs)
    # 7. Calcular la pérdida comparando la predicción con los valores reales
    loss = loss_function(outputs, targets)
    # 8. Retropropagar la pérdida para calcular los gradientes
    loss.backward()
    # 9. Actualizar los parámetros del modelo usando el optimizador
    optimizer.step()
    # 10. Ajustar la tasa de aprendizaje si se usa un scheduler
    scheduler.step()
```

Con Accelerate el código anterior sería modificado delegando a la clase **Accelerator** la selección y uso de un dispositivo acelerador (e.g., la GPU), así como para manejar la transferencia de objetos PyTorch (model, optimizer, scheduler, etc.) a éste.

```
from accelerate import Accelerator

# Inicializa Accelerator, que gestiona automáticamente el dispositivo (CPU, GPU, TPU)
accelerator = Accelerator()

device = accelerator.device  # No es necesario especificar manualmente "cuda()" o "cpu()"

# Prepara el modelo, optimizador, dataloader y scheduler para entrenamiento en múltiples dispositivos
# Esto reemplaza la necesidad de mover manualmente estos elementos a un dispositivo específico
model, optimizer, training_dataloader, scheduler = accelerator.prepare(
    model, optimizer, training_dataloader, scheduler
)

for batch in training_dataloader:
    optimizer.zero_grad()
    inputs, targets = batch

    # No es necesario mover nada al dispositivo acelerador ya que el nuevo training_dataloader lo hará
    #inputs = inputs.to(device)
    #targets = targets.to(device)

    outputs = model(inputs)
    loss = loss_function(outputs, targets)

    # Accelerate maneja la sincronización automática del gradiente si entorno distribuido
    accelerator.backward(loss)

    optimizer.step()
    scheduler.step()
```

A continuación, lo único que se tendría que hacer para poder lanzar el proceso de entrenamiento sobre GPU sería ejecutar el comando:

```
accelerate launch {my_script.py}
```

Donde {my_script.py}, sería el script de PyTorch donde estaría el código de la imagen anterior.

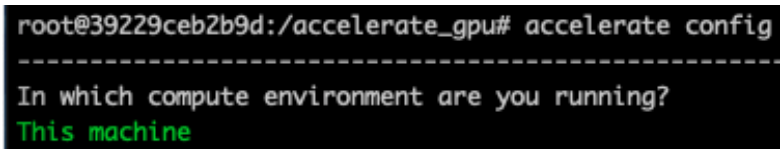
Para que Accelerate conozca el entorno de ejecución del script anterior, aunque en el script PyTorch se podría dar un argumento precisando el entorno de ejecución GPU cuando se crea el objeto "Accelerator" cambiando la línea 4 de la imagen anterior por:

```
Accelerator= Accelerator(cpu=False)
```

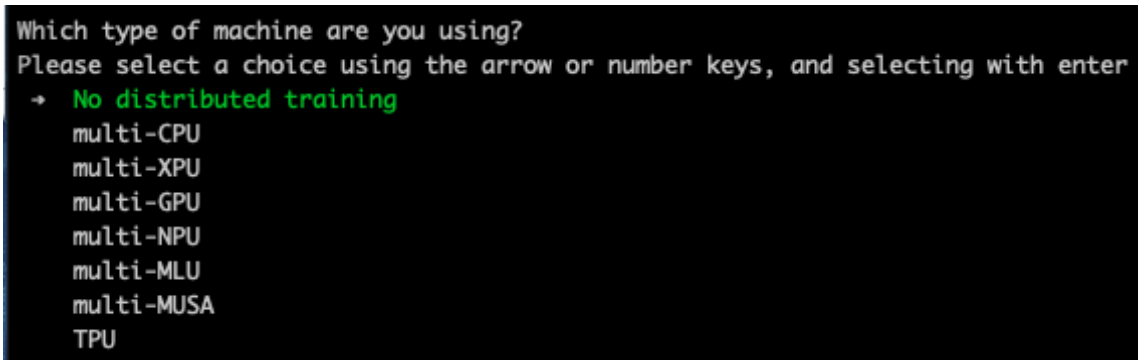
Accelerate ofrece un comando de configuración para definirlo:

```
accelerate config
```

Tras ejecutar el comando anterior, aparecen varios entornos de ejecución soportados actualmente por Accelerate y tendremos que ir seleccionando opciones de acuerdo al hardware del sistema de cómputo donde queramos ejecutar el entrenamiento o inferencia:



Estos entornos de ejecución posibles son los siguientes:



1. ***No distributed training***: Configura el entorno para ejecutar entrenamiento en un solo dispositivo sin distribución (seleccionar un nodo con CPU o GPU)
2. ***multi-CPU***: Habilita el entrenamiento distribuido utilizando múltiples CPU en paralelo.
3. ***multi-XPU***: Configura el entrenamiento distribuido en múltiples aceleradores compatibles con Intel XPU (arquitectura de cómputo heterogénea que combina CPU, GPU y FPGA)

4. ***multi-GPU***: Permite la distribución del entrenamiento en varias GPU para mejorar el rendimiento.
5. ***multi-NPU***: Configura el entrenamiento distribuido en múltiples NPU (*Neural Processing Units)*.
6. ***multi-MLU***: Habilita el entrenamiento en múltiples MLU (Cambricon Machine Learning Units; aceleradores creados para DL por la empresa Cambricon).
7. ***multi-MUSA***: Configura el entrenamiento distribuido en múltiples aceleradores MUSA (Mthreads Unified System Architecture; aceleradores creados para DL por la empresa Mthreads).
8. ***TPU***: Permite el uso de Tensor Processing Units (TPU) para el entrenamiento distribuido de modelos de machine learning.

En nuestro caso, podremos seleccionar las opciones 1 (permite distinguir entre CPU o GPU) y la 4 (para la cola03). Para testear qué configuración podríamos tener disponible para seleccionar, usaremos el comando:

```
accelerate env
```

Además, cuando se selecciona una de las opciones anteriores, Accelerate da la opción de seleccionar frameworks/librerías para optimizar la ejecución como son IPEX, Torch Dynamo, DeepSpeed, *mixed precision*, entre otras que serán descritas más abajo.

También tiene la opción para uso de Amazon SageMaker, servicio de Amazon Web Services (AWS) para entrenamiento e inferencia de modelos DL escalable y automatizado en la nube (un servicio de *cloud computing*).

En entornos clusterizados, como es el del clúster ATLAS gestionado por SLURM, se recomienda crear ficheros de configuración ".yaml" y seleccionarlos cuando se vaya a ejecutar el proceso de entrenamiento sobre un sistema de cómputo específico.

Por ejemplo, si queremos ejecutar el procesamiento de entrenamiento anterior en GPU, crearíamos primero un fichero ".yaml" con la configuración para ejecutar sobre GPU:

```
root@f0dae4912802:/accelerate_gpu# accelerate config --config_file config_gpu.yaml
----------------------------------------------------------------------------------------------------------------------------------------
In which compute environment are you running?
This machine
----------------------------------------------------------------------------------------------------------------------------------------
Which type of machine are you using?
No distributed training
Do you want to run your training on CPU only (even if a GPU / Apple Silicon / Ascend NPU device is available)? [yes/NO]:NO
Do you wish to optimize your script with torch dynamo?[yes/NO]:NO
Do you want to use DeepSpeed? [yes/NO]: NO
What GPU(s) (by id) should be used for training on this machine as a comma-seperated list? [all]:all
Would you like to enable numa efficiency? (Currently only supported on NVIDIA hardware). [yes/NO]: NO
----------------------------------------------------------------------------------------------------------------------------------------
Do you wish to use mixed precision?
no
accelerate configuration saved at config_gpu.yaml
```

Cuyo contenido sería el siguiente:

```
root@f0dae4912802:/accelerate_gpu# cat config_gpu.yaml
compute_environment: LOCAL_MACHINE
debug: false
distributed_type: 'NO'
downcast_bf16: 'no'
enable_cpu_affinity: false
gpu_ids: all
machine_rank: 0
main_training_function: main
mixed_precision: 'no'
num_machines: 1
num_processes: 1
rdzv_backend: static
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: false
```

Y después lanzar nuestro experimento de entrenamiento de la siguiente forma:

```
$ accelerate launch --config_file config_gpu.yaml ScriptEntrenamiento.py
```

3.1 Ejecución de inferencia y entrenamiento de modelos DL

El proceso de ejecución de la inferencia y entrenamiento de modelos DL definidos en PyTorch es bastante sencillo. A modo de ejemplo, vamos a utilizar un modelo del dominio de aplicación de visión por computador para clasificación de imágenes como es el modelo convolucional ResNet-50. El procedimiento de inferencia para ResNet-50, sería el siguiente (disponible en Aula Virtual con nombre resnet50-inf_cpu.py):

```python
from accelerate import Accelerator, ProfileKwargs
import torch
import torchvision.models as models
import torchvision.transforms as transforms

# Load ResNet-50 model
model = models.resnet50(pretrained=True)
model.eval()  # Set model to evaluation mode

# Define image transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # ResNet expects 224x224 images
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Create a batch of random images (batch size 128, 3 color channels, 224x224 resolution)
batch_size = 128
input_images = torch.rand((batch_size, 3, 224, 224))  # Random image batch

# Define profiling kwargs for CPU activities
profile_kwargs = ProfileKwargs(
    activities=["cpu"],  # Profile CPU activities instead of CUDA
    record_shapes=True
)
```

```
26
27 # Initialize the accelerator for CPU
28 accelerator = Accelerator(cpu=True, kwargs_handlers=[profile_kwargs])
29
30 # Prepare the model for CPU execution
31 model = accelerator.prepare(model)
32
33 # Move inputs to CPU
34 device = accelerator.device
35 input_images = input_images.to(device)
36
37 # Profile the model execution on the CPU
38 with accelerator.profile() as prof:
39     with torch.no_grad():
40         outputs = model(input_images)  # Forward pass
41
42 # Print profiling results
43 print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

Como puede observarse, se utilizará un modelo ResNet-50 pre-entrenado y un ejemplo de entrada ficticia con un *batch* de 128 imágenes. En la segunda imagen, se crea una instancia de Accelerator que se utilizará para llevar a cabo la predicción del batch. Finalmente, imprimimos el tiempo total que ha necesitado la CPU para realizar el procedimiento de inferencia.

Para poder lanzar este procedimiento de inferencia, ejecutaremos el siguiente comando, asumiendo que tenemos ya configurado en el fichero ".yaml" el entorno para sistema de cómputo CPU:

```
root@39229ceb2b9d:/accelerate_gpu# accelerate launch --config_file config_cpubase.yaml bert-inf_cpu.py
2025-02-25 14:50:16,590 - launch.py - accelerate.commands.launch - WARNING - The following values were not passed to `accelerate launch` a
        `--num_cpu_threads_per_process` was set to `8` to improve out-of-box performance when training on CPUs
To avoid this warning pass in values for each of the problematic parameters or run `accelerate config`.
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------  ------------
                                                   Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg    # of Calls
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------  ------------
                                       aten::linear         0.32%     184.146ms       182.69%     104.873s     718.307ms           146
                                       aten::addmm         88.51%      50.808s        89.45%      51.351s     703.434ms            73
                                       aten::copy_          4.84%       2.777s         4.84%       2.777s       9.412ms           295
                                          aten::to          0.00%     536.520us         3.90%       2.239s      10.177ms           220
                                    aten::_to_copy          0.00%       2.217ms         3.90%       2.238s      10.175ms           220
                 aten::scaled_dot_product_attention          0.02%       8.697ms         3.89%       2.232s     186.015ms            12
    aten::_scaled_dot_product_flash_attention_for_cpu          3.87%       2.223s         3.87%       2.223s     185.290ms            12
                                         aten::add          1.05%     604.371ms         1.50%     863.042ms      34.522ms            25
                                        aten::gelu          0.65%     371.463ms         0.65%     371.463ms      30.955ms            12
                                  aten::layer_norm          0.00%     222.897us         0.59%     340.061ms      13.602ms            25
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 57.406s
```

Como puede observarse este procedimiento ha tardado 57,4 segundos en completarse.

En la carpeta de Recursos/Practica1 del aula virtual se tiene disponible el código para lanzar este experimento, también para entrenamiento, así como otros ejemplos para entrenamiento e inferencia con el modelo BERT-base-cased. BERT-base-cased es un modelo de procesamiento de lenguaje natural pre-entrenado por Google que utiliza la arquitectura Transformer para comprender el significado de textos en inglés, respetando mayúsculas y minúsculas.

3.2 Optimizaciones de ejecución: GPU

La GPU está siendo masivamente explotada como acelerador para procesamiento DL. Accelerate nos permite con ligeras modificaciones sobre el código PyTorch anterior poder ejecutar procedimientos de inferencia y entrenamiento de modelos DL. Para demostrar esto, vamos a ejecutar el mismo experimento anterior, pero sobre GPU. Para ello, tendremos que configurar Accelerate para que utilice el sistema de cómputo GPU, siempre y cuando la máquina sobre la que vamos a realizar el experimento disponga de GPU.

Para ello, utilizaremos el script bert-inf_gpu.py disponible en la carpeta de Recursos/Practica1 del Aula Virtual (inferencia con el modelo Bert-base-cased) y el fichero de configuración ".yaml" config_gpubase.yaml" para delegar el cómputo a la GPU.

```
root@39229ceb2b9d:/accelerate_gpu# accelerate launch --config_file config_gpubase.yaml bert-inf_gpu.py
-------------------------------------------------------  ------------  ------------  ------------  ---------
                                                   Name    Self CPU %      Self CPU   CPU total %     CPU to
-------------------------------------------------------  ------------  ------------  ------------  ---------
ampere_bf16_s16816gemm_bf16_256x128_ldg8_relu_f2f_st...        0.00%       0.000us         0.00%       0.00
void cutlass::Kernel2<cutlass_80_tensorop_bf16_s1681...        0.00%       0.000us         0.00%       0.00
void pytorch_flash::flash_fwd_kernel<pytorch_flash::...        0.00%       0.000us         0.00%       0.00
void at::native::vectorized_elementwise_kernel<4, at...        0.00%       0.000us         0.00%       0.00
void at::native::(anonymous namespace)::vectorized_l...        0.00%       0.000us         0.00%       0.00
void at::native::unrolled_elementwise_kernel<at::nat...        0.00%       0.000us         0.00%       0.00
void at::native::vectorized_elementwise_kernel<4, at...        0.00%       0.000us         0.00%       0.00
void at::native::(anonymous namespace)::indexSelectL...        0.00%       0.000us         0.00%       0.00
void at::native::vectorized_elementwise_kernel<4, at...        0.00%       0.000us         0.00%       0.00
void at::native::elementwise_kernel<128, 2, at::nati...        0.00%       0.000us         0.00%       0.00
-------------------------------------------------------  ------------  ------------  ------------  ---------
Self CPU time total: 948.454ms
Self CUDA time total: 635.750ms
```

Como podemos observar, en este caso el tiempo en realizarse el procedimiento de inferencia ha sido de menos de un segundo, o 57 veces más rápido que con la CPU. Nótese que, "Self CPU time" y "CUDA time total", son los tiempos que cada dispositivo ha empleado en realizar el experimento. Aquí, la CPU se ha utilizado para unas operaciones que involucran fundamentalmente inicialización de parámetros de ejecución, los tensores de los modelos DL, los pesos y activaciones, así como los ejemplos de entrada que constituyen el batch. La GPU, por el contrario, se ha encargado de realizar el cómputo de la inferencia del modelo. Estos dos tiempos no han de sumarse para saber el tiempo total de ejecución, sino que se tiene que utilizar alguna herramienta externa para medir el tiempo que el usuario ha estado esperando a que se complete el experimento. Para ello, podemos utilizar el comando Linux `time`.

Esta herramienta nos ofrece como salida:

- **real**: Tiempo total desde que se ejecutó el comando hasta que finalizó (incluye espera por I/O y otros procesos).
- **user**: Tiempo total de CPU usado por el proceso en modo usuario. Si se tienen varios hilos/cores sobre los que se ha parelizado el experimento, user puede dar un valor que no coincide con el tiempo real de ejecución. User suma los tiempos individuales que cada hilo termina en ejecutarse.
- **sys**: Tiempo total de CPU usado en llamadas al sistema operativo.

Por ejemplo, se puede ver en la siguiente imagen el tiempo real que tarda realmente el experimento sobre CPU (sobre 63 segundos):

```
real    1m3.481s
user    7m16.946s
sys     0m20.592s
root@39229ceb2b9d:/accelerate_gpu# time accelerate launch --config_file config_cpubase.yaml bert-inf_cpu.py
```

Y sobre GPU (sobre 7 segundos):

```
real    0m7.792s
user    0m8.658s
sys     0m0.804s
root@39229ceb2b9d:/accelerate_gpu# time accelerate launch --config_file config_gpubase.yaml bert-inf_gpu.py
```

Considerando el valor de la métrica real, ahora se aprecia como el experimento se ha acelerado 9 veces.

Para excluir el tiempo que invierte la CPU en inicialización de tensores, preparación del batch, y configuración del entorno Accelerate, para centrares en medir solo el tiempo de ejecución de una sección específica dentro del código, como puede ser la inferencia del modelo o justo la sección de código para el entrenamiento, es mejor usar time.time() en Python o torch.cuda.Event para mediciones precisas en GPU.

3.3 Optimizaciones de ejecución: IPEX

Intel Extension for PyTorch (IPEX) está optimizado para CPUs con AVX-512 o superior, aunque también funciona en CPUs con solo AVX2. Se espera que mejore el rendimiento en procesadores Intel con AVX-512 o superior, mientras que en CPUs con solo AVX2 (como algunos AMD o Intel más antiguos) podría ofrecer beneficios, pero sin garantía.

IPEX optimiza el entrenamiento en CPU con precisión *Float32* y *BFloat16*. En lo que sigue adoptaremos *BFloat16* (**bf16** en Accelerate) como ejemplo. El tipo de dato de baja precisión *BFloat16* es compatible de forma nativa con los procesadores *Xeon® Scalable* de 3ª generación (*Cooper Lake*) con AVX-512 y será compatible con la próxima generación de procesadores *Intel® Xeon® Scalable* con instrucciones *Intel® AMX*, ofreciendo un mayor rendimiento.

Además, podemos configurar en Accelerate, Auto Mixed Precision (AMP), que es una técnica que permite entrenar modelos con una combinación de precisión de punto flotante (FP16 y FP32) de forma automática. Esto reduce el uso de memoria y acelera el entrenamiento sin afectar significativamente la precisión del modelo. AMP selecciona dinámicamente qué operaciones pueden ejecutarse en FP16 para mayor eficiencia y cuáles deben mantenerse en FP32 para estabilidad numérica.

Para poder lanzar experimentos con CPU e IPEX, tendremos que configurar en Accelerate las siguientes opciones:

```
root@39229ceb2b9d:/accelerate_gpu# cat config_ipexbase.yaml
compute_environment: LOCAL_MACHINE
debug: false
distributed_type: 'NO'
downcast_bf16: 'no'
enable_cpu_affinity: false
ipex_config:
  ipex: true
machine_rank: 0
main_training_function: main
mixed_precision: bf16
num_machines: 1
num_processes: 1
rdzv_backend: static
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: true
```

3.4 Profiling de la ejecución

El *profiling* en general es el proceso de analizar el rendimiento de un programa para identificar cuellos de botella, optimizar el uso de recursos y mejorar la eficiencia. Se centra en medir aspectos como el tiempo de ejecución, el uso de CPU y GPU, el consumo de memoria y la frecuencia de llamadas a funciones. En el contexto de Deep Learning, el profiling ayuda a detectar operaciones costosas y mejorar la velocidad de entrenamiento e inferencia de modelos.

ProfileKwargs es una utilidad de Accelerate que permite configurar el profiling del rendimiento de un modelo en PyTorch. Los tipos de métricas que puede reportar son:

- **cpu_time_total**: Tiempo total en CPU
- **cuda_time_total**: Tiempo total en GPU
- **self_cpu_time_total**: Tiempo en CPU sin incluir suboperaciones
- **self_cuda_time_total**: Tiempo en GPU sin incluir suboperaciones
- **self_cpu_memory_usage**: Uso de memoria en CPU
- **self_cuda_memory_usage**: Uso de memoria en GPU
- **flops**: Operaciones de punto flotante por segundo
- **record_shapes**: Formas de los tensores usados en cada operación

Es importante destacar que, a diferencia de en otras asignaturas, nos centramos más en métricas no funcionales como el tiempo de ejecución y consumo de memoria, más que en el *accuracy* del modelo entrenado. Así, para inferencia, analizaremos el número de operaciones que han sido ejecutadas, el tamaño del modelo, el tiempo de ejecución, que es independiente del accuracy. Por otro lado, en cuanto al entrenamiento, no será necesario ejecutar tantas épocas como sea preciso para obtener convergencia del modelo. Nos interesará analizar qué operaciones son más costosas en cuanto a tiempo de ejecución, cuánto aumenta el consumo de memoria del modelo cuando está siendo entrenado, etc.

Tras el proceso de ejecución que se explicó en el apartado anterior se observaban métricas de rendimiento tales como el tiempo de CPU o el de GPU. Además, podemos reportar otras métricas como el consumo de memoria modificando los scripts de los modelos anteriores (e.g., bert-inf_cpu.py) añadiendo estas líneas:

```
profile_kwargs = ProfileKwargs(
    activities=["cpu"],
    profile_memory=True,
    record_shapes=True
)
```

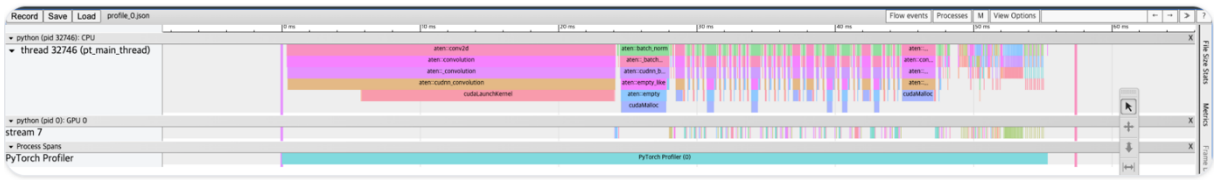
Después de lo anterior, vendría el código para realizar la inferencia del modelo. Y finalmente, realizaríamos el reporte de estadísticas que necesitamos:

```
print(prof.key_averages().table(sort_by="self_cpu_memory_usage", row_limit=10))
```

Así, tras ejecutar un experimento (tomado de la [web de Hugging Face](#)) obtendríamos una salida como la siguiente:

```
---------------------------------  ------------  ------------  ------------
                             Name       CPU Mem  Self CPU Mem    # of Calls
---------------------------------  ------------  ------------  ------------
                       aten::empty      94.85 Mb      94.85 Mb           205
    aten::max_pool2d_with_indices      11.48 Mb      11.48 Mb             1
                       aten::addmm      19.53 Kb      19.53 Kb             1
                        aten::mean      10.00 Kb      10.00 Kb             1
              aten::empty_strided         492 b         492 b             5
                         aten::cat         240 b         240 b             6
                         aten::abs         480 b         240 b             4
              aten::masked_select         120 b         112 b             1
                           aten::ne          61 b          53 b             3
                           aten::eq          30 b          30 b             1
---------------------------------  ------------  ------------  ------------
Self CPU time total: 69.332ms
```

Esta herramienta de profiling también permite exportar las trazas de ejecución a un fichero en formato json para poder ser visualizado por una herramienta externa, como por ejemplo Chrome trace viewer (en la barra de búsqueda del navegador Chrome, escribiríamos chrome://tracing para cargar el fichero de trazas gernerado). Esto es especialmente útil para llevar a cabo un profiling más exhaustivo con los kernels CUDA que se ejecutan en los experimentos que involucran un sistema de cómputo con GPU.



También se permite hacer profiling de experimentos que tarden demasiado (por ejemplo, experimentos de entrenamiento) añadiendo argumentos opcionales como:

- **schedule_option**: Permite controlar cuándo se activa el *profiling* para evitar recopilar demasiados datos en tareas largas. Sigue un ciclo con las siguientes fases:
- **skip_first**: Omitir los primeros pasos.
- **wait**: Esperar un número de pasos antes de iniciar.
- **warmup**: Ejecutar una fase de calentamiento.
- **active**: Registrar datos de rendimiento.
- **repeat**: Repetir el ciclo (si se especifica). Un valor de 0 mantiene el ciclo activo hasta que finaliza el perfilado.
- **on_trace_ready**: Especifica una función que recibe el *profiling* como entrada y se ejecuta cada vez que una nueva traza de datos está lista.

Al código tendríamos que añadir:

```
def trace_handler(p):
    output = p.key_averages().table(sort_by="self_cuda_time_total", row_limit=10)
    print(output)
    p.export_chrome_trace("/tmp/trace_" + str(p.step_num) + ".json")

profile_kwargs = ProfileKwargs(
    activities=["cpu", "cuda"],
    schedule_option={"wait": 5, "warmup": 1, "active": 3, "repeat": 2, "skip_first": 1},
    on_trace_ready=trace_handler
)
```

4 Tareas y puntuación asociada (máximo 8 puntos)

Para realizar las tareas, utilizaremos el framework Accelerate anteriormente descrito por su simplicidad, portabilidad y adecuación para permitir a los grupos de prácticas poder adquirir las competencias necesarias que permitan alcanzar los dos objetivos que se planteaban al comienzo del documento:

- Aprendizaje de librerías y frameworks modernos para aceleración de cargas de trabajo de Deep Learning en CPU y aceleradores.
- Implementación y evaluación de técnicas de programación paralela para realizar procesos de inferencia y entrenamiento de modelos de Deep Learning.

En el documento compartido en el que aparecen los grupos de prácticas que se puede consultar aquí, aparecen dos categorías de columnas adicionales, una para especificar los experimentos de inferencia y el otro para los de entrenamiento. En estas columnas, todos los grupos y el profesor tendrán una perspectiva global acerca de en qué está trabajando cada grupo de prácticas. El objetivo es que al final todos los grupos trabajen en modelos DL distintos y puedan estudiarse la mayor variabilidad de frameworks y librerías para optimización posibles.

4.1 Portado de modelos DNN a PyTorch-Accelerate (3 puntos)

Haciendo uso de Accelerate, se realizarán experimentos de inferencia y entrenamiento utilizando **cuatro modelos DNN (dos para Edge y dos para Cloud)** que el grupo de prácticas haya explorado en otras asignaturas durante el primer cuatrimestre. Si los modelos estuvieron codificados en otro framework diferente a PyTorch (e.g., TensorFlow), estos modelos tendrán que ser portados a PyTorch.

A continuación, para poder hacer uso de Accelerate, se tendrán que modificar los ficheros PyTorch de los modelos añadiendo las líneas de código necesarias para configurar la clase Accelerator, adaptar el entrenamiento, inferencia, así como para añadir instrucciones para realizar profiling tal y como se describió en el apartado 3.

Para garantizar que no se hace un uso abusivo de los recursos computacionales del clúster, todo procedimiento de inferencia y de entrenamiento en Accelerate tardará menos de 10 minutos utilizando un sistema de cómputo CPU. Esto se puede comprobar en cualquier equipo que tenga el estudiante configurando un entorno Ubuntu, instalando el sistema de virtualización Docker y creando un contenedor a partir del Dockerfile que está disponible en la carpeta Recursos/Práctica1 del Aula Virtual. Si no se dispusiera de máquina Linux, para realizar esta prueba, aunque sería más lenta la ejecución, podría utilizarse una máquina virtual Ubuntu sobre la herramienta de virtualización Oracle VirtualBox e instalar en esta máquina virtual lo necesario para realizar la prueba (Docker, contenedor con el Dockerfile).

Además, se podría estimar lo que ocupa en memoria un modelo DNN para procesos de inferencia y entrenamiento, para distintos tipos de representación de datos, utilizando el estimador de consumo de memoria de Hugging Face que viene descrito aquí. Esto sería útil también para no agotar los recursos de memoria de nuestros sistemas de cómputo antes de lanzar los experimentos.

Una vez confirmados los cuatro modelos DNN que se van a ejecutar, el grupo de prácticas tendrá que notificarlo al profesor a través de un mensaje privado quien lo dejará apuntado aquí.

4.2 Profiling exhaustivo de inferencia y entrenamiento (3 puntos)

A continuación, se realizarán experimentos de entrenamiento e inferencia con los modelos anteriores utilizando los sistemas de cómputo descritos en el apartado 2 de este documento.

A la correcta ejecución de los experimentos (no falla la ejecución), se tendrá que realizar un análisis de rendimiento en tiempo y consumo de memoria exhaustivo. Esto se podrá realizar con las herramientas proporcionadas por Accelerate descritas en el apartado 3. En particular se pide:

- Saber identificar cuáles son las funciones y/o kernels que consumen más tiempo de ejecución. Se valorará que se adjunte a las estadísticas de ejecución que ofrece el profiler descrito en este documento, un análisis visual de alguna traza de ejecución.
- Comprobar cómo se acelera la ejecución cuando se cambia el sistema de cómputo (e.g., cómo se reduce el tiempo de ejecución de CPU a GPU a Multi-GPU). Se añadirán gráficas que ayuden a interpretar los resultados y análisis realizados.
- Estudiar algunos parámetros básicos para entrenamiento e inferencia que el grupo de prácticas considere (e.g., tamaño de batch o tipo de formato de los datos), y cómo afecta a las métricas de ejecución (tiempo y consumo de memoria). Se añadirán gráficas que ayuden a interpretar los resultados y análisis realizados.

4.3 Estudio de opciones de aceleración en Accelerate y/u otros frameworks o librerías (2 puntos)

Además, se valorará que el grupo estudie algunas de las posibilidades que ofrece Accelerate y que no han sido descritas en este documento (más información aquí) para optimizar el entrenamiento e inferencia de los modelos DNN utilizados. El grupo de prácticas tendrá que documentar detalladamente cómo ha conseguido instalar (si procede) el framework/librería para poder utilizarla, así como demostrar su correcta ejecución, hacer un análisis de resultados exhaustivo (similar al apartado 4.2) que demuestre que domina adecuadamente las opciones de aceleración estudiadas.

En particular:

Entrenamiento:

- Gradient accumulation
- Local SGD
- Low precision (FP8) training
- DeepSpeed
- Using multiple models with DeepSpeed
- DDP Communication Hooks
- Fully Sharded Data Parallel

- Megatron-LM
- IPEX training with CPU

Inferencia:

- Big Model Inference
- Distributed inference

Como se mencionó anteriormente en el apartado 2 de este documento, excepcionalmente, a petición del grupo de prácticas, también se podrá explorar funcionalidades adicionales en otros frameworks de aceleración de rendimiento y/o librerías que no estén Accelerate. En ese caso, deberán consultar su viabilidad con el profesor y, de ser aprobada, proporcionar un fichero Dockerfile con los requisitos de instalación enviándolo al profesor como mensaje privado a través del Aula Virtual.

Una vez confirmados los frameworks/librerías que el grupo de prácticas va a utilizar para optimizar los procedimientos de inferencia y entrenamiento con los cuatro modelos de DNN seleccionados, el grupo de prácticas tendrá que notificarlo al profesor a través de un mensaje privado quien lo dejará apuntado aquí.

5 Entregables (2 puntos)

El grupo de prácticas entregará en la tarea correspondiente del Aula Virtual:

- **Memoria explicativa en formato PDF** (**extensión máxima 30 páginas excluyendo portada e índice**) detallando los siguientes apartados:
  1. **Técnicas de optimización para inferencia y entrenamiento utilizadas** para acelerar el proceso computacional dependiendo del sistema de cómputo (CPU, GPU y Multi-GPU). Se explicarán cada una de ellas detallando las ventajas que ofrecen para poder acelerar el cómputo y/o reducir el consumo de memoria. Se recomienda adjuntar diagramas explicativos que ayuden a su comprensión.
  2. **Metodología de Evaluación**:
     - Frameworks y librerías utilizadas para implementar las técnicas anteriores.
     - Soporte computacional: nodos y colas del clúster ATLAS usados y, en caso de uso de máquina local, descripción de las características HW de ésta.
     - Modelos de DL seleccionados para Edge y Cloud, con los que se han realizado los procesos de entrenamiento e inferencia.
     - Datasets que se han utilizado para realizar los experimentos.
  3. **Resultados Experimentales**:
     - Demostración con capturas de pantalla de la correcta ejecución de cada uno de ellos mostrando la salida que se espera obtener.
     - Gráficos que ayuden a analizar los resultados obtenidos para cada experimento, así como el *profiling hecho*.
     - Explicación detallada de los gráficos demostrando que la correcta interpretación de acuerdo con las optimizaciones realizadas.
- **Todos los ficheros de configuración (e.g., Dockerfile) y ejecución (e.g., *scripts SLURM, modelos de DL, datasets, etc.*) y listado de comandos Linux, Python, etc.** que hayan sido necesarios para poder instalar los frameworks/librerías, así como para poder ejecutar los experimentos tanto en el Clúster ATLAS como en máquina del estudiante (si procede). Además, se facilitarán las **instrucciones** pertinentes para poder reproducir fielmente todos los pasos de instalación y de ejecución de los experimentos que se hayan descrito en la memoria explicativa PDF.