HPML blogposts

Release 0.1

Bryan Cardenas Guevara

Dec 06, 2022

TUTORIALS:

CHAPTER ONE

SURF HPML DOCUMENTATION!

The high performance machine learning group at SURF facilitates efficient deep learning usage on the Dutch national supercomputer. Here we provide the documentation for our tutorials, presentations and blogposts!

Our group has in-house expertise on several topics including computational histopathology, GPU programming, physics informed DL, multi-modal Learning and large language modelling! For more information please go to this post about ML in HPC environments!

Warning: This project is currently under verocious development.

This GitHub template includes tutorials, blogposts and slides.

SURF Website: https://www.surf.nl/
Repository: https://github.com/Cryptheon/hpml-surf
Author: Bryan Cardenas Guevara

CHAPTER

TWO

CONTENTS:

2.1 Large Language Models on Snellius

In this post we demonstrate how to run a large language model on Snellius. The main purpose of this small experiment was explorative in nature; to which extent can we perform generation or latent extraction on Snellius? How much compute is needed for a single prompt?

We will mainly discuss:

1. How we got it working on Snellius.
2. how to run it.
3. a few examples.

The main repository and the tested downloaded models can be found on Snellius under `/projects/0/hpmlprjs/GALACTICA/`. For now, we named it GALACTICA as it was solely intended for the new Meta's Galactica scientific models. Although, we could use any causal language model uploaded to the Huggingface Huggingface hub. More specifically, any model that can be loaded using `AutoTokenizer` and `AutoModelForCausalLM`. Do note

Note: Testing is still necessary as some models break under specific `PyTorch`, `transformers` or `DeepSpeed` versions.

Warning: This blog is mainly intended for the HPML members for now. A more public version is coming soon, GPUs near you.

For now we have tested four different language models:

- BLOOM, a multi-language language model (40+ languages)
- Galactica 6.7b, the galactic models are a family of LMs trained solely on scientific data
- OPT-30b, LM trained on 800GB of text data (180B tokens).
- GPT-NeoX-20b, LM trained by EleutherAI on The Pile

These models all have one clearly overlapping feature; they are decoder-transformers similar in shape to GPT-2 and GPT-3. It stands to overemphasize that each has their own qualities and desired properties and as such, it would be beneficial to keep a few of these models on Snellius as the need arises.

2.1.1 1. How we got it running on Snellius

We will see how we downloaded and loaded the model for generation.

Let's take galactica uploaded by Meta on Huggingface as an example. The sharded model can be found under `files and versions`. We first need to have git lfs installed to be able to download these files on our disk.

We can use

```
git lfs clone https://huggingface.co/facebook/galactica-6.7b/
```

or we can just use `git clone` in this case.

Note: Using git lfs for larger language models such as BLOOM-176b, we would first be downloading specific binaries that would need to be constructed afterwards by running `git lfs checkout`.

Let's look at how to load this model using Hugginface. We use `transformers==4.21` and `accelerate`, which is Hugginface's own distributed computing framework that will make our lives easier for now.

Loading the Tokenizer and Model

To avoid bloat and confusion we show the important parts only, please take a look at `./GALACTICA/lm_gen.py` for more details.

```
tokenizer = AutoTokenizer.from_pretrained(args.model_path)

kwargs = dict(device_map="auto", load_in_8bit=False)

model = AutoModelForCausalLM.from_pretrained(args.model_path, **kwargs)
```

Here we see how we prepare the tokenizer and load the model for the given model path. In this case we use `/GALACTICA/langauge_models/galactica-6.7b`, in which we can find the model weights and the tokenizer. In kwargs we can see `device_map="auto"` and `load_in_8bit=False`.

With the former we tell the accelerate framework to load the checkpoint automatically. The accelerate framework enables us to run a model in any distributed configuration, it supports sharded models and full checkpoints. The model gets loaded first by initializing a model with `meta` (read: empty) weights and then it determines how to load the sharded model across the available GPUs. It employs a simple pipeline parallelism method and while this is not the most efficient method, it's the most flexible for a large variety of models. See this language modeling guide for a quick glance in how this works. For instance, with `./GALACTICA/lm_gen.py` we could load BLOOM 176b model with only one GPU! It might not be the most efficient execution, but hey, it works :).

The latter argument `load_in_8bit` makes it possible to load in a model while using less memory. This approach 8-bit quantizes the model with super minimal performance loss. The main idea is to make large language models more accessible with a smaller infrastructure. For instance, this method allows us to load the full BLOOM 176b model on eight A100 40GB GPUs, as opposed to using 16 A100 GPUs. However, as nothing is free in life, this comes at the cost of inference time. We can expect forward propagation slow downs of 16-40%. I encourage you to read this blog post as it's a good read (or, the paper).

Generation

As we tokenize our input and load our model we can easily generate a piece of text given our input by using Huggingface's generate function which is implemented for CausalLMs:

```
generate_kwargs = dict(max_new_tokens=args.num_tokens, do_sample=True, temperature=args.
↪temperature)

outputs = model.generate(**input_tokens, **generate_kwargs)
```

I trust that most of these arguments are familiar to us. The `input tokens` is a dictionary containing the tokenized input text (`input_ids`), an optional `attention mask` and `token_type_ids`. For the record, `token_type_ids` is not accepted by galactica-type models. Most of the time we are only interested in the `input_ids`, but some models require the other tensors as input as well.

DeepSpeed-Inference

The script `./GALACTICA/lm_gen_ds.py` contains code to run model inference with deepspeed. The biggest difference with `./GALACTICA/lm_gen.py` is the way deepspeed has to be initialized. Luckily, for our purposes for now this can remain minimal:

```
model = deepspeed.init_inference(
        model=model,      # Transformers models
        dtype=torch.float16, # dtype of the weights (fp16)
        replace_method=None, # Lets DS autmatically identify the layer to replace
        replace_with_kernel_inject=False, # replace the model with the kernel injector
    )
```

Deepspeed deploys Tensor parallelism that mainly distributes each layer ''horizontally"; it splits up the layer and distributes it across the GPUs, each shard then lives on its appointed gpu. Additionally, it gives us the capability to replace some modules with specialized CUDA kernels to run these layers faster. I've run this but we are not getting the correct output. This should be fixable though.

We have been having OOM problems running `lm_gen` with the `deepspeed` launcher. The galactica-6.7b model and any smaller model should work without the deepspeed launcher but we are yet to fix this for models such as gpt-neox-20b or bigger. We consistently see a 2x speedup using Deepspeed. Check out this tutorial that helped us setting this up.

Deepspeed ZeRO is an add-on to the usual DeepSpeed pipeline, it also performs sharding in a tensor parallelism fashion but with, what they call, ''stage 3" it is able to do some intelligent tensor off-loading. This can come in particularly handy with large models such as BLOOM 176b or OPT-175b. We haven't been able to get this one off the grounds for reasons unknown; it seems to get stuck forever, while generating with regular deepspeed takes a few seconds.

See the following links for more information about `ZeRO stage-3`:

1. https://www.deepspeed.ai/2021/03/07/zero3-offload.html
2. https://www.deepspeed.ai/tutorials/zero/
3. https://www.deepspeed.ai/2022/09/09/zero-inference.html

2.1.2 2. How to run as a module on Snellius

To module load OptimizedLMs add the following line to your bashrc:

```
export MODULEPATH="$MODULEPATH:/projects/0/hpmlprjs/scripts
source ~/.bashrc
```

Now we can load the module you linked to in your .bashrc.

```
module load OptimizedLMs
```

And then run with

```
lm_gen model_choice input output num_tokens temperature
```

Anoter way is to load and install your own packages:

The scripts `./GALACTICA/lm_gen.py` and `./GALACTICA/lm_gen_ds.py` can be run as is with the correct dependencies.

```
module load 2021
module load Python/3.9.5-GCCcore-10.3.0
module load PyTorch/1.11.0-foss-2021a-CUDA-11.6.0
module load Miniconda3/4.9.2

pip install mpi4py, deepspeed, pydantic
pip install transformers==4.24, accelerate
```

And then run:

```
python lm_gen.py --model_path ./language_models/galactica-6.7b/ --batch_size 2␣
↪--num_tokens 1000 --input_file ./texts/inputs/geny.txt --temperature 0.95 --
↪output_file ./texts/generations/out
```

Supported Models

For now, we have briefly tested the following models with `accelerate`.

1. galactica-6.7b
2. opt-30b
3. gpt-neox-20b
4. BLOOM

The weights of these models live under `/projects/0/hpmlprjs/GALACTICA/language_models/`.

Attention: As of now, deepspeed-inference is only compatible with galactica-6.7b.

2.1.3 3. Examples

Let's run a few examples.

```
lm_gen galactica-6.7b alpha.txt out 75 0.95
```

Where `alpha.txt` contains:

```
"The function of proteins is mainly dictated by its three dimensional␣
↪structure. Evolution has played its part in"
```

Output:

The function of proteins is mainly dictated by its three dimensional structure. Evolution has played its part in selecting the best possible protein structure that can perform its functions. This structure is called native structure and it corresponds to the minimum of potential. There are several methods to compute the structure of a protein starting from amino acid sequence. With the help of evolutionary knowledge, experimental information and many other techniques like computational tools etc. we have made significant progress in prediction of

This took 5.5s to generate excluding model loading (the model fits in memory). We actually generated a batch of 4 examples in 5.5s. With `lm_gen_ds` we generate this same batch size in 2.7s! For reference, running opt-30b with `lm_gen` takes 8s.

If you feel like it, you can run `lm_gen BLOOM input out 50 0.95` and see how it takes ~40 minutes to run.

2.2 Profiling with PyTorch

```
[ ]: # Kill old TensorBoard sessions
     import os
     username = os.getenv('USER')
     !kill -9 $(pgrep -u $username -f "multiprocessing-fork")
     !kill -9 $(pgrep -u $username tensorboard)
```

```
[ ]: import os
     from typing import Sequence, Tuple
     from datetime import datetime

     import torch
     import torch.nn as nn
     import torch.nn.functional as F
     import torch.optim as optim
     from torch.utils.data import Dataset
     import torchmetrics.functional as metrics
     import numpy as np
     from torchvision import datasets, transforms, models
     import matplotlib.pyplot as plt

     %matplotlib inline

     DATA_PATH = os.getenv('TEACHER_DIR', os.getcwd()) + '/JHL_data'
     os.environ["OMP_NUM_THREADS"]="3"
```

2.2.1 What is profiling?

According to wikipedia:

"Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering."

What this means is that you analyse your program, trying to identify bottlenecks, and thereby optimizing it's execution. As an example, you might have an application needs to read a lot of input data (quite typical in machine learning!) during its run. A profile might show you that while your code runs, your processor is mostly idling since it is waiting for input data. This might give you a hint on how to optimize your program: maybe you can read in *part* of the input, and already start computing on that *while* you're loading in your next samples. Or: maybe you can copy your data to a faster disk, before you start running.

2.2.2 Why should I care about profiling?

You may know that training large models like GPT-3 takes several *million* dollars source and a few hundred MWh source. If the engineers that trained these models did *not* spend time on optimization, it might have been several million dollars and hunderds of MWh more.

Sure, the model you'd like to train is probably not quite as big. But maybe you want to train it 10000 times, because you want to do hyperparameter optimization. And even if you only train it once, it may take quite a bit of compute resources, i.e. money and energy.

2.2.3 When should I care about profiling?

Well, you should *always* care if your code runs efficiently, but there's different levels of caring.

From personal experience: if I know I'm going to run a code only once, for a few days, on a single GPU, I'll probably not create a full profile. What I *would* do is inspect my GPU and CPU utilization during my runs, just to see if it is *somewhat* efficient, and if I didn't make any obvious mistakes (e.g. accidentally *not* using the GPU, even if I have one available).

If I know that I'll run my code on multiple GPUs, for multiple days, (potentially) on multiple nodes, and/or I need to run it multiple times, I know that my resource footprint is going to be large, and it's worth spending some time and effort to optimize the code. That's when I'll create a profile. The good part is: the more often you do it, the quicker and more adapt you become at it.

2.2.4 Define the necessary functions of code

First, we will define the necessary functions to run the training. That means we define - A model (as a class deriving from nn.Module) - Some code to plot the accuracy / loss curves - Some code to set the device on which we want to execute - A train & test loop (this time with some profiling code)

Define model

[ ]:
```
class CIFAR10CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # 4 convolution layers, with a non-linear activation after each.
        # maxpooling after the activations of the 2nd, 3rd, and 4th conv layers
        # 2 dense layers for classification
        # log_softmax
        #
        # As for the number of channels of each layers, try to experiment!

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1),
            nn.ReLU(),

            nn.Conv2d(in_channels=8, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )

        # in_features of the first layer should be the product of the output shape of␣
↪your feature extractor!
        # E.g. if the output of your feature extractor has size (batch x 128 x 4 x 4),␣
↪in_features = 128*4*4=2048
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=2048, out_features=2048),
            nn.ReLU(),
            nn.Linear(in_features=2048, out_features=10),
            nn.LogSoftmax(dim=1)
        )



    def forward(self, x):
        features = self.feature_extractor(x)

        return self.classifier(features)
```

Define function for plotting metric curves

```
[ ]: def plot_metric_curve(
         train_metric: Sequence[float],
         val_metric: Sequence[float],
         n_epochs: int,
         metric_name: str, # Label of the y-axis, e.g. 'Accuracy'
         x_axis_name: str = 'Epoch'
     ):
         # create values for the x-axis
         train_steps, val_steps = map(
             lambda metric_values: np.linspace(start=0, stop=n_epochs, num=len(metric_
     ↪values)),
             (train_metric, val_metric)
         )

         plt.plot(train_steps, train_metric, label='train')
         plt.plot(val_steps, val_metric, label='validation')
         plt.title(f"{metric_name} vs. {x_axis_name}")
         plt.legend()
         plt.show()
```

Define some code to select the right device

```
[ ]: use_cuda = torch.cuda.is_available()
     print(f"CUDA is {'' if use_cuda else 'not '}available")
     device = torch.device("cuda" if use_cuda else "cpu")

     if use_cuda:
         torch.cuda.set_per_process_memory_fraction(0.22)
```

Define train and test loop

The `train(...)` function is just for reference: this is the same train function you used in the CIFAR10 hands-on earlier. Then, we define the `train_profiling(...)` function, which is essentially the same training loop, but with the necessary code added to generate the profiles. Finally, we specify the `test(...)` function. This is the same as from the CIFAR10 hands-on earlier, but note that in theory we could also profile the testing part. Usually, training takes by far the most time, and is therefore the most important to optimize. There are exceptions however, e.g. if you use certain metrics that are very heavy to compute, and you want to compute those on a large test dataset. In that case, optimizing the test loop might make sense as well.

```
[ ]: # This train(...) loop is just for reference, so that you can compare to train_
     ↪profiling(...)
     def train(model, device, train_loader, optimizer, epoch, log_interval=10):
         model.train()
         for batch_idx, (data, target) in enumerate(train_loader):
             # move data and target to the gpu, if available and used
             data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
     ↪target))
```

(continues on next page)

(continued from previous page)

```
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

        accuracy = metrics.accuracy(output, target)

        if batch_idx % log_interval == 0:
            print(
                f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                f'\tLoss: {loss.detach().item():.6f}'
                f'\tAccuracy: {accuracy.detach().item():.2f}'
            )

        yield loss.detach().item(), accuracy.detach().item()

# This is the actual train loop we will use for profiling
def train_profiling(model, device, train_loader, optimizer, epoch, log_interval, logdir,␣
↪break_idx=None):
    model.train()

    # Create a torch.profiler.profile object, and call it as the last part of the␣
↪training loop
    with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=10,
        warmup=10,
        active=10),
    on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir, worker_name='worker0
↪'),
    record_shapes=True,
    profile_memory=True,  # This will take 1 to 2 minutes. Setting it to False could␣
↪greatly speedup.
    with_stack=True
) as p:
        for batch_idx, (data, target) in enumerate(train_loader):
            # move data and target to the gpu, if available and used
            data, target = map(lambda tensor: tensor.to(device, non_blocking=True),␣
↪(data, target))

            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            optimizer.step()

            accuracy = metrics.accuracy(output, target)
```

(continues on next page)

(continued from previous page)

```
            if batch_idx % log_interval == 0:
                print(
                    f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                    f'\tLoss: {loss.detach().item():.6f}'
                    f'\tAccuracy: {accuracy.detach().item():.2f}'
                )

            yield loss.detach().item(), accuracy.detach().item()

            p.step()

            # Allow to break early for the purpose of shorter profiling
            if (break_idx is not None) and (batch_idx == break_idx):
                break

# We leave the test loop unchanged. One thing to note though is that the test loop is␣
↪decorated
# with the @torch.no_grad() decorator. This tells PyTorch that it doesn't need to compute␣
↪gradients
# in the test loops, as those are not needed. This will speed up execution.
@torch.no_grad()
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for data, target in test_loader:
        # move data and target to the gpu, if available and used
        data, target = map(lambda tensor: tensor.to(device, non_blocking=True), (data,␣
↪target))

        # get model output
        output = model(data)

        # calculate loss
        test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch␣
↪loss

        # get most likely class label
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-
↪probability

        # count the number of correct predictions
        correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100 * correct / len(test_loader.dataset)))
```

(continues on next page)

(continued from previous page)

```
    yield test_loss, correct / len(test_loader.dataset)
```

Specify the fitting function

Here, we combine the functions we defined above into a single function that will take care of the training, validation, loop over multiple epochs, and finally plot the results. Note that we call the `train_profiling` function from here, which is the one that will generate the profile.

```
[ ]: def fit_profiling(model, optimizer, n_epochs, device, train_loader, test_loader, log_
↪interval, logdir, break_idx):

    # get the validation loss and accuracy of the untrained model
    start_val_loss, start_val_acc = tuple(test(model, device, test_loader))[0]

    # don't mind the following train/test loop logic too much, if you want to know what's␣
↪happening, let us know :)
    # normally you would pass a logger to your train/test loops and log the respective␣
↪metrics there
    (train_loss, train_acc), (val_loss, val_acc) = map(lambda arr: np.asarray(arr).
↪transpose(2,0,1), zip(*[
        (
            [*train_profiling(model, device, train_loader, optimizer, epoch, log_
↪interval, logdir, break_idx)],
            [*test(model, device, test_loader)]
        )
        for epoch in range(n_epochs)
    ]))

    # flatten the arrays
    train_loss, train_acc, val_loss, val_acc = map(np.ravel, (train_loss, train_acc, val_
↪loss, val_acc))

    # prepend the validation loss and accuracy of the untrained model
    val_loss, val_acc = (start_val_loss, *val_loss), (start_val_acc, *val_acc)

    plot_metric_curve(train_loss, val_loss, n_epochs, 'Loss')
    plot_metric_curve(train_acc, val_acc, n_epochs, 'Accuracy')
```

Defining a custom dataloader

One of the things we learn in this profiling tutorial is the importance of an efficient data I/O pipeline. We start here with a simple custom PyTorch Dataset. This Dataset can read individual `*.png` images from a directory, and can be passed a single `*.pkl` file with all the labels in a dictionary that is indexed by the filenames of the image files (so that we know which label belongs to which image).

```
[ ]: import pickle
import glob
from torchvision.io import read_image
from PIL import Image
```

(continues on next page)

(continued from previous page)

```
class Cifar10PNGDataset(Dataset):
    def __init__(self, label_file, img_dir, transform=None, target_transform=None):
        # Load labels
        with open(label_filename, 'rb') as fo:
            self.label_dict = pickle.load(fo, encoding='bytes')
        # List filenames with png extension
        self.img_filenames = glob.glob(os.path.join(img_dir, '*.png'))
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.label_dict)

    def __getitem__(self, idx):
        # Get filename with index idx:
        img_filename = self.img_filenames[idx]
        # Read file from disk
        # image = read_image(img_filename)
        image = Image.open(img_filename)
        # Read label from label dictionary
        label = self.label_dict[os.path.basename(img_filename)]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Let's check that our dataloader actually works. It should show an image and its corresponding label. You can change the `sample_idx` to check different samples

[ ]:
```
PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA)

# Inspect one image and one label as example:
sample_idx = 0
img_sample = train_dataset[sample_idx][0]
label_sample = train_dataset[sample_idx][1]

#plt.imshow(img_sample.permute(1,2,0))
plt.imshow(img_sample)
print(f"Label: {label_sample}")
print(f"(0 = airplane, 1 = automobile, 2 = bird, 3 = cat, 4 = deer, 5 = dog, 6 = frog, 7␣
↪= horse, 8 = ship, 9 = truck)")
```

2.2.5 Creating the profile

To create the profile, we now run a single epoch of the training

```
[ ]: BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 1
BREAK_AFTER_N_ITERATIONS = 60

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/baseline/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

2.2.6 Inspect the profile

Now that we have generated a profile, we want to inspect it. For that, we will start a TensorBoard session in the cell below. It will give you a link to the running tensorboard instance.

If, for some reason, you were not able to generated the logs yourself, you can uncomment the line `logdir = ...` and inspect a reference profile that we generated for you.

WARNING: profiles (so called 'trace files') contain a lot of data. It may take a while (up to a minute or so) before the TensorBoard interface actually displays the data.

```
[ ]: %%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/baseline/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

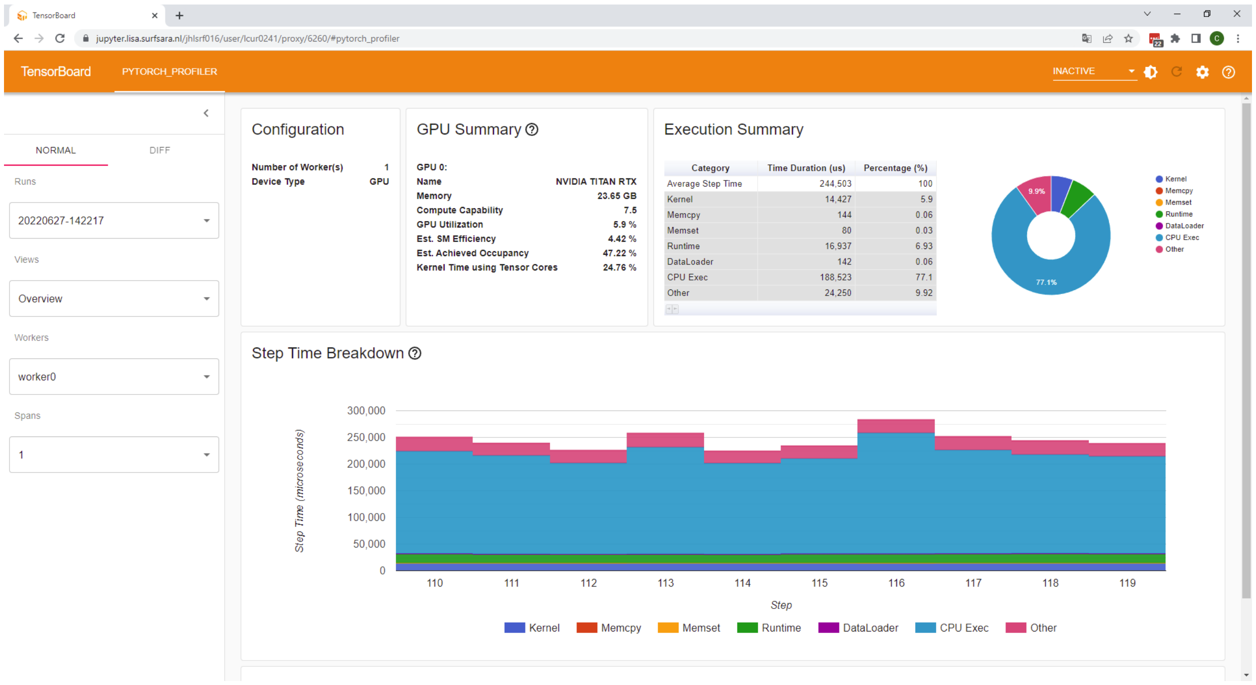
2.2.7 Understanding the profiler output

Views

In the first fiew, you see an overview of the profiling. This is probably the display that will provide the most actionable insights. What do we see? This blog contains a very detailed explaination. Below, we will cover the basics, but if you are going to profile your own code, the blog may prove very useful to get the most out of the profiler output.

Step Time Breakdown

Let's start with the central part, the Step Time Breakdown.



Configuration

Number of Worker(s)	1
Device Type	GPU

GPU Summary

GPU 0:	
Name	NVIDIA TITAN RTX
Memory	23.65 GB
Compute Capability	7.5
GPU Utilization	5.9 %
Est. SM Efficiency	4.42 %
Est. Achieved Occupancy	47.22 %
Kernel Time using Tensor Cores	24.76 %

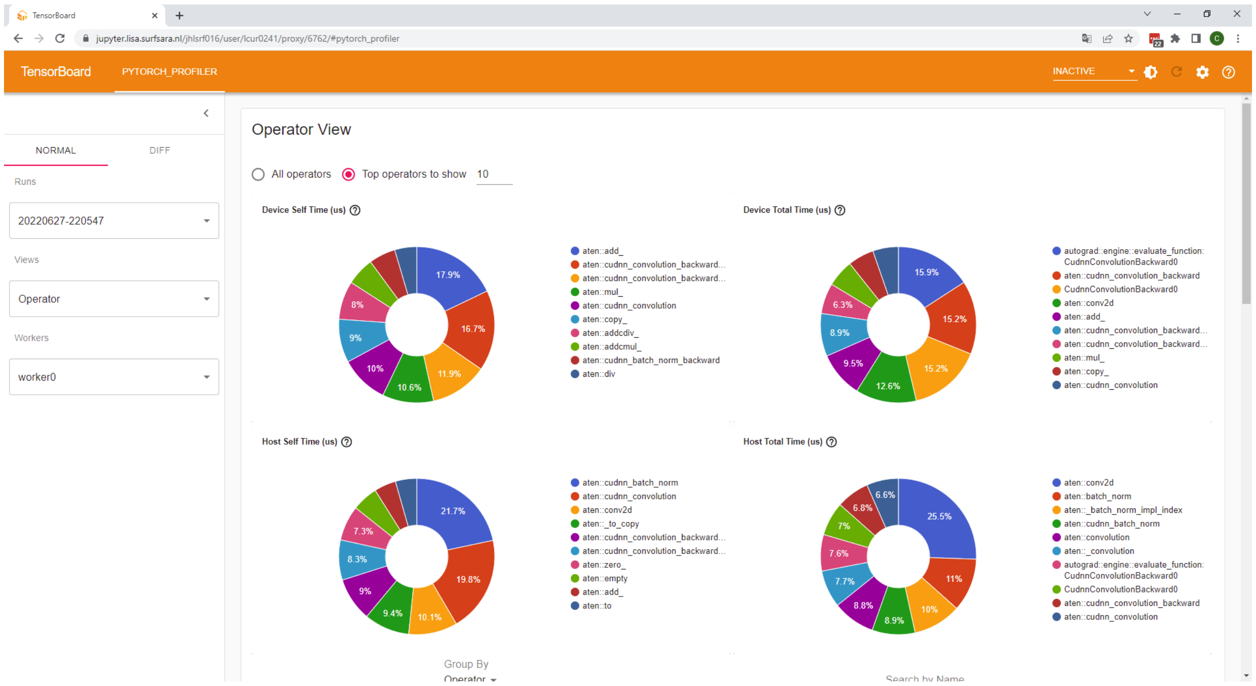
Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	244,503	100
Kernel	14,427	5.9
Memcpy	144	0.06
Memset	80	0.03
Runtime	16,937	6.93
DataLoader	142	0.06
CPU Exec	188,523	77.1
Other	24,250	9.92

Here, we see how long each 'Step' took, and what that time was spent on. A 'Step' is a single training step, i.e. a forward and backeward pass on a single batch. If you look carefully at the `train_profiling(...)` code, and in particular the `torch.profiler.profile` call we did there, you see that we passed `wait=10, warmup=10, active=10` as arguments. What that means is, it waits for the first 10 steps (i.e. step 0-9), then, the profiler gets activated, but discards it results (for step 10-19), and then 10 steps get recorded (i.e. step 20-29). This cycle will keep repeating itself as long as we were training. With a batch size of 128 and 50000 images in total, the cycle would normally be repeated 13 times if we trained for one epoch. The PyTorch profiler calls each of these cycles a 'Span'. In this case however, we cut the training shorter for faster profiling, and only trained for 60 iterations. Thus, we'll only see two spans in our profile. Thus, this cycle repeated 13 times.

Spans

Click the 'Spans' button. Here, you see that indeed, we recorded 2 'spans'. If you click on span 2, you'll see that this contains the timing for step 50-59.



Runs: 20220627-220547

Views: Operator

Workers: worker0

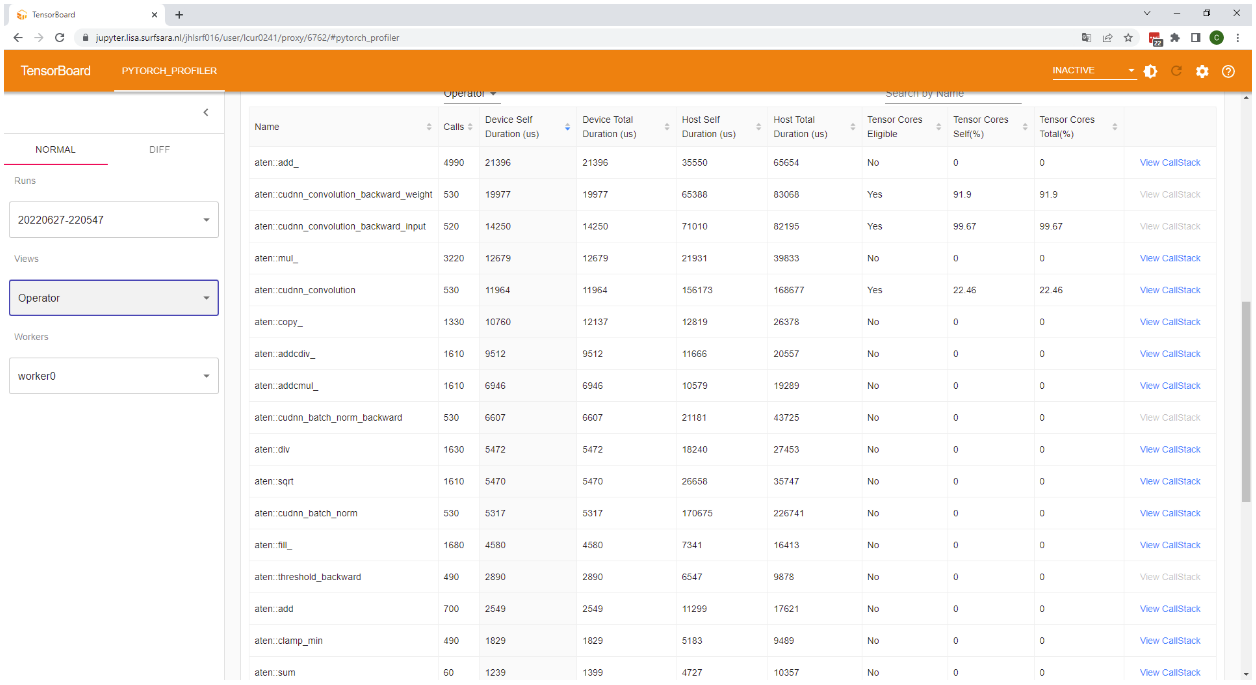
Operator View

Top operators to show: 10

Typically, recording only a few iterations/spans is enough. In very particular cases, you might be interested how the speed develops over time. Then, it is useful to record and inspect multiple spans.

GPU Summary

The next window that is useful (if training on GPUs) is the GPU Summary. However, in order to understand the GPU summary, it's good to know a little bit about how cores on a GPU are organized. Below, you see the schematic layout of an Nvidia A100 GPU chip (source)



Runs: 20220627-220547

Views: Operator

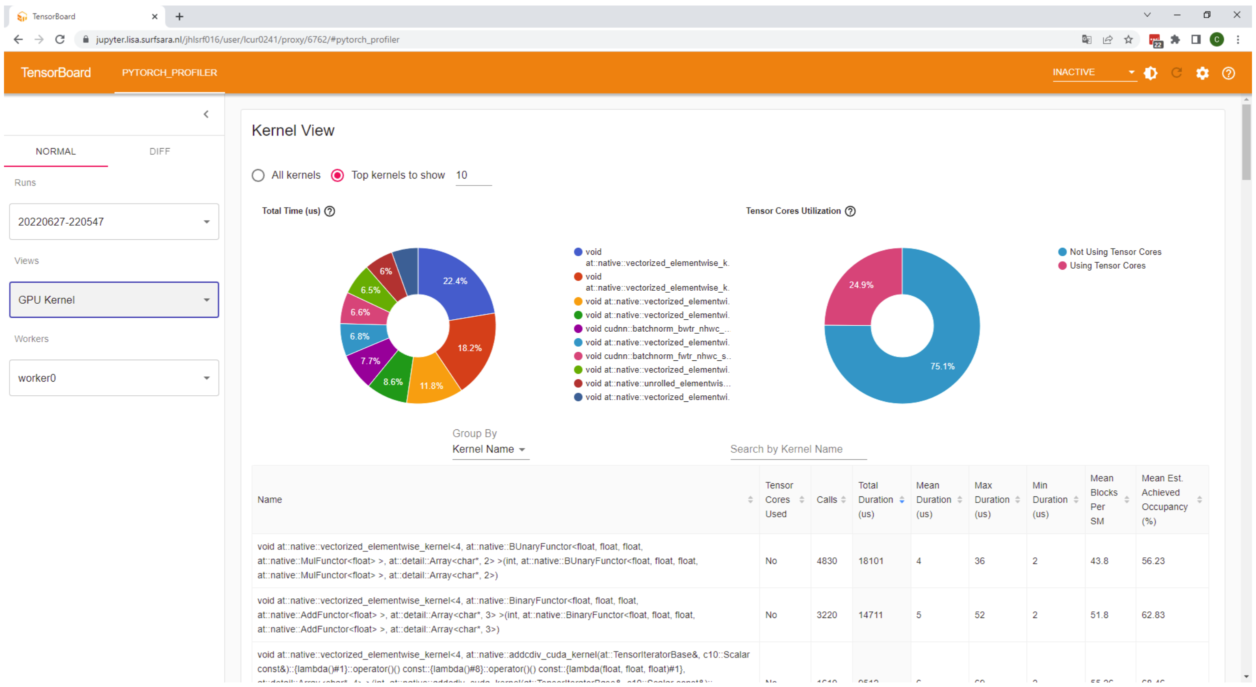
Workers: worker0

Name	Calls	Device Self Duration (us)	Device Total Duration (us)	Host Self Duration (us)	Host Total Duration (us)	Tensor Cores Eligible	Tensor Cores Self(%)	Tensor Cores Total(%)	
aten::add_	4990	21396	21396	35550	65654	No	0	0	View CallStack
aten::cudnn_convolution_backward_weight	530	19977	19977	65388	83068	Yes	91.9	91.9	View CallStack
aten::cudnn_convolution_backward_input	520	14250	14250	71010	82195	Yes	99.67	99.67	View CallStack
aten::mul_	3220	12679	12679	21931	39833	No	0	0	View CallStack
aten::cudnn_convolution	530	11964	11964	156173	168677	Yes	22.46	22.46	View CallStack
aten::copy_	1330	10760	12137	12819	26378	No	0	0	View CallStack
aten::addcdiv_	1610	9512	9512	11666	20557	No	0	0	View CallStack
aten::addcmul_	1610	6946	6946	10579	19289	No	0	0	View CallStack
aten::cudnn_batch_norm_backward	530	6607	6607	21181	43725	No	0	0	View CallStack
aten::div	1630	5472	5472	18240	27453	No	0	0	View CallStack
aten::sqrt	1610	5470	5470	26658	35747	No	0	0	View CallStack
aten::cudnn_batch_norm	530	5317	5317	170675	226741	No	0	0	View CallStack
aten::fill_	1680	4580	4580	7341	16413	No	0	0	View CallStack
aten::threshold_backward	490	2890	2890	6547	9878	No	0	0	View CallStack
aten::add	700	2549	2549	11299	17621	No	0	0	View CallStack
aten::clamp_min	490	1829	1829	5183	9489	No	0	0	View CallStack
aten::sum	60	1239	1399	4727	10357	No	0	0	View CallStack

The tiny green dots in this image are GPU cores. They are organized together in groups, so called SM (or: streaming multiprocessor) units. An A100 GPU has 128 SM units. Below, we see a schematic representation of a single SM on an Nvidia A100 GPU

Here, you see that a single SM has many different cores: 16 cores that can perform integer operations, 16 cores that can perform single precision floating point operations (i.e. operations on numbers represented in memory by 32 bits), 8 cores that can perform double precision floating point operations (i.e. operations on numbers represented in memory by 64 bits) and a block of Tensor Cores (special cores for tensor operations). An SM only has a single instruction unit. What that means that all cores in an SM can only perform *one* operation at the same time. I.e. we can not have 1 FP32 core doing subtraction, and the other multiplication, within the same SM unit. Of course, *different* SM units can work on different things. I.e. this particular GPU could in theory run 128 *different* operations, and run each of those operations on e.g. 16 single precision floating point numbers *in parallel*. This massive parallelism is what makes a GPU so good at processing many data elements with the same operation, as is done in a lot of matrix operations.

Ok, back to the summary.


TensorBoard
jupyter.lisa.surfsara.nl/jhlsrf016/user/lcur0241/proxy/6762/#pytorch_profiler
TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Runs
20220627-220547
Views
GPU Kernel
Workers
worker0
Kernel View
All kernels
Top kernels to show 10
Total Time (us)
22.4%
18.2%
11.8%
8.6%
7.7%
6.8%
6.6%
6.5%
6%
void at::native::vectorized_elementwise_k.
void at::native::vectorized_elementwise_k.
void at::native::vectorized_elementwi.
void at::native::vectorized_elementwi.
void cudnn::batchnorm_bwtr_nhwc_...
void at::native::vectorized_elementwi.
void cudnn::batchnorm_fwtr_nhwc_s..
void at::native::vectorized_elementwi.
void at::native::unrolled_elementwis...
void at::native::vectorized_elementwi.
Tensor Cores Utilization
24.9%
75.1%
Not Using Tensor Cores
Using Tensor Cores
Group By
Kernel Name
Search by Kernel Name

Name	Tensor Cores Used	Calls	Total Duration (us)	Mean Duration (us)	Max Duration (us)	Min Duration (us)	Mean Blocks Per SM	Mean Est. Achieved Occupancy (%)
void at::native::vectorized_elementwise_kernel<4, at::native::BUnaryFunctor<float, float, float, at::native::MulFunctor<float> >, at::detail::Array<char*, 2> >(int, at::native::BUnaryFunctor<float, float, float, at::native::MulFunctor<float> >, at::detail::Array<char*, 2>)	No	4830	18101	4	36	2	43.8	56.23
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float> >, at::detail::Array<char*, 3> >(int, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float> >, at::detail::Array<char*, 3>)	No	3220	14711	5	52	2	51.8	62.83
void at::native::vectorized_elementwise_kernel<4, at::native::addcdiv_cuda_kernel(at::TensorIteratorBase&, c10::Scalar const&)::{lambda()#1}::operator()() const::{lambda()#8}::operator()() const::{lambda(float, float, float)#1},	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

The first thing to note here is the GPU Utilization. As we can see, in this example it is only 3.39%. That means that 96.61% of the time our GPU was doing *nothing*, typically because it is waiting for other resources (I/O, CPU, . . .). That's bad: we have a very expensive GPU in this machine, and we're hardly using it!

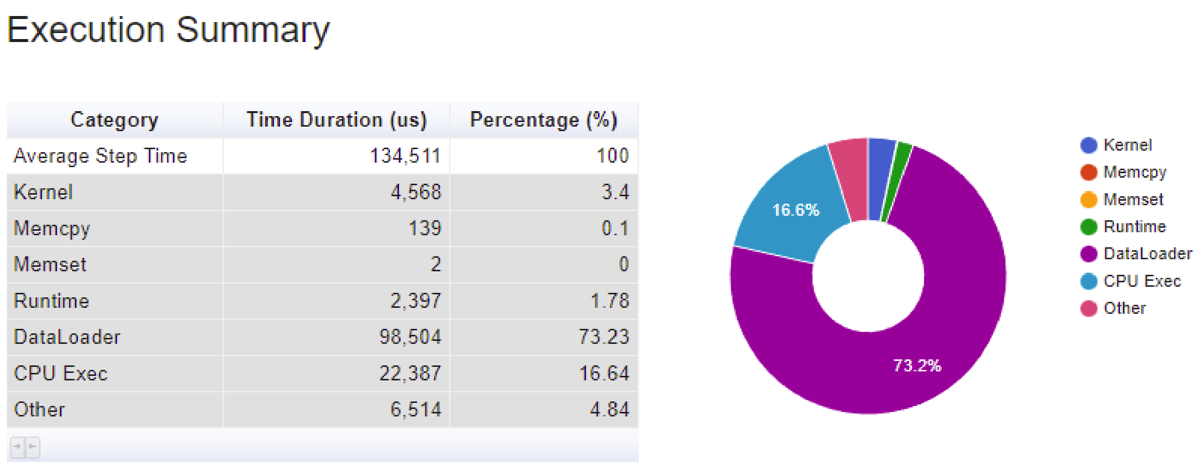
Note that a high GPU utilization does not *necessarily* mean we are using the GPU efficiently. Even if just a *single* core (out of the thousands that a GPU has) is working on a task the full time, GPU utilization would be 100%. However, it would mean we are only using a single core, on a single SM, leaving all of those others idle. As you can see, a high GPU utilization is not a *guarantee* for efficient usage, but it is a *minimum requirement*.

The Est SM Efficiency provides us some deeper insight. It is the (average) fraction of SMs that were in use over the profiled time period. A low percentage here means SMs have been idling. The aforementioned extreme case of a single core (in a single SM) working all the time would immediately be identified with this metric: GPU utilization would be 100%, but Est SM Efficiency would be 1/128 (in case of an A100 with 128 SM units in total). However, a high Est SM Efficiency is *still* not enough to guarantee efficient usage. A use case that occupies *one* core in each SM would also show a high Est SM Efficiency. In a machine learning workload, (partially) empty SMs can occur if there are operations being performed on relatively small matrices. That can be because input sizes are small, or because the batch size is small. I.e. batch size is *one* of the things you can explore if you see a low Est. SM Efficiency.

Finally, we see Kernel Time using Tensor Cores. We'll get back to those later.

Execution Summary

The execution summary gives is an overview of which components took the most time (averaged over the steps in this Span).

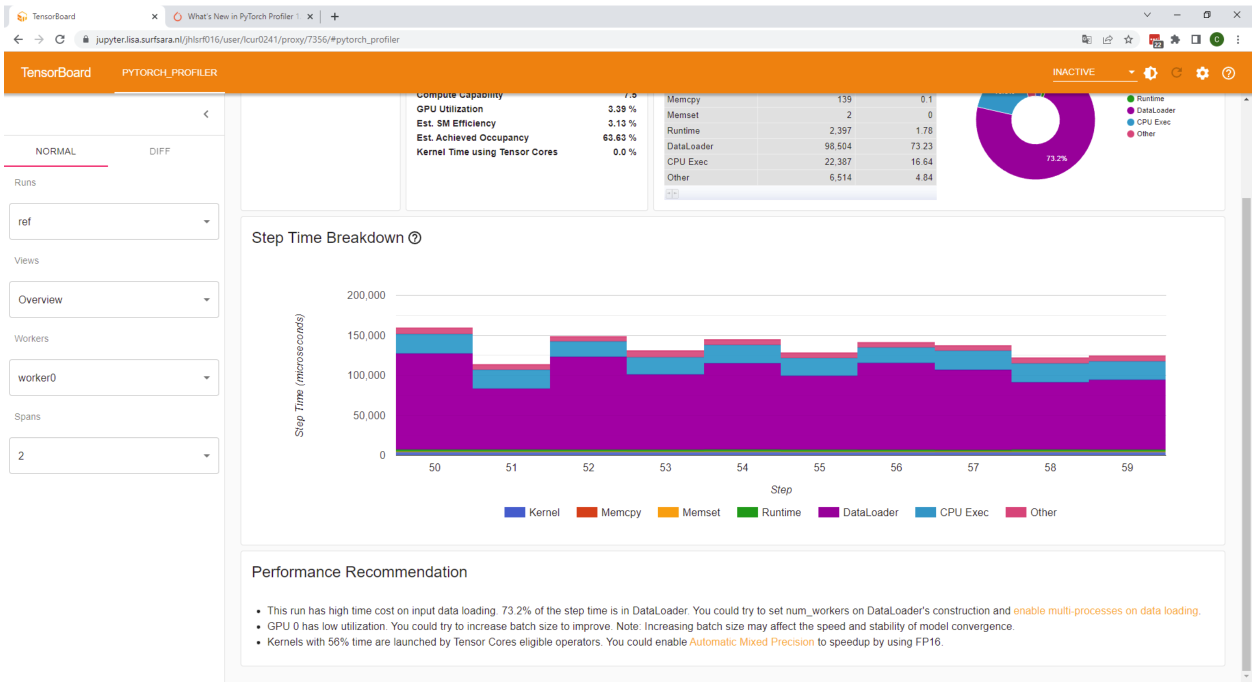

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	134,511	100
Kernel	4,568	3.4
Memcpy	139	0.1
Memset	2	0
Runtime	2,397	1.78
DataLoader	98,504	73.23
CPU Exec	22,387	16.64
Other	6,514	4.84

Here, we see that the average step time was 134 ms. The vast majority of that (98ms) was taken by data loading. Another substantial chunk was CPU execution.

Performance Recommendation

The final part of the overview window is the performance recommendation. Based on the timings measured in the profile, the PyTorch Profiler will try to give you some advice on what to do to speed up your code.



Compute Capability 7.5
GPU Utilization 3.39 %
Est. SM Efficiency 3.13 %
Est. Achieved Occupancy 63.63 %
Kernel Time using Tensor Cores 0.0 %

Memcpy	139	0.1
Memset	2	0
Runtime	2,397	1.78
DataLoader	98,504	73.23
CPU Exec	22,387	16.64
Other	6,514	4.84

Performance Recommendation

- This run has high time cost on input data loading. 73.2% of the step time is in DataLoader. You could try to set num_workers on DataLoader's construction and enable multi-processes on data loading.
- GPU 0 has low utilization. You could try to increase batch size to improve. Note: Increasing batch size may affect the speed and stability of model convergence.
- Kernels with 56% time are launched by Tensor Cores eligible operators. You could enable Automatic Mixed Precision to speedup by using FP16.

Often, these tips are quite decent, though the Profiler doesn't know what you've already done. Also, which piece of advice will help the most might be better judged by you.

2.2.8 Improving I/O performance

Increasing the number of workers for dataloading

We've seen that 73.2% of our time was spent in dataloading, rather than computing anything. Clearly, this is the first problem we want to tackle. Let's start by following the PyTorch profilers advice, and increase the amount of workers (i.e. CPU cores) used for dataloading.

First, let's see how many cores we have available in this environment:

```
[ ]: def get_cpu_count():
         return len(os.sched_getaffinity(0))

     print(f"Number of CPUs: {get_cpu_count()}")
```

Since the bulk of our computation happens on the GPU, we can probably use all of the CPU cores we have available for dataloading (if you would be training on the CPUs as well, it might be more efficient to designate a few CPU cores for dataloading, and the rest for training).

Try setting the NUM_DATALOADER_WORKERS to 3,

```
[ ]: BATCH_SIZE = 128
     EPOCHS = 1
     LEARNING_RATE = 1e-4
     NUM_DATALOADER_WORKERS = 3
     BREAK_AFTER_N_ITERATIONS = 60

     LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

     model = CIFAR10CNN().to(device)
     optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
     ↪(model.parameters(), lr=LEARNING_RATE)

     transform=transforms.Compose([
         transforms.ToTensor(),
         # Normalize the data to 0 mean and 1 standard deviation, now for all channels of RGB
         transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
     ])

     PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
     label_filename = os.path.join(DATA_PATH, "labels.pkl")
     train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
     ↪transform=transform)

     train_loader = torch.utils.data.DataLoader(
             train_dataset,
             batch_size=BATCH_SIZE,
             pin_memory=use_cuda,
             shuffle=True,
             num_workers=NUM_DATALOADER_WORKERS
     )

     test_loader = torch.utils.data.DataLoader(
             datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
```

(continues on next page)

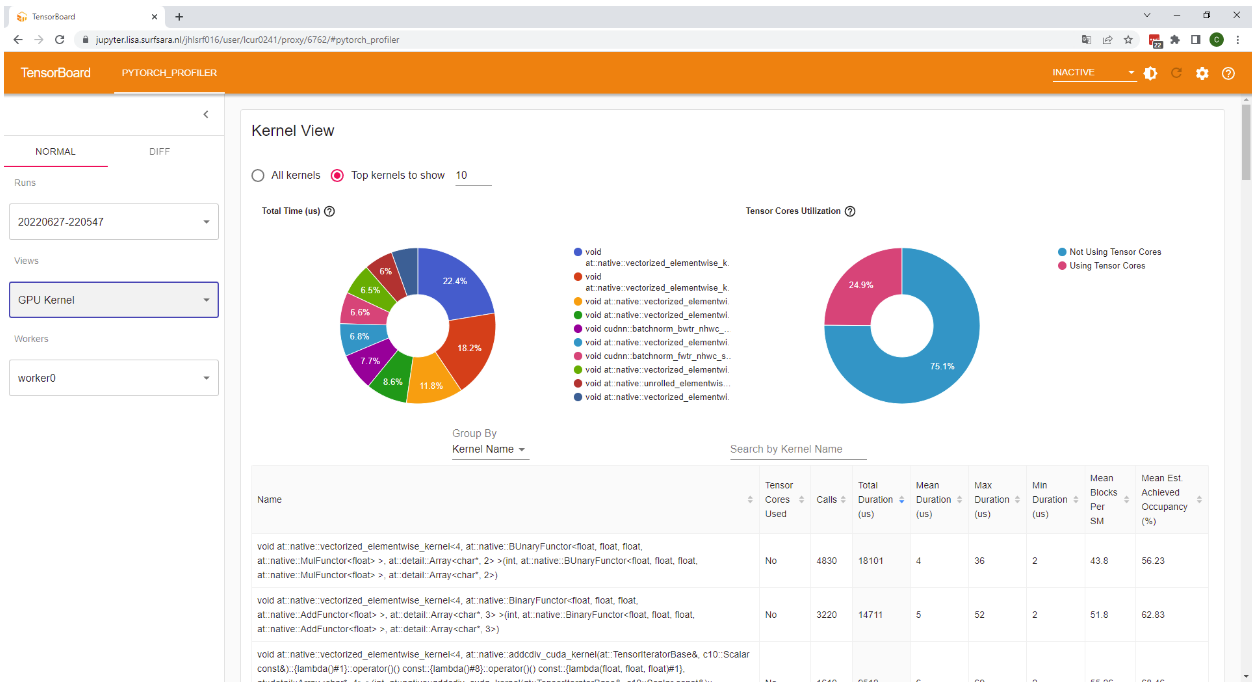
(continued from previous page)

```
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/num_dataloaders/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/num_dataloaders/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting & Interpreting results

You should see something like this:


TensorBoard
jupyter.lisa.surfsara.nl/jhlsrf016/user/lcur0241/proxy/6762/#pytorch_profiler
TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Runs
20220627-220547
Views
GPU Kernel
Workers
worker0
Kernel View
All kernels
Top kernels to show 10
Total Time (us)
22.4%
18.2%
11.8%
8.6%
7.7%
6.8%
6.6%
6.5%
6%
void at::native::vectorized_elementwise_k.
void at::native::vectorized_elementwise_k.
void at::native::vectorized_elementwi.
void at::native::vectorized_elementwi.
void cudnn::batchnorm_bwtr_nhwc_...
void at::native::vectorized_elementwi.
void cudnn::batchnorm_fwtr_nhwc_s..
void at::native::vectorized_elementwi.
void at::native::unrolled_elementwis...
void at::native::vectorized_elementwi.
Tensor Cores Utilization
24.9%
75.1%
Not Using Tensor Cores
Using Tensor Cores
Group By
Kernel Name
Search by Kernel Name

Name	Tensor Cores Used	Calls	Total Duration (us)	Mean Duration (us)	Max Duration (us)	Min Duration (us)	Mean Blocks Per SM	Mean Est. Achieved Occupancy (%)
void at::native::vectorized_elementwise_kernel<4, at::native::BUnaryFunctor<float, float, float, at::native::MulFunctor<float> >, at::detail::Array<char*, 2> >(int, at::native::BUnaryFunctor<float, float, float, at::native::MulFunctor<float> >, at::detail::Array<char*, 2>)	No	4830	18101	4	36	2	43.8	56.23
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float> >, at::detail::Array<char*, 3> >(int, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float> >, at::detail::Array<char*, 3>)	No	3220	14711	5	52	2	51.8	62.83
void at::native::vectorized_elementwise_kernel<4, at::native::addcdiv_cuda_kernel(at::TensorIteratorBase&, c10::Scalar const&)::{lambda()#1}::operator()() const::{lambda()#8}::operator()() const::{lambda(float, float, float)#1},	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

As we can see, the average step time has gone down substantially, to 134 to 39 ms. Note that for codes that are limited by I/O, you might see large variations: we are working on a network filesystem here, meaning that all of us in the course are reading from the same disks. If one of your fellow students is hitting the filesystem at the same time as you are, it might slow down substantially. This variation is also visible between step 51, 54 and 57.

Note that this variation is ***not*** the reason we see so little dataloader time at e.g. step 52 and 53. The data for step 51-53 gets loaded during step 48-50 by the three dataloaders. However, since the computation is faster than the dataloading, the 3 dataloaders are not yet finished when step 51 starts. Thus, step 51 is waiting for the dataloaders to finish before it can start computing, which is what you see as 'DataLoader' time in the profile. Once that I/O is completed, we immediately have ***three*** batches, since we had three dataloaders. That's why step 52 and 53 don't show any Dataloading time anymore.

As we can see, GPU utilization has also gone up to 11% or so. Not great, but still a lot better than before.

Transformations

Transformations are also part of the Dataloading pipeline. However, these are performed on the CPU and that may take substantially more time that the GPU needs to subsequently train the network. One way to speedup the dataIO is to perform the normalization as part of the network itself, so that it can be executed on the GPU. Note that this is not *always* useful: it depends on the balance between the amount of work between the GPU and the CPU. If the dataloading on the CPU can already keep up with the GPU, there's no point in offloading more operations to the GPU. In this case, we have a pretty light network however, and we see if we can shave off a little bit more of the dataloading time.

Below, we redefine the network, with the first layer now being the normalization.

```
[ ]: class CIFAR10CNN(nn.Module):
         def __init__(self):
             super().__init__()

             # 4 convolution layers, with a non-linear activation after each.
             # maxpooling after the activations of the 2nd, 3rd, and 4th conv layers
             # 2 dense layers for classification
```

(continues on next page)

(continued from previous page)

```
        # log_softmax
        #
        # As for the number of channels of each layers, try to experiment!

        self.feature_extractor = nn.Sequential(
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
            nn.Conv2d(in_channels=3, out_channels=8, kernel_size=3, padding=1),
            nn.ReLU(),

            nn.Conv2d(in_channels=8, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )

        # in_features of the first layer should be the product of the output shape of␣
↪your feature extractor!
        # E.g. if the output of your feature extractor has size (batch x 128 x 4 x 4),␣
↪in_features = 128*4*4=2048
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=2048, out_features=2048),
            nn.ReLU(),
            nn.Linear(in_features=2048, out_features=10),
            nn.LogSoftmax(dim=1)
        )


    def forward(self, x):
        features = self.feature_extractor(x)

        return self.classifier(features)
```

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
BREAK_AFTER_N_ITERATIONS = 60

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

model = CIFAR10CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)
```

(continues on next page)

(continued from previous page)

```
transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalization now done in the network
    # transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])


PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/transforms/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

```
[ ]: %%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/transforms/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false
```

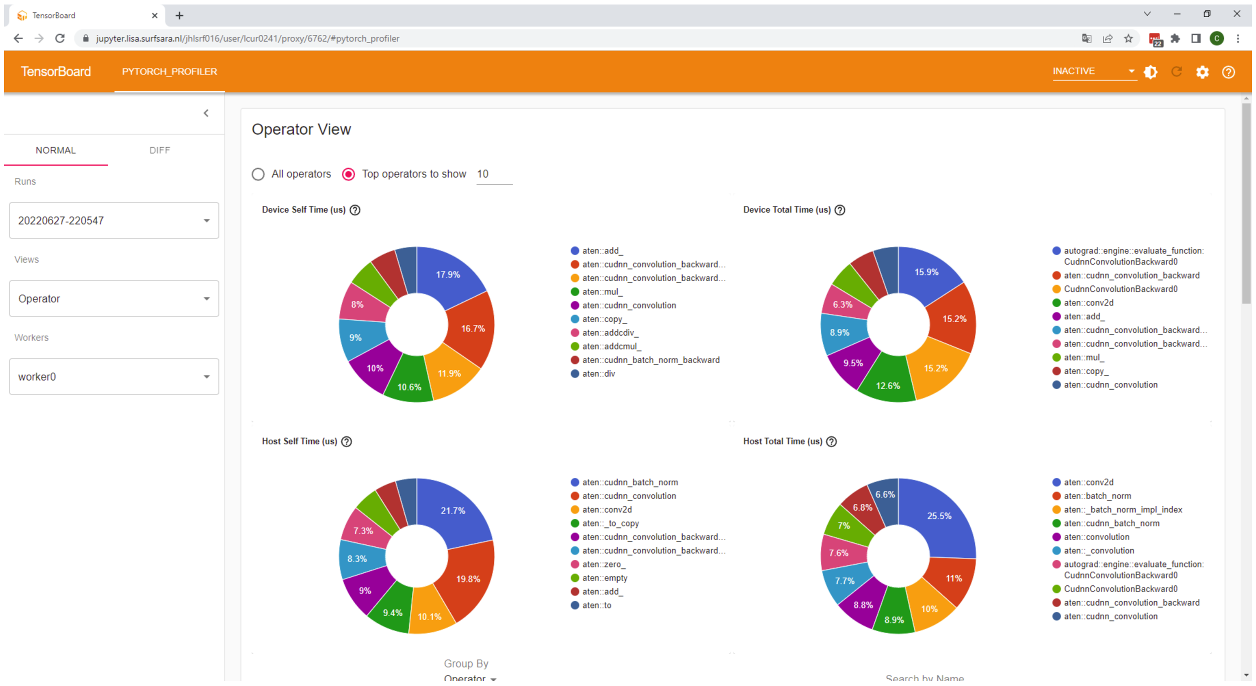
(continues on next page)

(continued from previous page)

```
print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting and interpreting results

You should see something like this



We've managed to reduce step time to 33s now: not a huge gain, but the dataloading part is only 1.9s per step now on average. So, there's not much more to gain. You may see a peak at some steps. If you're eager, you can try to increase the `prefetch_factor`, which is an additional argument you can pass to the `torch.utils.data.DataLoader`. This means more samples will be loaded in advance, and you're less sensitive to fluctuations in I/O speed. However, it will increase (CPU) memory consumption, as you'll need to hold more samples in (CPU) memory. If you feel up to it, you can add it in the code above, rerun the profiling and start a new TensorBoard.

Final remarks on optimizing I/O

In this case, we've managed to reduce the data loading overhead to almost zero. The reason is that the dataset we work on in this tutorial is tiny: 198 MB in total. This is simply cached by our network filesystem, and therefore I/O is very fast.

In general however, the way this dataset is stored, as one file per sample (image), is ***very bad*** for performance. This is because filesystems need to read metadata (where is the file located, how big is it, updating the last access date, etc) for every individual file. On average, each cifar10 image is about 2KB. Imagine how large the overhead of reading overhead (metadata) for such a small file! This is not a huge issue on e.g. a local laptop with an SSD, since since you're the only one using that disk and since SSDs can do metadata operations very quickly. On large clusters however, metadata operations are (comparatively speaking) very slow. Large clusters typically use network filesystems that are optimized for throughput, i.e. bandwidth, but can process relatively modest amounts of files per second.

Thus, generally, it is advised to pack samples together in a packed file format. It doesn't hurt on your local laptop, and is all-but-essential on large clusters. Many packed file formats are available, examples are LMDB, HDF5/h5py, TFRecords, petastorm, or even zip or tarballs (but don't compress, since uncompressing takes time too!). One thing to note is that you'll want a packed file format where you can read a *single* sample without having to load the whole file in memory.

In some clusters, the nodes have local storage. In that case, an option to increase the performance is to first copy the data to local storage. But: also these data copies are a lot faster if you copy a single file, rather than a large amount of small files. So even in those cases, packed file formats are benificial.

2.2.9 New baseline: a heavier model

Of course, in this tutorial, we've been mostly playing with a toy model. Realistic models are a lot heavier to train. Let's try a real RESNET50 model. Torchvision has such a model defined by default. Let's use it to set a new baseline for our training run.

```
[ ]: model = models.resnet50()
     print(model)
```

Because this code produces much larger trace files, we decrease the number of active iterations, and stop the training after 16 iterations (i.e. one 'span'). Otherwise, the trace files will be very slow to analyze.

```
[ ]: # This is the actual train loop we will use for profiling
     def train_profiling(model, device, train_loader, optimizer, epoch, log_interval, logdir,␣
     ↪break_idx):
         model.train()

         # Create a torch.profiler.profile object, and call it as the last part of the␣
     ↪training loop
         with torch.profiler.profile(
         activities=[
             torch.profiler.ProfilerActivity.CPU,
             torch.profiler.ProfilerActivity.CUDA],
         schedule=torch.profiler.schedule(
             wait=10,
             warmup=3,
             active=3),
         on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir, worker_name='worker0
     ↪'),
         record_shapes=True,
         profile_memory=True,  # This will take 1 to 2 minutes. Setting it to False could␣
     ↪greatly speedup.
         with_stack=True
     ) as p:
             for batch_idx, (data, target) in enumerate(train_loader):
                 # move data and target to the gpu, if available and used
                 data, target = map(lambda tensor: tensor.to(device, non_blocking=True),␣
     ↪(data, target))

                 optimizer.zero_grad()
                 output = model(data)
                 loss = F.nll_loss(output, target)
                 loss.backward()
```

(continues on next page)

(continued from previous page)

```
            optimizer.step()

            accuracy = metrics.accuracy(output, target)

            if batch_idx % log_interval == 0:
                print(
                    f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                    f'\tLoss: {loss.detach().item():.6f}'
                    f'\tAccuracy: {accuracy.detach().item():.2f}'
                )

            yield loss.detach().item(), accuracy.detach().item()

            p.step()

            # Allow to break early for the purpose of shorter profiling
            if (break_idx is not None) and (batch_idx == break_idx):
                break
```

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
BREAK_AFTER_N_ITERATIONS = 16

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

# Now use resnet50
model = models.resnet50()
model = model.to(device, memory_format=torch.channels_last)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalization now done in the network
    # transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])


PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
```

(continues on next page)

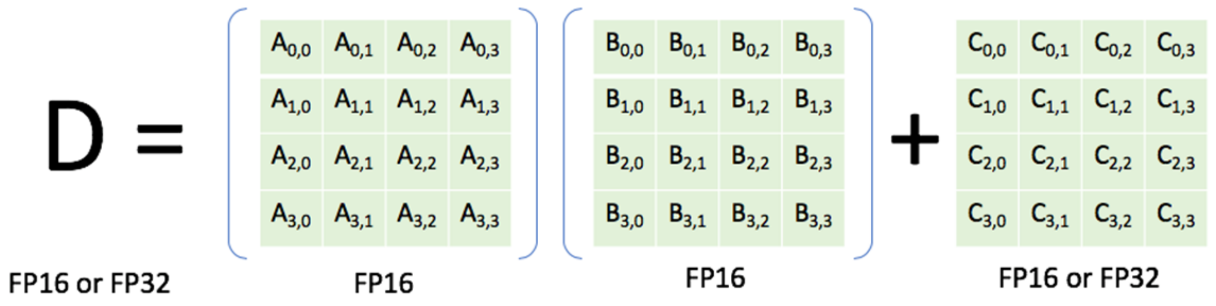
(continued from previous page)

```
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/resnet50_baseline/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

```
[ ]: %%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/resnet50_baseline/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')

# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting and interpreting results

Let’s have a look at our new baseline:

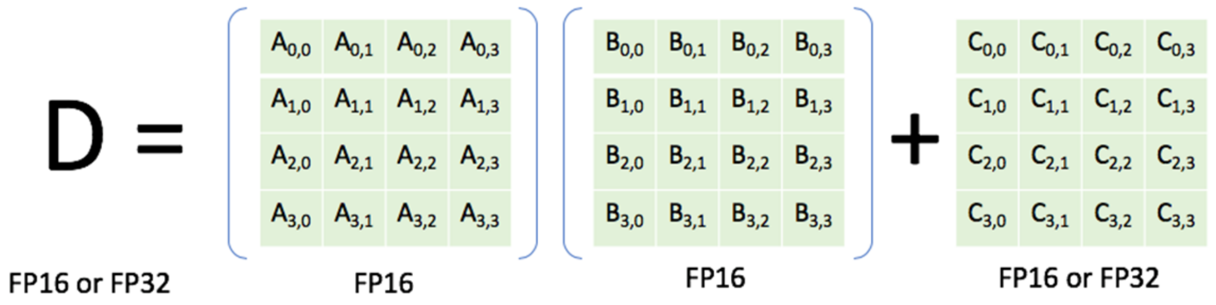


As we can see, a step now takes 229 ms, out of which 37s are spend on GPU kernel execution.

Improving GPU kernel execution time by mixed precision

Cores on an Nvidia GPU have a dedicated processing unit for processing tensor operations, so-called "Tensor Cores". These can do operations on tensors much faster than when those tensors would have to be performed with the traditional FP32 units in the core.

The operation that these Tensor Cores perform is called a fused-multiply-add, and they do it on a (small) matrix. They can do one of these operations every clock cycle of the GPU:



There are some requirements though. On the GPUs that we are working on in this training, the tensor cores only support operations where tensors A and B are stored as 16-bit precision floating point numbers. Tensor C and D can be either 16 or 32 bit. Torch has a specific feature called "automatic mixed precision" (AMP for short), which is supported through the `torch.cuda.amp` package (see this documentation). In principle, this can be used to automatically convert the right tensors in the model to 16-bit floating point, so that the Tensor Cores can be used. Note however that you may need to 'scale' your gradients: FP16 floating point numbers have a smaller representable range (since they have fewer bits to represent the number), which may cause underflows in your gradients (i.e. numbers in your gradient may become smaller than the smallest number that can be represented with the FP16 datatype). If that happens, you need to do gradient scaling using the `torch.cuda.amp.GradScaler`. In the below example, we exlcuded that, since here we just want to demonstrate the potential speedup.

```
[ ]: from torch import autocast

     # This is the actual train loop we will use for profiling
     def train_profiling(model, device, train_loader, optimizer, epoch, log_interval, logdir,␣
     ↪break_idx=None):
         model.train()

         # Create a torch.profiler.profile object, and call it as the last part of the␣
     ↪training loop
```

(continues on next page)

(continued from previous page)

```
    with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=10,
        warmup=3,
        active=3),
    on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir, worker_name='worker0
↪'),
    record_shapes=True,
    profile_memory=True,  # This will take 1 to 2 minutes. Setting it to False could␣
↪greatly speedup.
    with_stack=True
) as p:
        for batch_idx, (data, target) in enumerate(train_loader):
            # move data and target to the gpu, if available and used
            data = data.to(device, non_blocking=True, memory_format=torch.channels_last)
            target = target.to(device, non_blocking=True)

            optimizer.zero_grad()

            # Here, we use automatic mixed precision:
            with autocast(device_type='cuda'):
                output = model(data)
                loss = F.nll_loss(output, target)

            loss.backward()
            optimizer.step()

            accuracy = metrics.accuracy(output, target)

            if batch_idx % log_interval == 0:
                print(
                    f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.
↪dataset)} ({100 * batch_idx / len(train_loader):.0f}%)]'
                    f'\tLoss: {loss.detach().item():.6f}'
                    f'\tAccuracy: {accuracy.detach().item():.2f}'
                )

            yield loss.detach().item(), accuracy.detach().item()

            p.step()

            # Allow to break early for the purpose of shorter profiling
            if (break_idx is not None) and (batch_idx == break_idx):
                break
```

[ ]:
```
BATCH_SIZE = 128
EPOCHS = 1
LEARNING_RATE = 1e-4
NUM_DATALOADER_WORKERS = 3
```

(continues on next page)

(continued from previous page)

```
BREAK_AFTER_N_ITERATIONS = 16

LOGGING_INTERVAL = 10  # Controls how often we print the progress bar

# Now use resnet50
model = models.resnet50()
model = model.to(device, memory_format=torch.channels_last)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE) # optim.<OPTIMIZER_FLAVOUR>
↪(model.parameters(), lr=LEARNING_RATE)

transform=transforms.Compose([
    transforms.ToTensor(),
    # Normalization now done in the network
    # transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])


PNG_DATA = os.path.join(DATA_PATH, 'cifar10_png')
label_filename = os.path.join(DATA_PATH, "labels.pkl")
train_dataset = Cifar10PNGDataset(label_file = label_filename, img_dir = PNG_DATA,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=True,
        num_workers=NUM_DATALOADER_WORKERS
)

test_loader = torch.utils.data.DataLoader(
        datasets.CIFAR10(DATA_PATH, train=False, transform=transform, download=True),
        batch_size=BATCH_SIZE,
        pin_memory=use_cuda,
        shuffle=False,
        num_workers=NUM_DATALOADER_WORKERS
)

logdir = "logs/autocast/" + datetime.now().strftime("%Y%m%d-%H%M%S")

fit_profiling(model, optimizer, EPOCHS, device, train_loader, test_loader, LOGGING_
↪INTERVAL, logdir, BREAK_AFTER_N_ITERATIONS)
```

[ ]:
```
%%capture --no-stdout

# REFERENCE PROFILE:
# logdir = os.path.join(DATA_PATH,"logs/autocast/ref")

import random, os

rand_port = random.randint(6000, 8000)
username = os.getenv('USER')
```

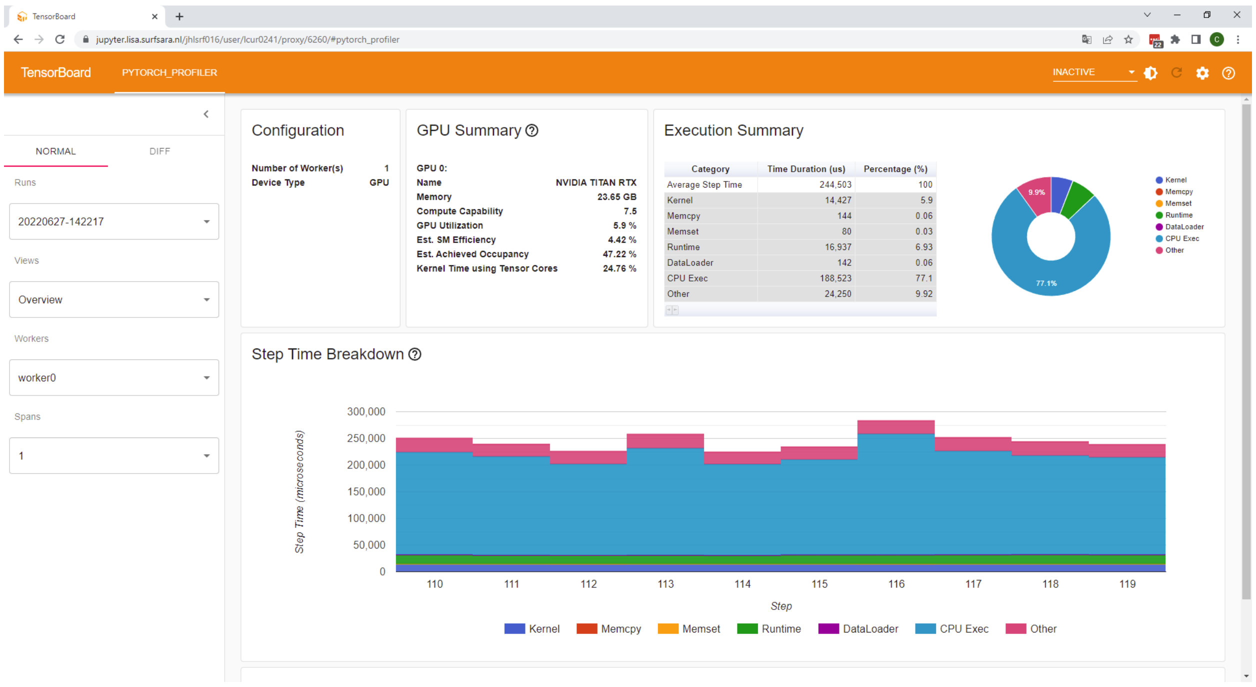
(continues on next page)

(continued from previous page)

```
# Kill old TensorBoard sessions
!kill -9 $(pgrep -u $username -f "multiprocessing-fork")
!kill -9 $(pgrep -u $username tensorboard)

%reload_ext tensorboard
# Run with --load_fast=false, since current TensorBoard version has an issue with the␣
↪profiler plugin
# (more info https://github.com/tensorflow/tensorboard/issues/4784)
%tensorboard --logdir=$logdir --port=$rand_port --load_fast=false

print(f"Go to https://jupyter.lisa.surfsara.nl/jhlsrf016/user/{username}/proxy/{rand_
↪port}/#pytorch_profiler")
```

Inspecting and interpreting results

You should see something like this



Configuration

Number of Worker(s)	1
Device Type	GPU

GPU Summary

GPU 0:	
Name	NVIDIA TITAN RTX
Memory	23.65 GB
Compute Capability	7.5
GPU Utilization	5.9 %
Est. SM Efficiency	4.42 %
Est. Achieved Occupancy	47.22 %
Kernel Time using Tensor Cores	24.76 %

Execution Summary

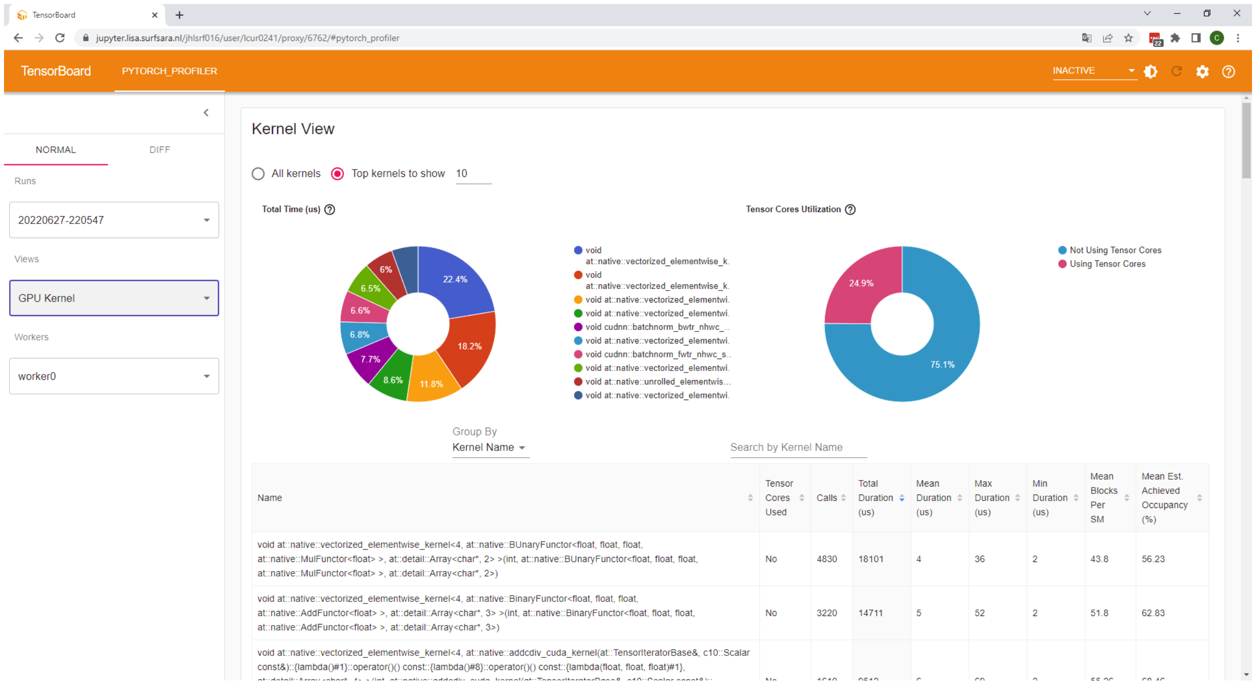
Category	Time Duration (us)	Percentage (%)
Average Step Time	244,503	100
Kernel	14,427	5.9
Memcpy	144	0.06
Memset	80	0.03
Runtime	16,937	6.93
DataLoader	142	0.06
CPU Exec	188,523	77.1
Other	24,250	9.92

We'll focus on the GPU kernel execution time here. It has decreased from about 37 to 14 ms, and the GPU summary now tells us that almost 25% of the time spent on executing kernels, it is using the Tensor Cores. You might have noticed that GPU utilization has gone *down* in this case, compared to our baseline run. However, here, that's not a bad thing: we have essentially reduced the *amount of work* that needed to be done by moving to reduced precision.

What *is* problematic though is that the CPU Exec time has gone up. While we haven't confirmed this, we think it is due to the extra overhead introduced by the type conversion from FP32 to FP16. So, for this particular network and (small) input, it is not worth it to use mixed precision - but in many real world cases it does provide a substantial speedup!

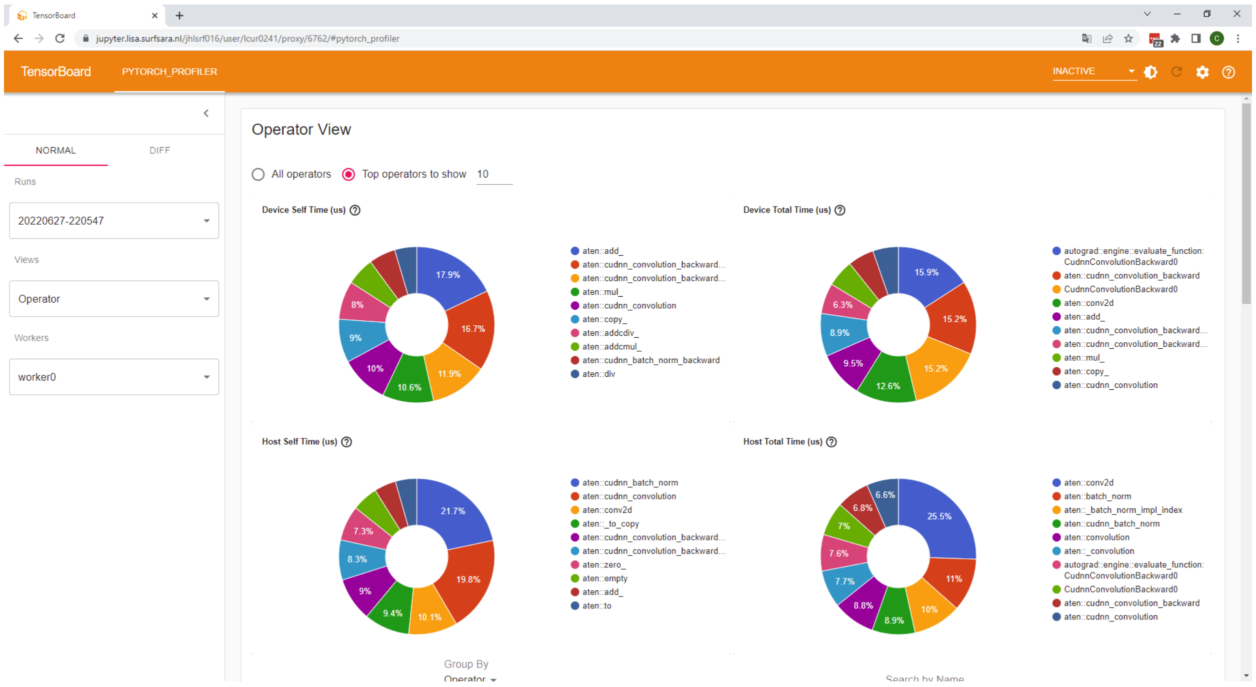
2.2.10 Other views of the profiler

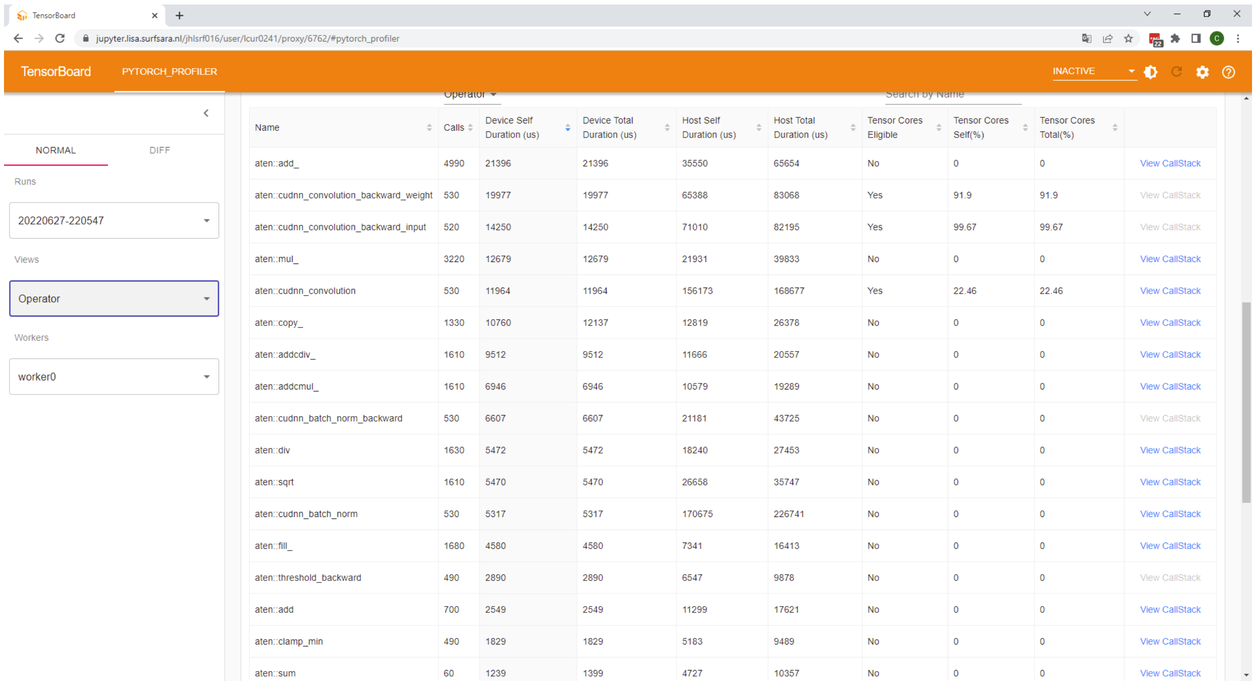
Here, we'll just discuss some of the other views of the profiler, without exercises.


TensorBoard
jupyter.lisa.surfsara.nl/jhlsrf016/user/lcur0241/proxy/6762/#pytorch_profiler
TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Runs
20220627-220547
Views
GPU Kernel
Workers
worker0
Kernel View
All kernels
Top kernels to show
10
Total Time (us)
22.4%
18.2%
11.8%
8.6%
7.7%
6.8%
6.6%
6.5%
6%
void at::native::vectorized_elementwise_k.
void at::native::vectorized_elementwise_k.
void at::native::vectorized_elementwi.
void at::native::vectorized_elementwi.
void cudnn::batchnorm_bwtr_nhwc_...
void at::native::vectorized_elementwi.
void cudnn::batchnorm_fwtr_nhwc_s..
void at::native::vectorized_elementwi.
void at::native::unrolled_elementwis...
void at::native::vectorized_elementwi.
Tensor Cores Utilization
24.9%
75.1%
Not Using Tensor Cores
Using Tensor Cores
Group By
Kernel Name
Search by Kernel Name

Name	Tensor Cores Used	Calls	Total Duration (us)	Mean Duration (us)	Max Duration (us)	Min Duration (us)	Mean Blocks Per SM	Mean Est. Achieved Occupancy (%)
void at::native::vectorized_elementwise_kernel<4, at::native::BUnaryFunctor<float, float, float, at::native::MulFunctor<float> >, at::detail::Array<char*, 2> >(int, at::native::BUnaryFunctor<float, float, float, at::native::MulFunctor<float> >, at::detail::Array<char*, 2>)	No	4830	18101	4	36	2	43.8	56.23
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float> >, at::detail::Array<char*, 3> >(int, at::native::BinaryFunctor<float, float, float, at::native::AddFunctor<float> >, at::detail::Array<char*, 3>)	No	3220	14711	5	52	2	51.8	62.83
void at::native::vectorized_elementwise_kernel<4, at::native::addcdiv_cuda_kernel(at::TensorIteratorBase&, c10::Scalar const&)::{lambda()#1}::operator()() const::{lambda()#8}::operator()() const::{lambda(float, float, float)#1},	No							

In the GPU kernel view, you see a list of all GPU kernels that have been executed. You can see a lot of details, like how often these particular kernels have been run, how much time that took, if the kernel ran on tensor cores or not, and things like mean estimated achieved occupancy (we have talked about this metric before when discussing the GPU summary). It's hard to gain actionable information from this view.




TensorBoard
jupyter.lisa.surfsara.nl/jhlsrf016/user/lcur0241/proxy/6762/#pytorch_profiler
TensorBoard
PYTORCH_PROFILER
INACTIVE
NORMAL
DIFF
Search by Name

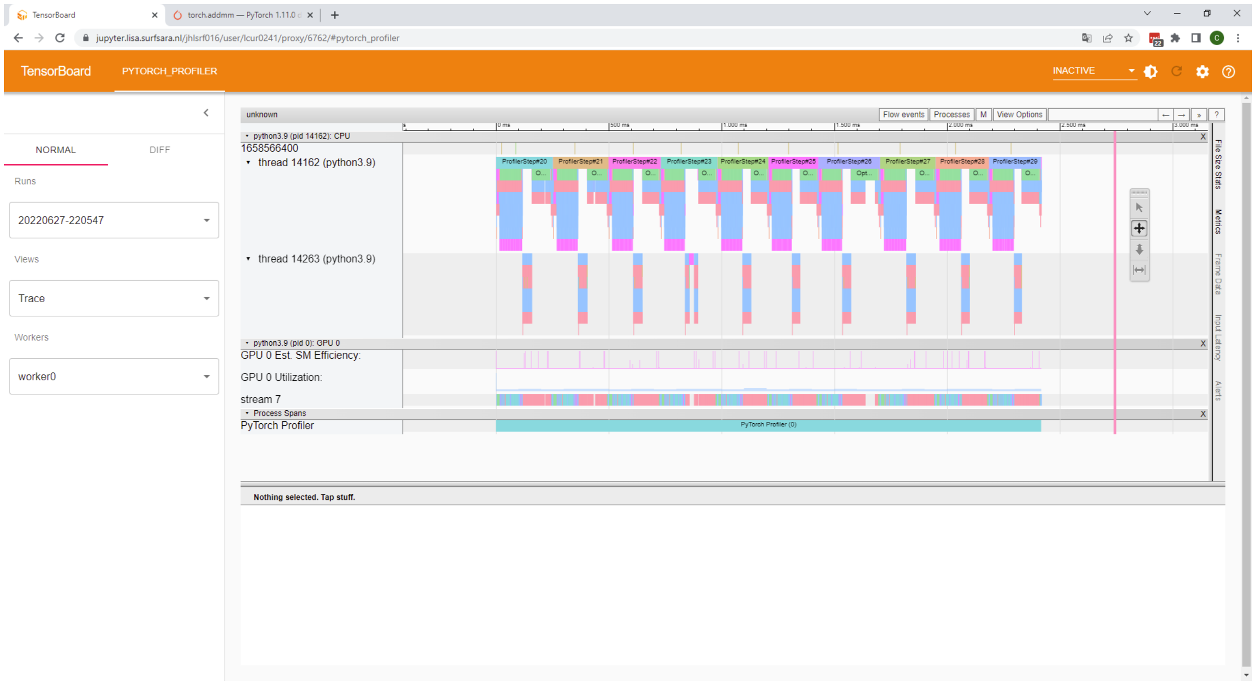
Runs: 20220627-220547

Views: Operator

Workers: worker0

Name	Calls	Device Self Duration (us)	Device Total Duration (us)	Host Self Duration (us)	Host Total Duration (us)	Tensor Cores Eligible	Tensor Cores Self(%)	Tensor Cores Total(%)	
aten::add_	4990	21396	21396	35550	65654	No	0	0	View CallStack
aten::cudnn_convolution_backward_weight	530	19977	19977	65388	83068	Yes	91.9	91.9	View CallStack
aten::cudnn_convolution_backward_input	520	14250	14250	71010	82195	Yes	99.67	99.67	View CallStack
aten::mul_	3220	12679	12679	21931	39833	No	0	0	View CallStack
aten::cudnn_convolution	530	11964	11964	156173	168677	Yes	22.46	22.46	View CallStack
aten::copy_	1330	10760	12137	12819	26378	No	0	0	View CallStack
aten::addcdiv_	1610	9512	9512	11666	20557	No	0	0	View CallStack
aten::addcmul_	1610	6946	6946	10579	19289	No	0	0	View CallStack
aten::cudnn_batch_norm_backward	530	6607	6607	21181	43725	No	0	0	View CallStack
aten::div	1630	5472	5472	18240	27453	No	0	0	View CallStack
aten::sqrt	1610	5470	5470	26658	35747	No	0	0	View CallStack
aten::cudnn_batch_norm	530	5317	5317	170675	226741	No	0	0	View CallStack
aten::fill_	1680	4580	4580	7341	16413	No	0	0	View CallStack
aten::threshold_backward	490	2890	2890	6547	9878	No	0	0	View CallStack
aten::add	700	2549	2549	11299	17621	No	0	0	View CallStack
aten::clamp_min	490	1829	1829	5183	9489	No	0	0	View CallStack
aten::sum	60	1239	1399	4727	10357	No	0	0	View CallStack

The operator view shows all PyTorch operations that have been executed. The question marks next to the pie graphs give some additional explaination about what you see there. Similarly, though it gives some interesting insights into which operators take the most time in your computation, its hard to gain actionable insight from this.

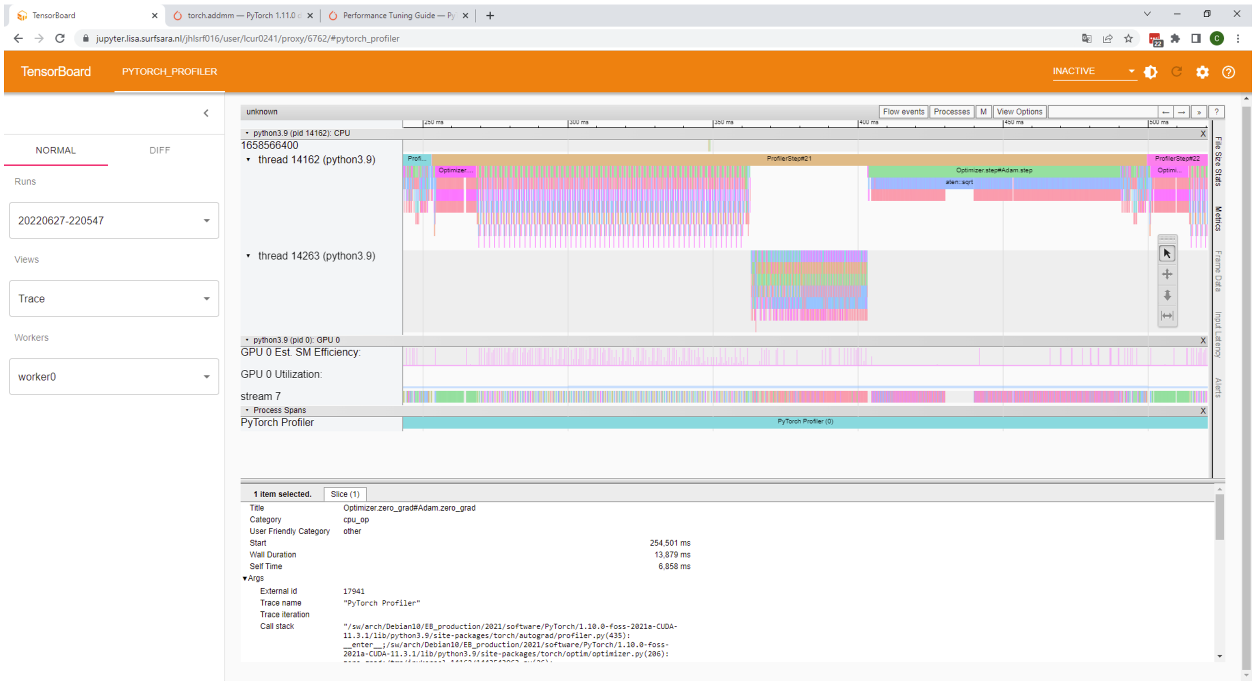


Runs: 20220627-220547

Views: Trace

Workers: worker0

In the trace view, we see a timeline of the execution of the training iterations. At the top, we clearly recognize the 10 iterations that we profiled. Let's zoom in on one step:



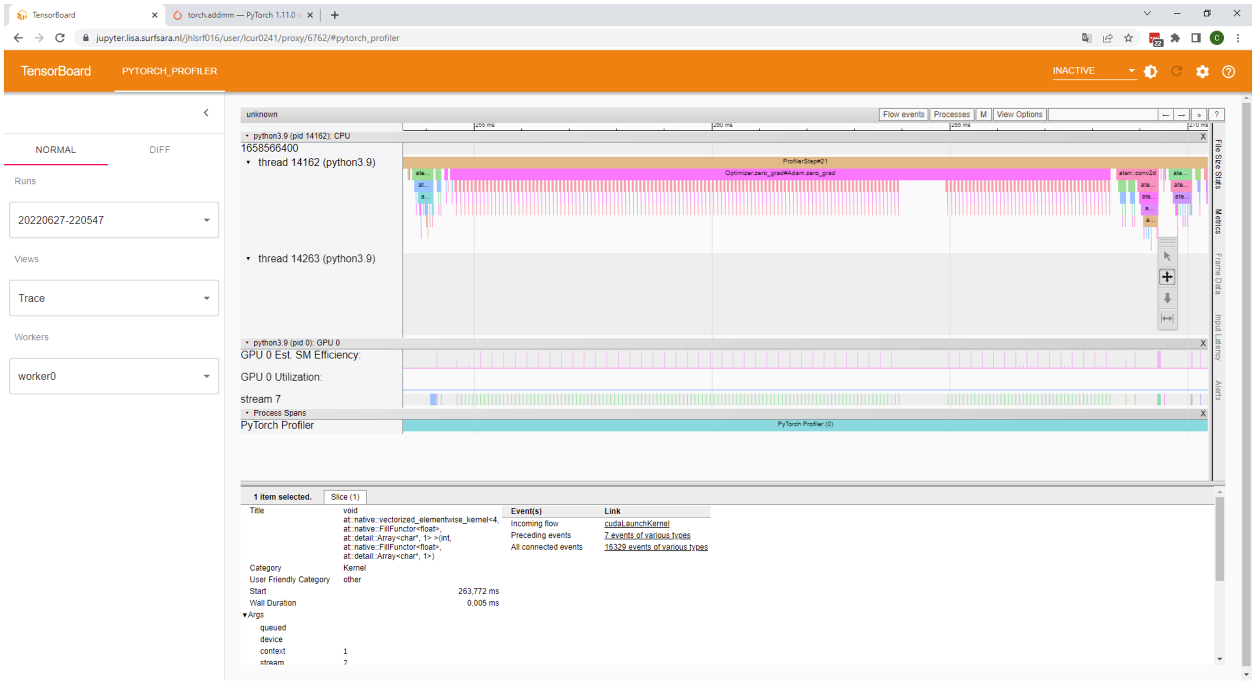
1 item selected. Slice (1)

Title	Optimizer.zero_grad#Adam.zero_grad
Category	cpu_op
User Friendly Category	other
Start	254,501 ms
Wall Duration	13,879 ms
Self Time	6,858 ms

▾Args

External id	17941
Trace name	"PyTorch Profiler"
Trace iteration	
Call stack	"/sw/arch/Debian10/EB_production/2021/software/PyTorch/1.10.0-foss-2021a-CUDA-11.3.1/lib/python3.9/site-packages/torch/autograd/profiler.py(435): __enter__;/sw/arch/Debian10/EB_production/2021/software/PyTorch/1.10.0-foss-2021a-CUDA-11.3.1/lib/python3.9/site-packages/torch/optim/optimizer.py(206):

The two 'threads' at the top show the processes that execute on the CPU. The 'stream 7' shows the kernels that execute on the GPU. You can click on certain operations to see what they are, and how long they took. For example, the purple block at the top left that says 'Optimizer…' is now selected. It shows that the full title is 'Optmizer.zero_grad', i.e. this is the process that fills the gradients with zeroes. Let's zoom in on that:



1 item selected. Slice (1)

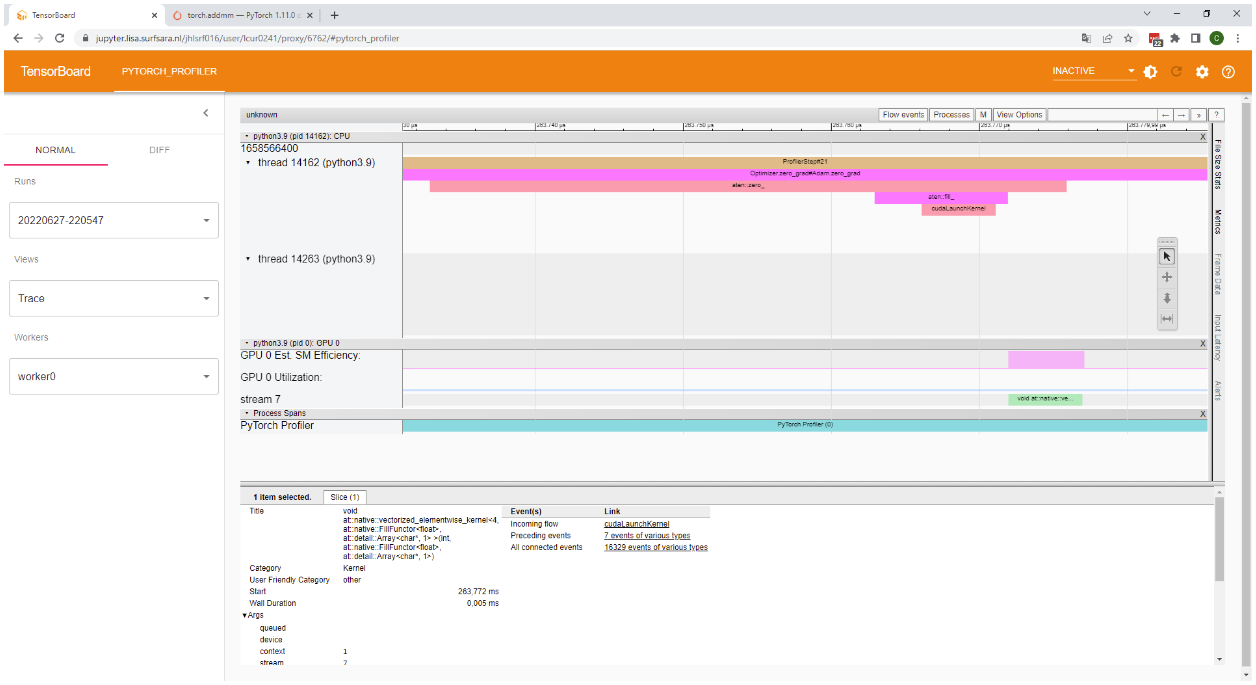
Title	void at::native::vectorized_elementwise_kernel<4, at::native::FillFunctor<float>, at::detail::Array<char*, 1> >(int, at::native::FillFunctor<float>, at::detail::Array<char*, 1>)
Category	Kernel
User Friendly Category	other
Start	263,772 ms
Wall Duration	0,005 ms

▾Args

queued	
device	
context	1
stream	7

Event(s)	Link
Incoming flow	cudaLaunchKernel
Preceding events	7 events of various types
All connected events	16329 events of various types

We see some pattern being repeated. Zooming in further, we see:



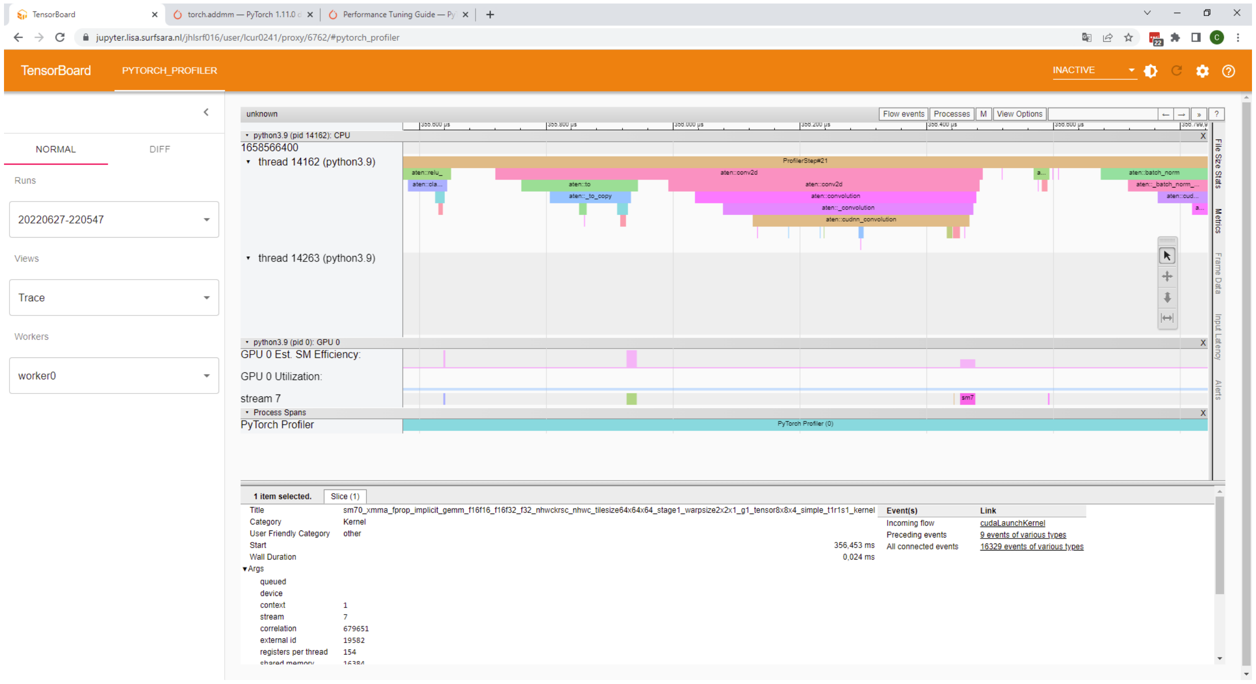
1 item selected. Slice (1)

		Event(s)	Link
Title	void at::native::vectorized_elementwise_kernel<4, at::native::FillFunctor<float>, at::detail::Array<char*, 1> >(int, at::native::FillFunctor<float>, at::detail::Array<char*, 1>)	Incoming flow	cudaLaunchKernel
		Preceding events	7 events of various types
		All connected events	16329 events of various types
Category	Kernel		
User Friendly Category	other		
Start	263,772 ms		
Wall Duration	0,005 ms		
▾Args			
queued			
device			
context	1		
stream	7		

Essentially, you see a call stack here. `Optmizer.zero_grad` called `aten::zero_`, which called `aten::fill_`, which called `cudaLaunchKernel`, which in turn launced a cuda kernel (the green block in stream 7).

While this process doesn't take a *ton* of time, it is mentioned in the PyTorch tuning guide as one of the things that could be optimized: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html . It is mentioned that one can set the parameters to 'None' instead of to zero explicitely, which may be more efficient, but might incur slightly different numerical behaviour.

Zooming in on another part of the step, we see e.g. the 2D convolutions from the forward pass:

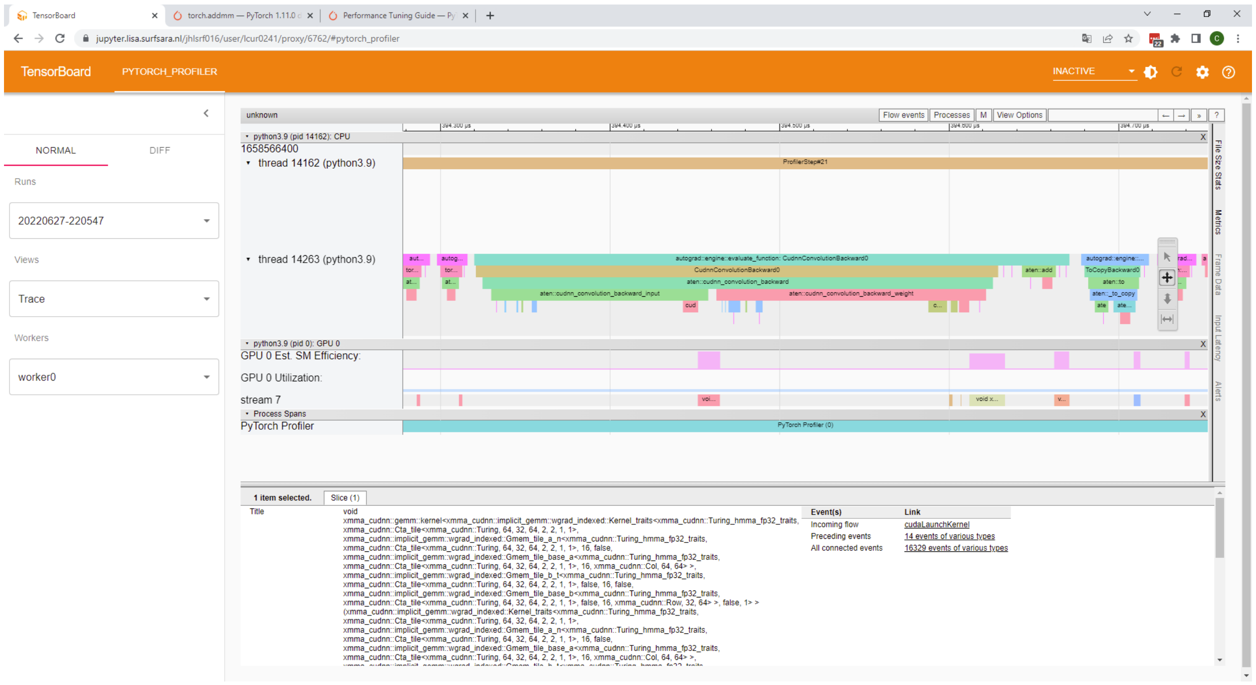


1 item selected. Slice (1)

		Event(s)	Link
Title	sm70_xmma_fprop_implicit_gemm_f16f16_f16f32_f32_nhwckrsc_nhwc_tilesize64x64x64_stage1_warpsize2x2x1_g1_tensor8x8x4_simple_t1r1s1_kernel	Incoming flow	cudaLaunchKernel
Category	Kernel	Preceding events	9 events of various types
User Friendly Category	other	All connected events	16329 events of various types
Start	356,453 ms		
Wall Duration	0,024 ms		
▾Args			
queued			
device			
context	1		
stream	7		
correlation	679651		
external id	19582		
registers per thread	154		
shared memory	16384		

Like before, you can recognize the CPU call stack for the aten::conv2d operation, which eventually results in the laucnh of a kernel on the GPU.

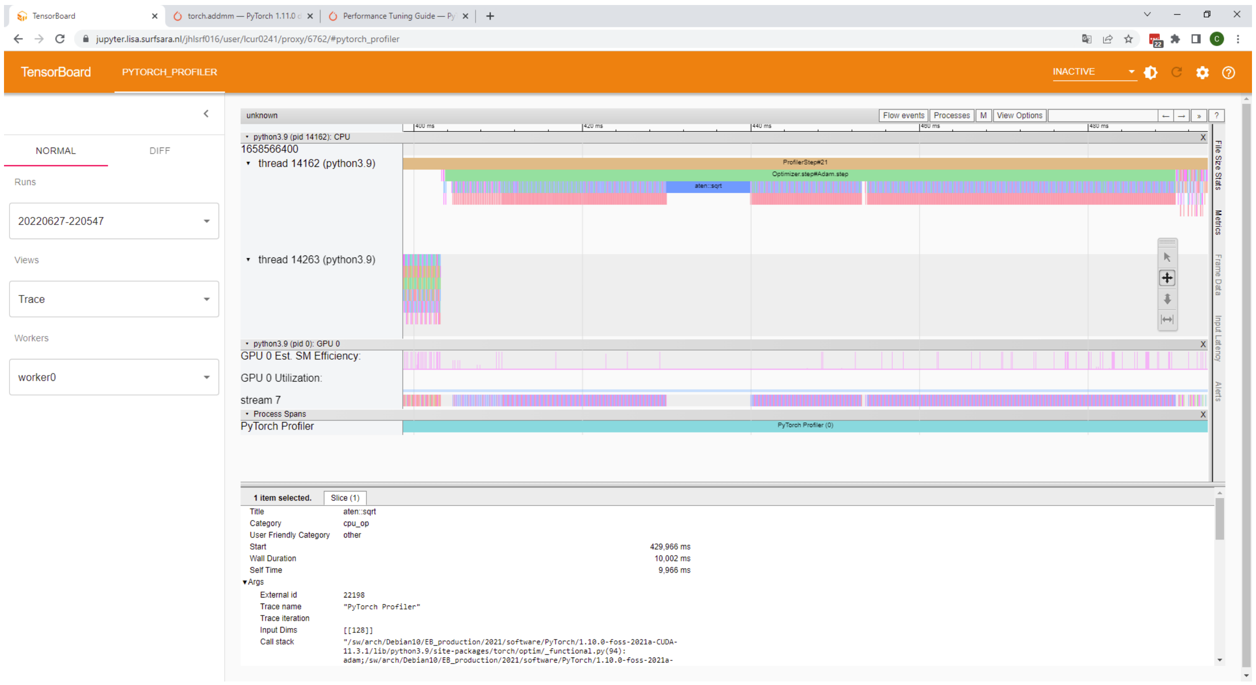
Zooming in on another part, we see some typical part of the backward pass:



```
1 item selected.    Slice (1)

Title    void
         xmma_cudnn::gemm::kernel<xmma_cudnn::implicit_gemm::wgrad_indexed::Kernel_traits<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>,
         xmma_cudnn::implicit_gemm::wgrad_indexed::Gmem_tile_a_n<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>, 16, false,
         xmma_cudnn::implicit_gemm::wgrad_indexed::Gmem_tile_base_a<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>, 16, xmma_cudnn::Col, 64, 64> >,
         xmma_cudnn::implicit_gemm::wgrad_indexed::Gmem_tile_b_t<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>, false, 16, false,
         xmma_cudnn::implicit_gemm::wgrad_indexed::Gmem_tile_base_b<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>, false, 16, xmma_cudnn::Row, 32, 64> >, false, 1> >
         (xmma_cudnn::implicit_gemm::wgrad_indexed::Kernel_traits<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>,
         xmma_cudnn::implicit_gemm::wgrad_indexed::Gmem_tile_a_n<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>, 16, false,
         xmma_cudnn::implicit_gemm::wgrad_indexed::Gmem_tile_base_a<xmma_cudnn::Turing_hmma_fp32_traits,
         xmma_cudnn::Cta_tile<xmma_cudnn::Turing, 64, 32, 64, 2, 2, 1, 1>, 16, xmma_cudnn::Col, 64, 64> >,
```

Event(s)	Link
Incoming flow	cudaLaunchKernel
Preceding events	14 events of various types
All connected events	16329 events of various types

Finally, there's the optimizer step, which will do things like updating the weights:



```
1 item selected.    Slice (1)

Title                   aten::sqrt
Category                cpu_op
User Friendly Category  other
Start                                                  429,966 ms
Wall Duration                                           10,002 ms
Self Time                                                9,966 ms
▾Args
  External id           22198
  Trace name            "PyTorch Profiler"
  Trace iteration
  Input Dims            [[128]]
  Call stack            "/sw/arch/Debian10/EB_production/2021/software/PyTorch/1.10.0-foss-2021a-CUDA-
                        11.3.1/lib/python3.9/site-packages/torch/optim/_functional.py(94):
                        adam;/sw/arch/Debian10/EB_production/2021/software/PyTorch/1.10.0-foss-2021a-
```