Optimizing PyTorch

Accelerating Training and Inference with Compilation, Custom Kernels, and Beyond

Alvaro Moran - Hugging Face - 2024-10-03

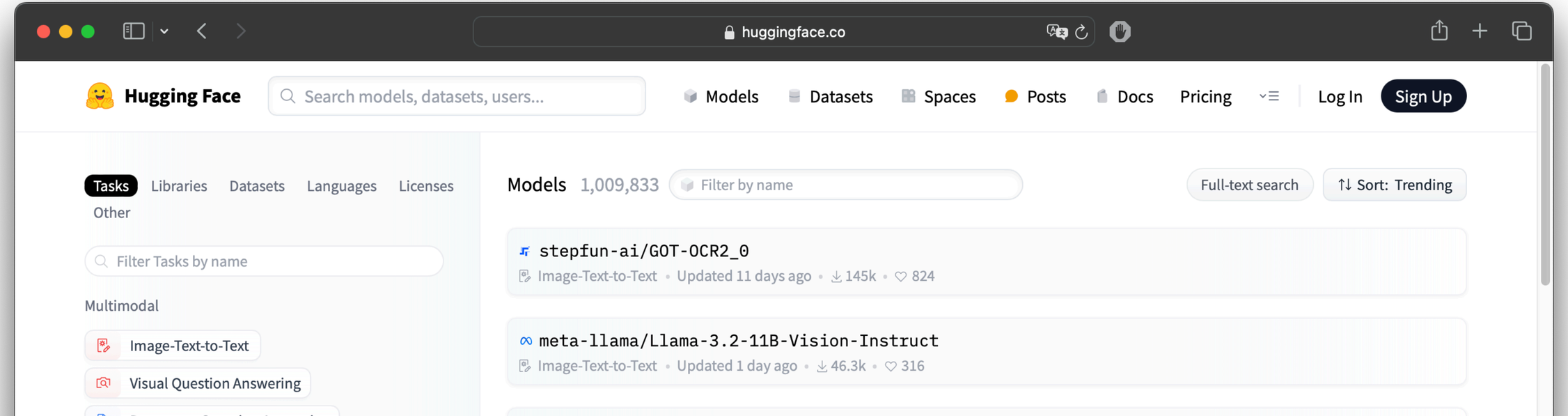
What is Hugging Face?

📅 Founded in 2016

🏢 HQ’d in New York, offices in Paris and few other cities.

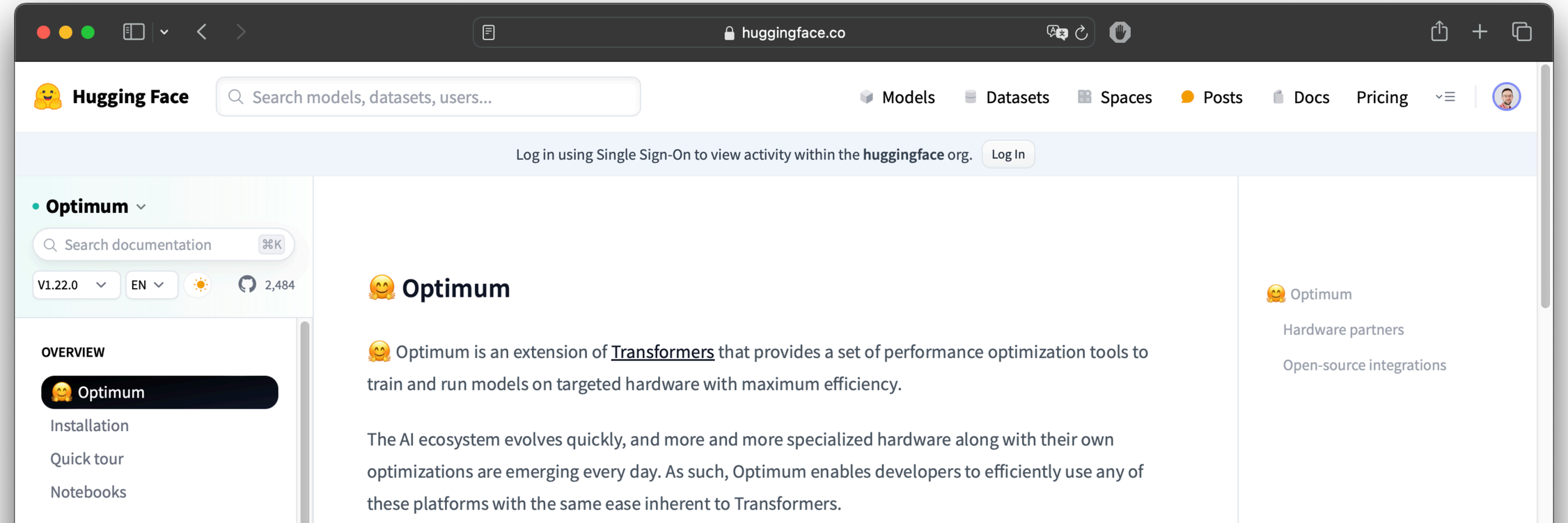
🌟 A Hub with over 1 000 000 models available.

👉 Focused on having a positive impact on the AI field.



Who am I?

👋 ML Engineer, member of the *Optimization Team*.



Agenda

- Overview of PyTorch.
- Why optimization is useful.
- Key techniques: hardware usage, `torch.compile`, custom kernels, and mixed precision and distributed processing.

Pytorch Overview

- One of the most popular machine learning libraries.
- Accelerated tensor computing for CPUs, GPUs, etc.
- Deep neural library built on automatic differentiation system.
- Used in science to create and use models for complex tasks.

Why Do We Need Optimization

🏃 Faster inference and training.

🗜️ Compress data and information, avoid out-of-memory errors.

🪡 Tailor a model for constrained hardware environments.

Use the Hardware

Pytorch can be accelerated on different hardware.

```
a = torch.tensor([1, 2, 3]).to("cuda")

model.cuda()

pipe = pipeline("image-segmentation",
                device="cuda",
                framework="pt")
```

torch.compile

It makes code faster when running several times on a given hardware.

```
@torch.compile
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b

outputs = foo(x, y)
```

NOTE: *torch.compile* can give massive speed-up, but it can be hard to use it on a whole model. Try using it on smaller code blocks.

Custom Kernels

Some operations can be optimized to be even more efficient on a given hardware. To do that it’s possible to use custom kernels:

- XLA and Pallas - easy, only on GPU and TPU
- Triton - somewhat hard - GPU
- CUDA (and C++ extensions) - hard, GPU

Half and Mixed Precision, Quantization

- Use `torch.float16`, `torch.bfloat16` if the hardware you use allows it. ➡️ Better performance, lower memory, similar accuracy.
- Quantize your model: use `bitsandbytes`, `optimum-quanto`, `marlin` or others. Mostly for matrix multiplication. ➡️ Much lower memory, lower accuracy, but usually slower.

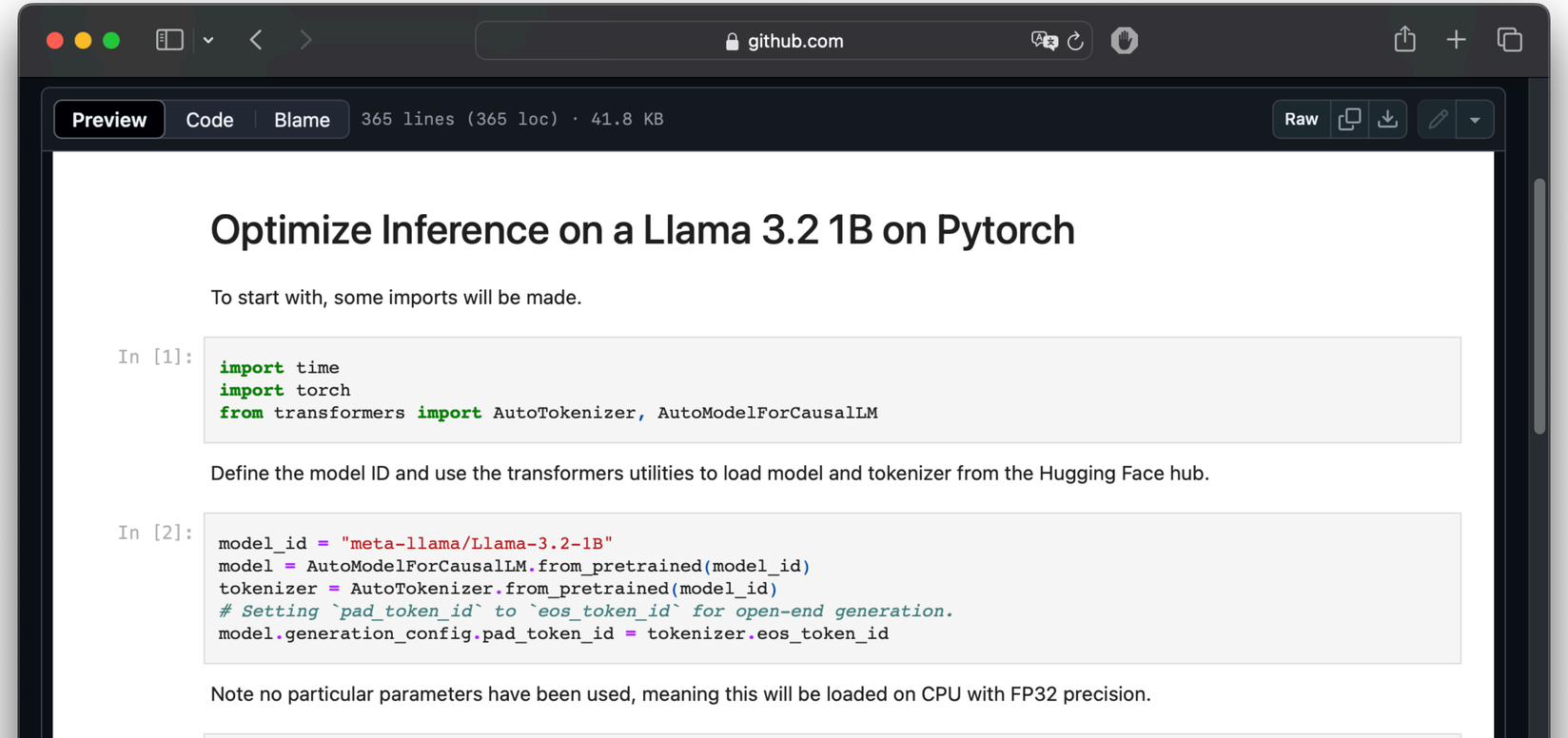
Distributed Inference and Training

If models are too big to fit into one GPU, it is possible to distribute processing across GPUs. To do that, it is possible to use different tools:

- Use `torch.distributed` to split processing across GPUs and synchronize (this can be difficult).
- When training, consider using Fully Sharded Data Parallel (FSDP) or similar techniques to *shard* the model across GPUs. See `torch.distributed.fsdp.FullyShardedDataParallel`
- Consider using HuggingFace's `accelerate` library to simplify these scenarios.

Practical Example

A Jupyter notebook is available to walk through some of the mentioned techniques, and it is possible to run it on a common laptop.



Thank You!