Technion - Reinforcement Learning Resarch Labs - RL^2

Tal Daniel

Tutorial - Maximizing CPU and GPU Utilization in PyTorch

Introduction

- In this tutorial we will provide tips and tricks for efficient code execution on CPU and GPU with PyTorch.
- The main goal: an optimized code that runs faster.
- We wish to maximize the utility of the GPU, and allow efficient transfer of data between CPU-GPU.
- Here we provide general directions, and you can refer to the link in each section for more detailed examples.

Agenda

- Monitoring Utilization
- General Tips
- Data Loading Tips to Maximize GPU Utility
- Training Tips
- Reinforcement Learning
- Recommended Videos
- Credits

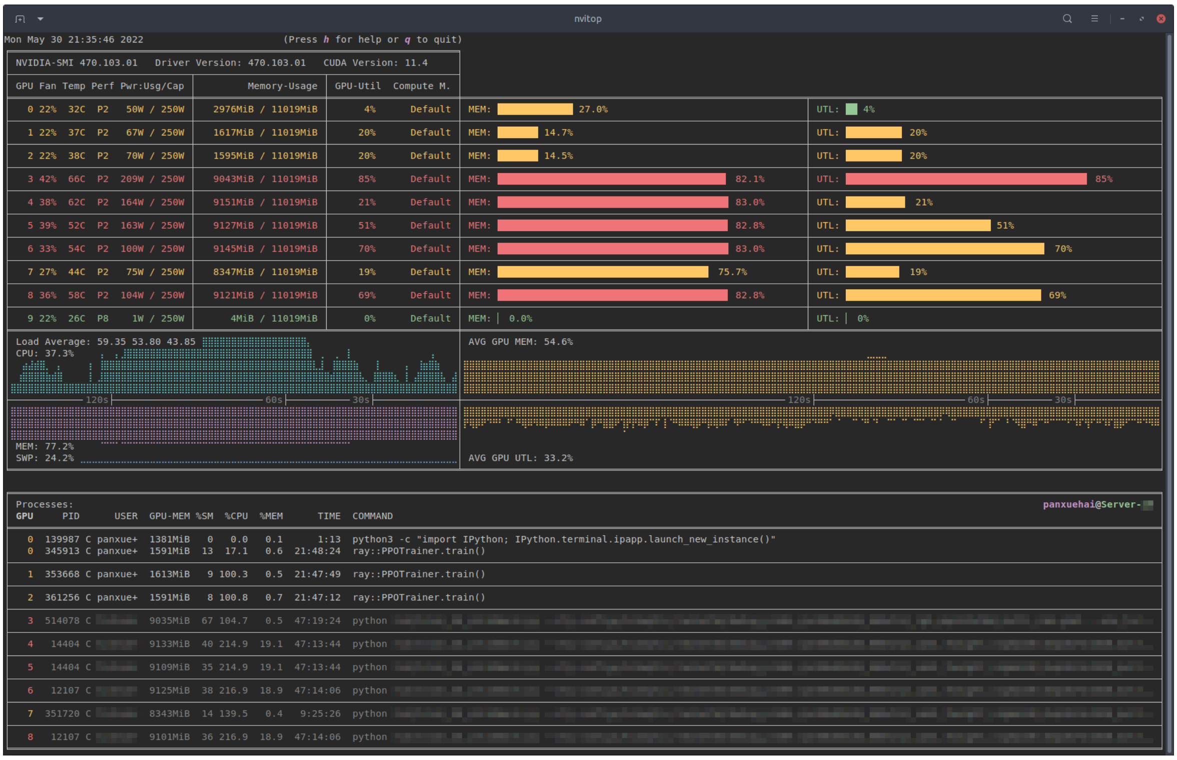
In [2]:

```python
# imports
import numpy as np
import torch
from torch.utils.data import DataLoader
import kornia  # `pip install kornia`
```

Monitoring Utilization

- How can we monitor the CPU and GPU utilization of our code?
- The simpleset way to monitor GPU utilization is to run `nvidia-smi` in the Terminal/CMD/PowerShell (can also use `!nvidia-smi` inside a Jupyter Notebook).
- For CPU monitoring running `htop` in Ubuntu's Terminal usually gets the job done.
- Recommedation: a great tool for detailed usage: NVITOP (`pip install nvitop`) and then run `nvitop -m` .

`nvitop -m`



```
Mon May 30 21:35:46 2022                         (Press h for help or q to quit)
 NVIDIA-SMI 470.103.01   Driver Version: 470.103.01   CUDA Version: 11.4
 GPU Fan Temp Perf Pwr:Usg/Cap        Memory-Usage   GPU-Util  Compute M.
   0 22%  32C  P2   50W / 250W   2976MiB / 11019MiB       4%     Default   MEM: 27.0%   UTL: 4%
   1 22%  37C  P2   67W / 250W   1617MiB / 11019MiB      20%     Default   MEM: 14.7%   UTL: 20%
   2 22%  38C  P2   70W / 250W   1595MiB / 11019MiB      20%     Default   MEM: 14.5%   UTL: 20%
   3 42%  66C  P2  209W / 250W   9043MiB / 11019MiB      85%     Default   MEM: 82.1%   UTL: 85%
   4 38%  62C  P2  164W / 250W   9151MiB / 11019MiB      21%     Default   MEM: 83.0%   UTL: 21%
   5 39%  52C  P2  163W / 250W   9127MiB / 11019MiB      51%     Default   MEM: 82.8%   UTL: 51%
   6 33%  54C  P2  100W / 250W   9145MiB / 11019MiB      70%     Default   MEM: 83.0%   UTL: 70%
   7 27%  44C  P2   75W / 250W   8347MiB / 11019MiB      19%     Default   MEM: 75.7%   UTL: 19%
   8 36%  58C  P2  104W / 250W   9121MiB / 11019MiB      69%     Default   MEM: 82.8%   UTL: 69%
   9 22%  26C  P8    1W / 250W      4MiB / 11019MiB       0%     Default   MEM: 0.0%    UTL: 0%
 Load Average: 59.35 53.80 43.85                                           AVG GPU MEM: 54.6%
 CPU: 37.3%
            120s            60s        30s                                 120s         60s      30s
 MEM: 77.2%
 SWP: 24.2%                                                                AVG GPU UTL: 33.2%

 Processes:                                                                      panxuehai@Server-[redacted]
 GPU     PID      USER  GPU-MEM %SM  %CPU  %MEM      TIME  COMMAND
   0  139987 C panxue+  1381MiB   0   0.0   0.1      1:13  python3 -c "import IPython; IPython.terminal.ipapp.launch_new_instance()"
   0  345913 C panxue+  1591MiB  13  17.1   0.6  21:48:24  ray::PPOTrainer.train()
   1  353668 C panxue+  1613MiB   9 100.3   0.5  21:47:49  ray::PPOTrainer.train()
   2  361256 C panxue+  1591MiB   8 100.8   0.7  21:47:12  ray::PPOTrainer.train()
   3  514078 C [redacted] 9035MiB  67 104.7   0.5  47:19:24  python [redacted]
   4   14404 C [redacted] 9133MiB  40 214.9  19.1  47:13:44  python [redacted]
   5   14404 C [redacted] 9109MiB  35 214.9  19.1  47:13:44  python [redacted]
   6   12107 C [redacted] 9125MiB  38 216.9  18.9  47:14:06  python [redacted]
   7  351720 C [redacted] 8343MiB  14 139.5   0.4   9:25:26  python [redacted]
   8   12107 C [redacted] 9101MiB  36 216.9  18.9  47:14:06  python [redacted]
```

In [10]:
```
!nvidia-smi
```

```
Tue Oct  4 10:28:20 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 517.48       Driver Version: 517.48       CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ... WDDM  | 00000000:3B:00.0 Off |                  N/A |
| N/A   42C    P8    N/A /  N/A |      0MiB /  4096MiB |      2%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     14804      C   ...nda\envs\torch\python.exe    N/A      |
+-----------------------------------------------------------------------------+
```

In [13]:
```
# check the current CUDA toolkit version installed in this `conda` environment
!nvcc -V
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Wed_Oct_23_19:32:27_Pacific_Daylight_Time_2019
Cuda compilation tools, release 10.2, V10.2.89
```

Understanding the Output of `nvidia-smi`

Adapted from: Explained Output of Nvidia-smi Utility - Shachi Kaul

- Top row:
    - **Driver Version**: the current version of the NVIDIA drivers installed (not the CUDA drivers, general drivers).
    - **CUDA Version**: the **maximal** CUDA driver version the current GPU and NVIDIA drivers (the ones from above) support. THIS IS NOT THE CURRENT CUDA DRIVERS VERSION INSTALLED. To check the current CUDA drivers installed, run `nvcc -V` (see above).
- First table:
    - **Temp**: Core GPU temperature is in degrees Celsius. Usual operation temperature should be around 80-85C, if it reaches 90C, there might be a problem with the cooling or some other hardware issue.
    - **Perf**: Denotes GPU's current performance state. It ranges from P0 to P12 referring to the maximum and minimum performance respectively.

    - **Persistence-M**: The value of Persistence Mode flag where “On” means that the NVIDIA driver will remain loaded (persist) even when no active client such as `nvidia-smi` is running. This reduces the driver load latency with dependent apps such as CUDA programs. Usually set to "Off".
    - **Pwr: Usage/Cap**: It refers to the GPU’s current power usage out of total power capacity. It samples in Watts.
    - **Disp.A**: Display Active is a flag that decides if you want to allocate memory on GPU device for display i.e. to initialize the display on GPU. “Off” indicates that there isn’t any display using a GPU device.
    - **Memory-Usage**: Denotes the memory allocation on GPU out of total memory. This should help you balance model size and batch size and other hyper-parameters that greatly affect the memory required from the GPU.
    - **Volatile Uncorr. ECC**: ECC stands for Error Correction Code which verifies data transmission by locating and correcting transmission errors. NVIDIA GPUs provide an error count of ECC errors. Here, Volatile error counter detects error count since the last driver loaded.
    - **GPU-Util**: Indicates the percent of GPU utilization i.e. percent of the time when kernels were using GPU over the sample period. Here, the period could be between 1 to 1/6th second (depending on the product being queried). In the case of low percentage, the GPU is under-utilized (e.g., code spends time on reading data from disk).
    - **Compute M.**: Compute Mode of specific GPU refers to the shared access mode where compute mode sets to default after each reboot. “Default” value allows multiple clients to access the CPU at the same time.
- Second table:
    - **GPU**: Indicates the GPU index, beneficial for multi-GPU setup. This determines which process is utilizing which GPU. This index represents the NVML Index of the device.
    - **PID**: Refers to the process by its ID using GPU. If you need to kill a ghost process that uses the GPU (usually happnes in distributed training / multiprocessing), use that PID for the `kill` command.
    - **Type:**: Refers to the type of processes such as “C” (Compute), “G” (Graphics), and “C+G” (Compute and Graphics context).
    - **GPU Memory Usage**: Memory of specific GPU utilized by each process.

For more options, run `nvidia-smi --help`.

- **Advanced**: manually control the cooling of the GPU (by setting temperature ranges for fan speed): `coolgpus`, GitHub link.

It also possible to access the GPU utilization in code as follows (PyTorch 1.12):

`torch.cuda.utilization(device=None)`: Returns the percent of time over the past sample period during which one or more kernels was executing on the GPU as given by `nvidia-smi`.

`device`: Selected device. Returns statistic for the current device, given by `current_device()`, if device is `None` (default).

Cleaning CUDA Cache

If you want to manually free up memory in the GPU, it sometimes helps to clean the cache stored in the GPU by calling `torch.cuda.empty_cache()`. This is useful especially if you are working in a Jupyter Notebook, but can also be useful after an epoch has ended (use carefully).

In [15]:

```
torch.cuda.empty_cache()
```

General Tips

More tricks and hacks can be found HERE.

In [3]:

```
# reminder - define device at the top of your code, and send models and tensors to it
# check if there is a CUDA device available
print(f'torch.cuda.is_available(): {torch.cuda.is_available()}')
# define device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'device: {device}')
```

```
torch.cuda.is_available(): True
device: cuda:0
```

In [9]:

```
# a simple neural network
class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.nn = torch.nn.Sequential(torch.nn.Linear(10, 128),
                            torch.nn.ReLU(inplace=True),
                            torch.nn.Linear(128, 256),
                            torch.nn.ReLU(inplace=True),
                            torch.nn.Linear(256, 10))
    def forward(self, x):
        return self.nn(x)

# `inplace=True`: performs the operation in-place -- not creating a copy of the tensor, can save memory.
model = Model()
# send model to device
model = model.to(device)
print(model)
print(f'model device: {next(model.parameters()).device}')
```

```
Model(
  (nn): Sequential(
    (0): Linear(in_features=10, out_features=128, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=256, out_features=10, bias=True)
  )
)
model device: cuda:0
```

Avoid unnecessary CPU-GPU synchronization

Avoid unnecessary synchronizations, to let the CPU run ahead of the accelerator as much as possible to make sure that the accelerator work queue contains many operations.

When possible, avoid operations which require synchronizations, for example:

- `print(cuda_tensor)`
- `cuda_tensor.item()`
- Memory copies: `tensor.cuda(), cuda_tensor.cpu()` and equivalent `tensor.to(device)` calls.
- `cuda_tensor.nonzero()`
- `cuda_tensor.data.cpu().numpy()`
- Python control flow which depends on results of operations performed on cuda tensors e.g. `if (cuda_tensor != 0).all()`.

Create tensors directly on the target device

Instead of calling `torch.rand(size).cuda()` to generate a random tensor, produce the output directly on the target device: `torch.rand(size, device=torch.device('cuda'))`.

This is applicable to all functions which create new tensors and accept device argument: `torch.rand(), torch.zeros(), torch.ones(), torch.full()` and similar.

In [ ]:

```
# example
a = torch.randn(32, 10).to(device)  # BAD
b = torch.randn(32, 10, device=device)  # GOOD
```

cuDNN auto-tuner

NVIDIA cuDNN supports many algorithms to compute a convolution. Autotuner runs a short benchmark and selects the kernel with the best performance on a given hardware for a given input size.

For convolutional networks (other types currently not supported), enable cuDNN autotuner before launching the training loop by setting:

`torch.backends.cudnn.benchmark = True`

Notes:

- The auto-tuner decisions may be **non-deterministic**; different algorithm may be selected for different runs.

- In some rare cases, such as with highly variable input sizes, it's better to run convolutional networks with autotuner *disabled* to avoid the overhead associated with algorithm selection for each input size.

- If you care about **reproducibility**, it is better to use:

`torch.backends.cudnn.benchmark = False`

`torch.backends.cudnn.deterministic = True`

In [ ]:

```python
# example
# imports
import torch
# put the line here
torch.backends.cudnn.benchmark = True
# rest of the training function goes below
```

Disable gradient calculation for validation or inference

- PyTorch saves intermediate buffers from all operations which involve tensors that require gradients.
- Typically gradients aren't needed for validation or inference.
- `torch.no_grad()` context manager can be applied to disable gradient calculation within a specified block of code, this accelerates execution and reduces the amount of required memory. `torch.no_grad()` can also be used as a function decorator.

In [ ]:

```python
# validation loop
valid_dataloader = DataLoader()
model.eval()
with torch.no_grad():
    for batch in valid_dataloader:
        x = batch[0]
        output = model(x)  # no gradient cache
        # metrics caclulation on output
# make sure to put the model back in training mode after validation ends
model.train()
```

Disable bias for convolutions directly followed by a Batch Normalization

- `torch.nn.Conv2d()` has a bias parameter which defaults to `True` (the same is true for `Conv1d` and `Conv3d`).

- If a `nn.Conv2d` layer is directly followed by a `nn.BatchNorm2d` layer, then the bias in the convolution is not needed, instead use `nn.Conv2d(..., bias=False, ....)` . Bias is not needed because in the first step BatchNorm subtracts the mean, which effectively cancels out the effect of bias.

- This is also applicable to 1D and 3D convolutions as long as BatchNorm (or other normalization layer) normalizes on the same dimension as convolution's bias.

In [ ]:

```python
conv_layer = torch.nn.Sequential(torch.nn.Conv2d(3, 64, stride=1, kernel_size=3, padding=1, bias=False),
                                 torch.nn.BatchNorm2d(64),
                                 torch.nn.ReLU(inplace=True))
```

Use `parameter.grad = None` instead of `optimizer.zero_grad()`

- Note: this is a minor optimization, do not expect large gains from it.

Instead of calling: `optimizer.zero_grad()` to zero out gradients, use the following method instead:

`for param in model.parameters():`
`param.grad = None`

From PyTorch 1.7, can also use: `optimizer.zero_grad(set_to_none=True)`

The second code snippet does not zero the memory of each individual parameter, also the subsequent backward pass uses assignment instead of addition to store gradients, this reduces the number of memory operations.

Setting gradient to `None` has a slightly **different numerical behavior** than setting it to zero, so be careful when using it.

In [ ]:

```python
# code doesn't change much
output = model(x)
```

```
loss = loss_fn(x, output)
optimizer.zero_grad(set_to_none=True)
loss.backward()
optimizer.step()
```

`set_to_none (bool)` – instead of setting to zero, set the grads to `None`. This will in general have lower memory footprint, and can modestly improve performance.

However, it changes certain behaviors.

For example:

1. When the user tries to access a gradient and perform manual ops on it, a `None` attribute or a `Tensor` full of 0s will behave differently.
2. If the user requests `zero_grad(set_to_none=True)` followed by a backward pass, `.grad`s are guaranteed to be `None` for params that did not receive a gradient.
3. `torch.optim optimizers` have a different behavior if the gradient is 0 or `None` (in one case it does the step with a gradient of 0 and in the other it skips the step altogether).

Use `model.requires_grad_(False)` when using pre-trained models

- When using pre-trained models in another model, we want to preserve the gradients of the input of the pre-trained network, but we don't want to calculate the gradients of the pre-trained model.
- Examples:
  - Perceptual loss: use the features of a pre-trained VGG network to calculate reconstruction loss.
  - Use CLIP score as guidance for a generative network.
- Why is `torch.no_grad()` not good enough? Because wrapping the forward pass of the pre-trained model with `torch.no_grad()` will not calculate gradients for the input, which is the **output** of a neural network that is being trained.
- Solution: after loading the pre-trained model, call `model.requires_grad_(False)`.

In [ ]:

```
# example: vgg for perceptual loss
from torchvision import models

vggnet = models.vgg16(weigths='DEFAULT')
vggnet.eval()  # for dropout and batch-norm
vggnet.requires_grad_(False)
```

Avoid `for` loops (duh...)

- It goes without saying that avoiding loops in the code is crucial for fast training.
- Always try to batch operations, though you might eventually find out there is no avoiding a loop, it is worth looking for ways to batch operations.
- Trivial cases are when we need to work with tensors of shape `[batch_size, dim_a, dim_b]` or `[batch_size, dim_a, dim_b, dim_c]`, it might be tempting to loop over `dim_a`, but if the GPU memory allows, we can just stack everything on the batch dimension.
  - Another popular case is applying convolutional layers to patches: consider an image tensor of size `[batch_size, ch, h, w]` which is patchified to a tensor of size `[batch_size, num_patches, ch, h_p, w_p]`, then we don't need to loop over `num_patches`, but just stack the `num_patches` dimension onto the the `batch_size` dimension.

In [17]:

```
batch_size = 32
dim_a = 10
dim_b = 16

func = torch.nn.Linear(16, 32)

a = torch.rand(batch_size, dim_a, dim_b)
print(f'a: {a.shape}')  # [batch_size, dim_a, dim_b]
# want to apply some function on dim_b -> batch dim_a in the batch dimension
a = a.view(-1, a.shape[-1])  # [batch_size * dim_a, dim_b]
print(f'a: {a.shape}')
# apply the function and then reshape to the original dimension
a_f = func(a)  # [batch_size * dim_a, 32]
print(f'a_f: {a_f.shape}')
a_f = a_f.view(batch_size, dim_a, a_f.shape[-1])
print(f'a_f: {a_f.shape}')
```

```
# note: torch.nn.Linear actually does this automatically
a = torch.rand(batch_size, dim_a, dim_b)  # [batch_size, dim_a, dim_b]
a_f_1 = func(a)  # [batch_size, dim_a, 32]
print(f'a_f_1: {a_f_1.shape}')
```

```
a: torch.Size([32, 10, 16])
a: torch.Size([320, 16])
a_f: torch.Size([320, 32])
a_f: torch.Size([32, 10, 32])
a_f_1: torch.Size([32, 10, 32])
```

Model Compiling: `torch.compile`

- From PyTorch 2.0 it is possible to compile models for speed-up gains on certain GPU architectures (e.g., H100, A100, or V100).
- `torch.compile` makes PyTorch code run faster by **Just-in-Time**(JIT)-compiling PyTorch code into optimized kernels.
- Speedup mainly comes from reducing Python overhead and GPU read/writes, and so the observed speedup may vary on factors such as model architecture and batch size.
  - For example, if a model's architecture is simple and the amount of data is large, then the bottleneck would be GPU compute and the observed speedup may be less significant.
- The `mode` parameter specifies what the compiler should be optimizing while compiling.
  - The `default` mode is a preset that tries to compile efficiently without taking too long to compile or using extra memory.
  - Other modes such as `reduce-ovehead` reduce the framework overhead by a lot more, but cost a small amount of extra memory.
  - `max-autotune` mode compiles for a long time, trying to give you the fastest code it can generate.
  - More information on modes.
- Currently, `torch.compile` only works on **Linux** machines (no support for Windows yet, an error will be thrown if using `torch.compile`).
- More details on `torch.compile` can be found on the official PyTorch website.

In [ ]:
```
# example
import torch
class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = torch.nn.Linear(100, 10)

    def forward(self, x):
        return torch.nn.functional.relu(self.lin(x))

mod = MyModule()
opt_mod = torch.compile(mod, mode='default')
print(opt_mod(torch.randn(10, 100)))
```

In [ ]:
```
# another example
import torch
import torchvision.models as models

device = torch.device('cuda:0')  # note: must be GPU
model = models.resnet18().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
compiled_model = torch.compile(model)

x = torch.randn(16, 3, 224, 224).cuda()
optimizer.zero_grad()
out = compiled_model(x)
out.sum().backward()
optimizer.step()
```

Advanced: Fuse pointwise operations with Just-in-Time (JIT)

- Pointwise operations (elementwise addition, multiplication, math functions - `sin()`, `cos()`, `sigmoid()` etc.) can be fused into a single kernel to amortize memory access time and kernel launch time.

- PyTorch JIT can fuse kernels automatically, although there could be additional fusion opportunities not yet implemented in the compiler, and not all device types are supported equally.

- Pointwise operations are **memory-bound**, for each operation PyTorch launches a separate kernel. Each kernel loads data from the memory, performs computation (this step is usually inexpensive) and stores results back into the memory.

- Fused operator launches only one kernel for multiple fused pointwise ops and loads/stores data **only once** to the memory. This makes JIT very useful for **activation functions, optimizers, custom RNN cells** etc.

- In the simplest case fusion can be enabled by applying `torch.jit.script` decorator to the function definition.

In [ ]:

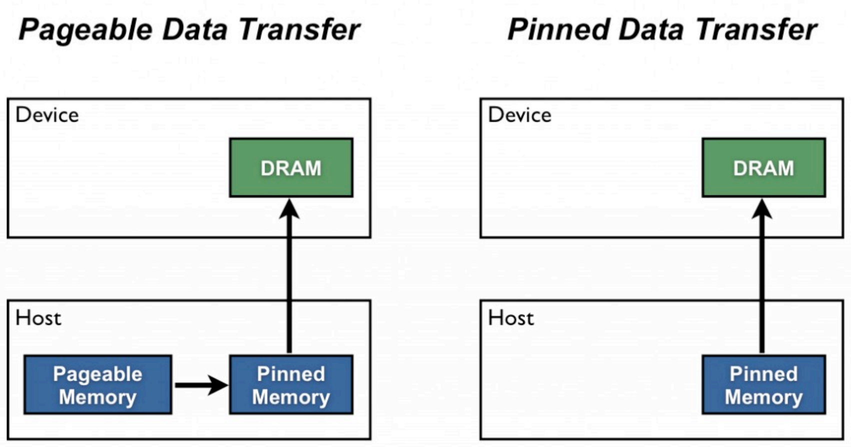
```
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

Data Loading Tips to Maximize GPU Utility

- One of the greatest bottlenecks in training NNs is the data loading into memory and transferring it to the GPU for computation.
- If the data loading process is lengthy (e.g., the data itself is complex or a lot of pre-processing is required), this can result in a lot of CPU time while the GPU sits idle (i.e., low GPU utilization in `nvidia-smi`).
- We'd like to fetch the data as fast as possible and quickly transfer it to the GPU. We'll review a couple of approaches to accelerate this process.

Use pinned memory data loading

- Host to GPU copies are much faster when they originate from pinned (page-locked) memory.
- In CUDA, non-paged CPU (RAM) memory is referred to as pinned memory. Pinning a block of memory can be done via a CUDA API call, which issues an OS call that reserves the memory block and sets the constraint that it cannot be spilled to disk.
- Pinned memory is used to speed up a CPU to GPU memory copy operation (as executed by e.g. `tensor.cuda()` in PyTorch) by ensuring that none of the memory that is to be copied is on disk.
- Memory cached to disk has to be read into RAM before it can be transferred to the GPU—e.g. *it has to be copied twice*. You can naively expect this to be twice as slow (the true slowness depends on the size and business of the relevant memory buses).
- The `pin_memory` field (`pin_memory=True`) on `DataLoader` invokes this memory management model.
  - Note: this technique requires that the OS is willing to give the PyTorch process as much main memory as it needs to complete its load and transform operations—e.g. the batch must fit into RAM in its entirety.



In [ ]:

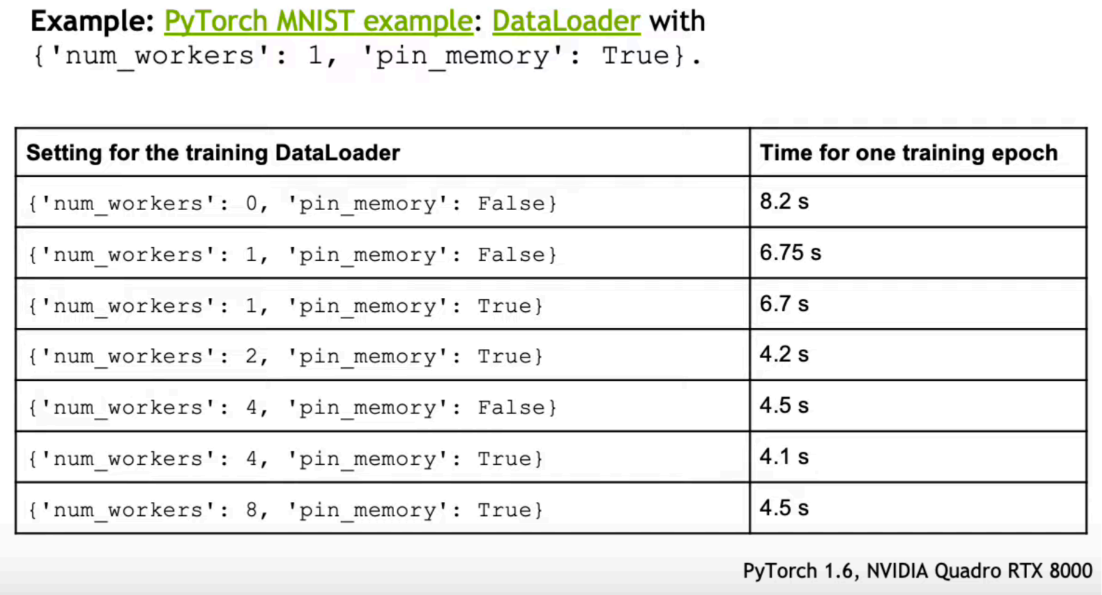
```
# example
train_dataset = Dataset()
train_dataloader = DataLoader(dataset, batch_size=128, shuffle=True, pin_memory=True)
```

Use multiprocessing for data loading

- Applying the following tip should speed-up your training code significantly.
- PyTorch allows loading data on multiple processes simultaneously.
- To set the number of workers: `DataLoader(num_workers=4)` .

- How to set the number of workers? A general rule is `num_workers = 4 * num_gpus` ;however, this is highly machine-depndent, and should be treated as a hyper-parameter.
    - Usually 4 or 8 is a good number to start with.
    - A nice discussion on PyTorch forums: Guidelines for assigning num_workers to DataLoader
    - A script to search for the optimal `num_workers` : Num-Workers-Search.
- Important notes:
    - Having more workers will **increase the memory usage**.
    - `num_workers=0` means that it’s the main process that will do the data loading when needed
    - `num_workers=1` is the same as any n, but you’ll only have a single worker, which is probably slower.

In [ ]:
```
train_dataloader = DataLoader(dataset, batch_size=128, shuffle=True, pin_memory=True, num_workers=4)
```



Example: PyTorch MNIST example: DataLoader with `{'num_workers': 1, 'pin_memory': True}.`

Setting for the training DataLoader	Time for one training epoch
`{'num_workers': 0, 'pin_memory': False}`	8.2 s
`{'num_workers': 1, 'pin_memory': False}`	6.75 s
`{'num_workers': 1, 'pin_memory': True}`	6.7 s
`{'num_workers': 2, 'pin_memory': True}`	4.2 s
`{'num_workers': 4, 'pin_memory': False}`	4.5 s
`{'num_workers': 4, 'pin_memory': True}`	4.1 s
`{'num_workers': 8, 'pin_memory': True}`	4.5 s

PyTorch 1.6, NVIDIA Quadro RTX 8000

Perform image augmentations directly on the GPU with Kornia

- Instead of performing augmentations with `torchvision` , we can perform the augmentations on the GPU, which can speed-up the data loading process.
- Kornia is a differentiable library that allows classical computer vision to be integrated into deep learning models.
    - That menas augementations and filters can utilize the GPU but also be differentiable for the backpropagation process!
    - `pip install kornia`

In [ ]:
```
from kornia import augmentation as K
from kornia.augmentation import AugmentationSequential

aug_list = AugmentationSequential(
    K.ColorJitter(0.1, 0.1, 0.1, 0.1, p=1.0),
    K.RandomAffine(360, [0.1, 0.1], [0.7, 1.2], [30., 50.], p=1.0),
    K.RandomPerspective(0.5, p=1.0),
    return_transform=False,
    same_on_batch=False,
)

img_aug = aug_list(img_tensor)  # [batch_size, num_ch, h, w]
```

Here is a benchmark performed on Google Colab K80 GPU with different libraries and batch sizes. This benchmark shows strong GPU augmentation speed acceleration brought by Kornia data augmentations. The image size is fixed to 224x224 and the unit is milliseconds (ms).

Libraries	**TorchVision**	**Albumentations**	**Kornia (GPU)**		
Batch Size	**1**	**1**	**1**	**32**	**128**
RandomPerspective	4.88±1.82	4.68±3.60	4.74±2.84	0.37±2.67	0.20±27.00
ColorJiggle	4.40±2.88	3.58±3.66	4.14±3.85	0.90±24.68	0.83±12.96

Libraries	TorchVision	Albumentations	Kornia (GPU)		
Batch Size	1	1	1	32	128
RandomAffine	3.12±5.80	2.43±7.11	3.01±7.80	0.30±4.39	0.18±6.30
RandomVerticalFlip	0.32±0.08	0.34±0.16	0.35±0.82	0.02±0.13	0.01±0.35
RandomHorizontalFlip	0.32±0.08	0.34±0.18	0.31±0.59	0.01±0.26	0.01±0.37
RandomRotate	1.82±4.70	1.59±4.33	1.58±4.44	0.25±2.09	0.17±5.69
RandomCrop	4.09±3.41	4.03±4.94	3.84±3.07	0.16±1.17	0.08±9.42
RandomErasing	2.31±1.47	1.89±1.08	2.32±3.31	0.44±2.82	0.57±9.74
RandomGrayscale	0.41±0.18	0.43±0.60	0.45±1.20	0.03±0.11	0.03±7.10
RandomResizedCrop	4.23±2.86	3.80±3.61	4.07±2.67	0.23±5.27	0.13±8.04
RandomCenterCrop	2.93±1.29	2.81±1.38	2.88±2.34	0.13±2.20	0.07±9.41

Training Tips

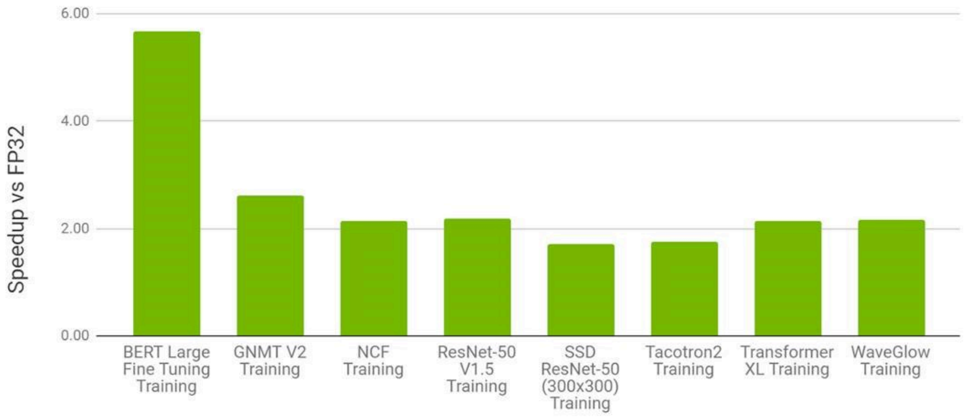
There are 2 main approaches (that can be combined) to achieve a speed-up during train time:

1. Automatic Mixed Precision (AMP) - instead of working with tensors at full-precision (`float32`), we can (sometimes) work at half-precision (`float16`). Current advances in this field also experiment with even lower precision.
2. Distributed training - utilizing multiple GPUs to train a single model.

Use mixed precision and AMP

- Mixed precision leverages Tensor Cores and offers up to 3x overall speedup on Volta and newer GPU architectures.
- To use Tensor Cores AMP should be enabled and matrix/tensor dimensions should satisfy requirements for calling kernels that use Tensor Cores.
- Deep Neural Network training has traditionally relied on FP32 (32-bit Floating Point, IEEE single-precision format).
- The (automatic) mixed precision technique - training with FP16 (16-bit Floating Point, half-precision) while maintaining the network accuracy achieved with FP32.
- Enabling mixed precision involves two steps:
  - Porting the model to use the half-precision data type **where appropriate**.
  - Using loss scaling to preserve small gradient values.

Performance of mixed precision training on NVIDIA 8xV100 vs. FP32 training on 8xV100 GPU



- Bars represent the speedup factor of V100 AMP over V100 FP32. The higher the better.

Image Source

- AMP Recipe from PyTorch
- ECE 046211 AMP Tutorial

In [ ]:

```
# general recipe

use_amp = True

net = make_model(in_size, out_size, num_layers)
opt = torch.optim.SGD(net.parameters(), lr=0.001)
scaler = torch.cuda.amp.GradScaler(enabled=use_amp) # notice the `enabled` parameter
# Gradient scaling helps prevent gradients with small magnitudes from flushing
# to zero (“underflowing”) when training with mixed precision

start_timer()
for epoch in range(epochs):
    for inputs, target in zip(data, targets):
        # notice the `enabled` parameter
        with torch.cuda.amp.autocast(enabled=use_amp):
            output = net(inputs)
            loss = loss_fn(output, target)

        # set_to_none=True here can modestly improve performance, replace 0 (float) with None (save mem)
        opt.zero_grad(set_to_none=True)
        scaler.scale(loss).backward()
        scaler.step(opt)
        scaler.update()
```

Distributed multi-GPU training with HuggingFace's Accelerate

- In the past, a lot of boilerplate code was needed to make our PyTorch code run on multiple GPUs with `torch.distributed`.
- Today, things are a lot more easier with wrappers like HuggingFace Accelerate.
    - `pip instal accelerate`
- Accelerate is a library that enables the same PyTorch code to be run across any distributed configuration by adding just four lines of code.
- Accelerate supports the following configurations: CPU only, multi-CPU on one node (machine), multi-CPU on several nodes (machines), single GPU, multi-GPU on one node (machine), multi-GPU on several nodes (machines), TPU, FP16 with native AMP (apex on the roadmap), DeepSpeed support (Experimental), PyTorch Fully Sharded Data Parallel (FSDP) support (Experimental).
- Examples can be found here

In [ ]:

```
import torch
import torch.nn.functional as F
from datasets import load_dataset
from accelerate import Accelerator

accelerator = Accelerator()  # NEW

model = torch.nn.Transformer()
optimizer = torch.optim.Adam(model.parameters())

dataset = load_dataset('my_dataset')
data = torch.utils.data.DataLoader(dataset, shuffle=True)

model, optimizer, data = accelerator.prepare(model, optimizer, data)  # NEW

model.train()
for epoch in range(10):
    for source, targets in data:
        output = model(source)
        loss = F.cross_entropy(output, targets)
        optimizer.zero_grad()
        accelerator.backward(loss)  # NEW
        optimizer.step()
```

- To launch the new distributed code we create a config file with: `accelerate config` which will ask you a few questions (e.g., how many GPUs to use).
- Finally, run `accelerate launch my_script.py --args_to_my_script`.

- If you work with PyTorch Lightning, then it already installs Accelerate and wraps it in its API.

PyTorch Profiler

- PyTorch Profiler is useful to determine the most expensive operators in the model and analyze execution times.
- PyTorch Profiler tutorial

In [1]:

```
# usage example
import torch
import torchvision.models as models
from torch.profiler import profile, record_function, ProfilerActivity

model = models.resnet18()
inputs = torch.randn(5, 3, 224, 224)
```

In [2]:

```
# analyze execution time
with profile(activities=[ProfilerActivity.CPU], profile_memory=True, record_shapes=True) as prof:
    with record_function("model_inference"):
        model(inputs)
```

In [3]:

```
# stats for the execution
print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

```
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------
                             Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg       CPU
Mem  Self CPU Mem    # of Calls
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------
                  model_inference         4.44%      10.480ms       100.00%     236.255ms     236.255ms
0 b    -106.30 Mb             1
                     aten::conv2d         2.04%       4.815ms        61.43%     145.141ms       7.257ms      47.3
7 Mb           0 b            20
                aten::convolution         1.75%       4.137ms        59.40%     140.326ms       7.016ms      47.3
7 Mb           0 b            20
               aten::_convolution         0.63%       1.482ms        57.64%     136.189ms       6.809ms      47.3
7 Mb           0 b            20
         aten::mkldnn_convolution        56.15%     132.665ms        57.02%     134.707ms       6.735ms      47.3
7 Mb           0 b            20
                 aten::max_pool2d         0.30%     704.000us        14.15%      33.425ms      33.425ms      11.4
8 Mb           0 b             1
    aten::max_pool2d_with_indices        13.85%      32.721ms        13.85%      32.721ms      32.721ms      11.4
8 Mb      11.48 Mb             1
                 aten::batch_norm         0.30%     717.000us        10.10%      23.860ms       1.193ms      47.4
1 Mb           0 b            20
     aten::_batch_norm_impl_index         0.45%       1.066ms         9.80%      23.143ms       1.157ms      47.4
1 Mb           0 b            20
          aten::native_batch_norm         9.09%      21.469ms         9.33%      22.044ms       1.102ms      47.4
1 Mb     -64.00 Kb            20
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------
Self CPU time total: 236.255ms
```

In [4]:

```
# include operator input shapes and sort by the self cpu time
print(prof.key_averages(group_by_input_shape=True).table(sort_by="cpu_time_total", row_limit=10))
```

```
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------  -------------------------------------------------------------------------------
-
                             Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg       CPU
Mem  Self CPU Mem    # of Calls                                                                      Input Shapes
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------  -------------------------------------------------------------------------------
-
                  model_inference         4.44%      10.480ms       100.00%     236.255ms     236.255ms
0 b    -106.30 Mb             1                                                                                []
                     aten::conv2d         1.99%       4.701ms        16.21%      38.296ms      38.296ms      15.3
1 Mb           0 b             1                             [[5, 3, 224, 224], [64, 3, 7, 7], [], [], [], [],
[]]
                aten::convolution         1.57%       3.721ms        14.22%      33.595ms      33.595ms      15.3
1 Mb           0 b             1                     [[5, 3, 224, 224], [64, 3, 7, 7], [], [], [], [], [], [],
[]]
                 aten::max_pool2d         0.30%     704.000us        14.15%      33.425ms      33.425ms      11.4
8 Mb           0 b             1                                           [[5, 64, 112, 112], [], [], [], [],
[]]
    aten::max_pool2d_with_indices        13.85%      32.721ms        13.85%      32.721ms      32.721ms      11.4
8 Mb      11.48 Mb             1                                           [[5, 64, 112, 112], [], [], [], [],
[]]
               aten::_convolution         0.52%       1.239ms        12.64%      29.874ms      29.874ms      15.3
1 Mb           0 b             1     [[5, 3, 224, 224], [64, 3, 7, 7], [], [], [], [], [], [], [], [], [], [],
[]]
         aten::mkldnn_convolution        11.39%      26.910ms        12.12%      28.635ms      28.635ms      15.3
1 Mb           0 b             1                             [[5, 3, 224, 224], [64, 3, 7, 7], [], [], [], [],
[]]
                     aten::conv2d         0.01%      23.000us        11.55%      27.276ms       6.819ms      15.3
1 Mb           0 b             4                             [[5, 64, 56, 56], [64, 64, 3, 3], [], [], [], [],
[]]
                aten::convolution         0.03%      78.000us        11.54%      27.253ms       6.813ms      15.3
1 Mb           0 b             4                     [[5, 64, 56, 56], [64, 64, 3, 3], [], [], [], [], [], [],
[]]
               aten::_convolution         0.02%      49.000us        11.50%      27.175ms       6.794ms      15.3
1 Mb           0 b             4     [[5, 64, 56, 56], [64, 64, 3, 3], [], [], [], [], [], [], [], [], [], [],
[]]
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------  -------------------------------------------------------------------------------
-
Self CPU time total: 236.255ms
```

In [5]:

```
# sort by memory usage
print(prof.key_averages().table(sort_by="cpu_memory_usage", row_limit=10))
```

```
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------
                             Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg       CPU
Mem  Self CPU Mem    # of Calls
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------
                      aten::empty         0.25%     588.000us         0.25%     588.000us       2.940us      94.8
5 Mb      94.85 Mb           200
                 aten::batch_norm         0.30%     717.000us        10.10%      23.860ms       1.193ms      47.4
1 Mb           0 b            20
     aten::_batch_norm_impl_index         0.45%       1.066ms         9.80%      23.143ms       1.157ms      47.4
1 Mb           0 b            20
          aten::native_batch_norm         9.09%      21.469ms         9.33%      22.044ms       1.102ms      47.4
1 Mb     -64.00 Kb            20
                     aten::conv2d         2.04%       4.815ms        61.43%     145.141ms       7.257ms      47.3
7 Mb           0 b            20
                aten::convolution         1.75%       4.137ms        59.40%     140.326ms       7.016ms      47.3
7 Mb           0 b            20
               aten::_convolution         0.63%       1.482ms        57.64%     136.189ms       6.809ms      47.3
7 Mb           0 b            20
         aten::mkldnn_convolution        56.15%     132.665ms        57.02%     134.707ms       6.735ms      47.3
7 Mb           0 b            20
                 aten::empty_like         0.12%     277.000us         0.15%     343.000us      17.150us      47.3
7 Mb           0 b            20
                 aten::max_pool2d         0.30%     704.000us        14.15%      33.425ms      33.425ms      11.4
8 Mb           0 b             1
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------
Self CPU time total: 236.255ms
```

In [7]:

```
# analyze performance of models executed on GPUs
device = torch.device("cuda:0")
model = models.resnet18().to(device)
inputs = torch.randn(5, 3, 224, 224).to(device)
```

```
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA], record_shapes=True) as prof:
    with record_function("model_inference"):
        model(inputs)

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

```
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------  ------------  ------------
                             Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg     Self 
CUDA   Self CUDA %    CUDA total  CUDA time avg    # of Calls
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------  ------------  ------------
                  model_inference         0.38%       3.567ms       100.00%     947.626ms     947.626ms     195.0
00us         0.03%     679.493ms     679.493ms             1
                     aten::conv2d         0.02%     154.000us        81.83%     775.462ms      38.773ms      74.0
00us         0.01%     515.147ms      25.757ms            20
                aten::convolution         0.04%     338.000us        81.82%     775.308ms      38.765ms      68.0
00us         0.01%     515.073ms      25.754ms            20
               aten::_convolution         0.04%     340.000us        81.78%     774.970ms      38.748ms      84.0
00us         0.01%     515.005ms      25.750ms            20
          aten::cudnn_convolution        81.74%     774.630ms        81.74%     774.630ms      38.731ms     514.9
21ms        75.78%     514.921ms      25.746ms            20
                       aten::add_         3.32%      31.494ms         3.32%      31.494ms       1.125ms     100.8
32ms        14.84%     100.832ms       3.601ms            28
                 aten::batch_norm         0.01%     109.000us         5.53%      52.364ms       2.618ms      75.0
00us         0.01%      38.227ms       1.911ms            20
     aten::_batch_norm_impl_index         0.11%       1.011ms         5.51%      52.255ms       2.613ms      70.0
00us         0.01%      38.152ms       1.908ms            20
           aten::cudnn_batch_norm         5.17%      49.028ms         5.41%      51.244ms       2.562ms      37.7
90ms         5.56%      38.082ms       1.904ms            20
        aten::adaptive_avg_pool2d         0.00%      17.000us         3.35%      31.774ms      31.774ms       3.0
00us         0.00%      10.017ms      10.017ms             1
---------------------------------  ------------  ------------  ------------  ------------  ------------  --------
----  ------------  ------------  ------------  ------------
Self CPU time total: 947.626ms
Self CUDA time total: 679.493ms
```

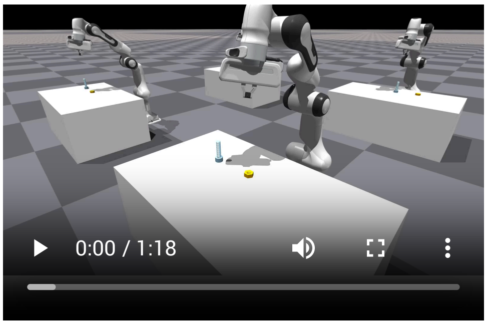
Reinforcement Learning

- In (online) RL, the main bottleneck is the interaction with the environment in order to collect data, which usually occurs in a simulated environment that runs on the CPU.
- This setting results in "spikes" in the GPU utility, where the GPU is idle during the interaction in the environment.
- There are several approaches to speed-up the RL process. For example, one can utilize simulators that natively run on GPUs where inputs and ouputs are already tensors, reducing the CPU-GPU latency and utilizes the tensor cores to speed-up the simulation itself (faster data collection). Another persepctive is CPU multiprocessing -- utilizing several environments in parallel, which will hopefully reduce CPU time.

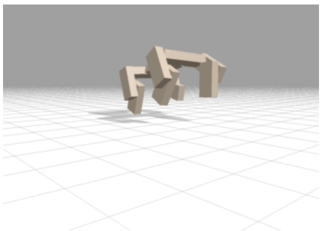
GPU-based Simulators

- NVIDIA's Isaacs Gym - NVIDIA’s physics simulation environment for reinforcement learning research.
  - GPU accelerated tensor API for evaluating environment state and applying actions.
  - Physics simulation in Isaac Gym runs on the GPU, storing results in PyTorch GPU tensors.
  - Using the Isaac Gym tensor-based APIs, observations and rewards can be calculated on the GPU in PyTorch, enabling thousands of environments to run in parallel on a single workstation.
  - Isaac Gym Video Tutorial
  - Code examples: Isaacs Gym Environments, ElegantRL, Collection of Isaacs Resources, PPO Example, Adversarial Skill Embedding

In [2]:
```
%%HTML
<center>
<video width="320" height="240" controls>
  <source src="https://developer.download.nvidia.com/video/Nut_Bolt_Screw_IK_OSC.mp4" type="video/mp4">
</video>
</center>
```

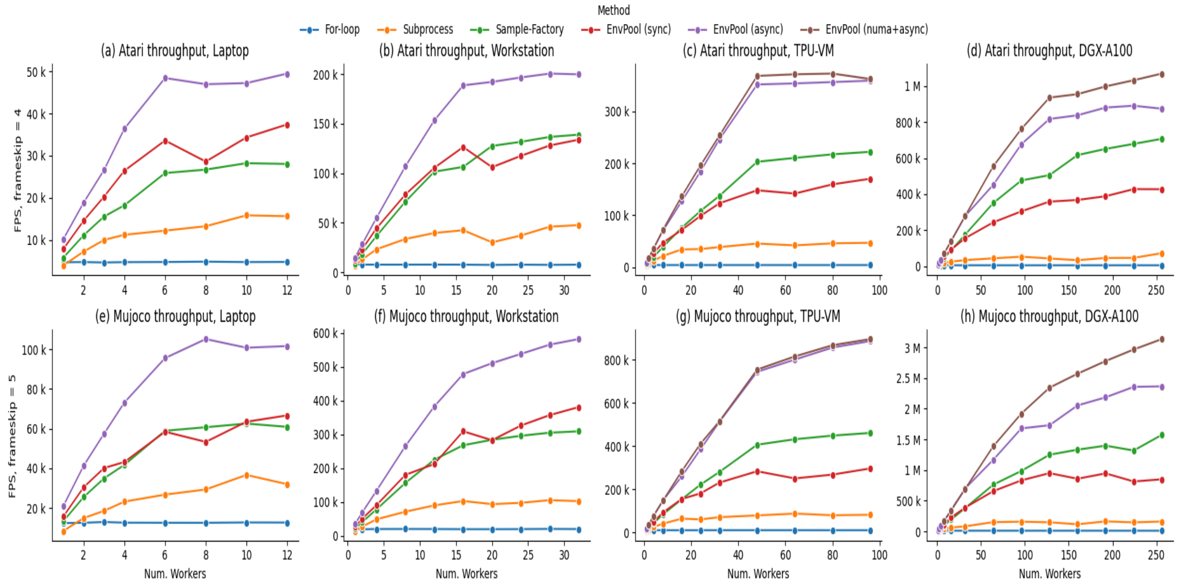


- Google's BRAX - a differentiable physics engine that simulates environments made up of rigid bodies, joints, and actuators.
    - Brax is written in JAX and is designed for use on acceleration hardware.
    - It is both efficient for single-device simulation, and scalable to massively parallel simulation on multiple devices, without the need for pesky datacenters.
    - Brax also includes a suite of learning algorithms that train agents in seconds to minutes.
    - Code examples: Training in Brax with PyTorch on GPUs, PPO Example



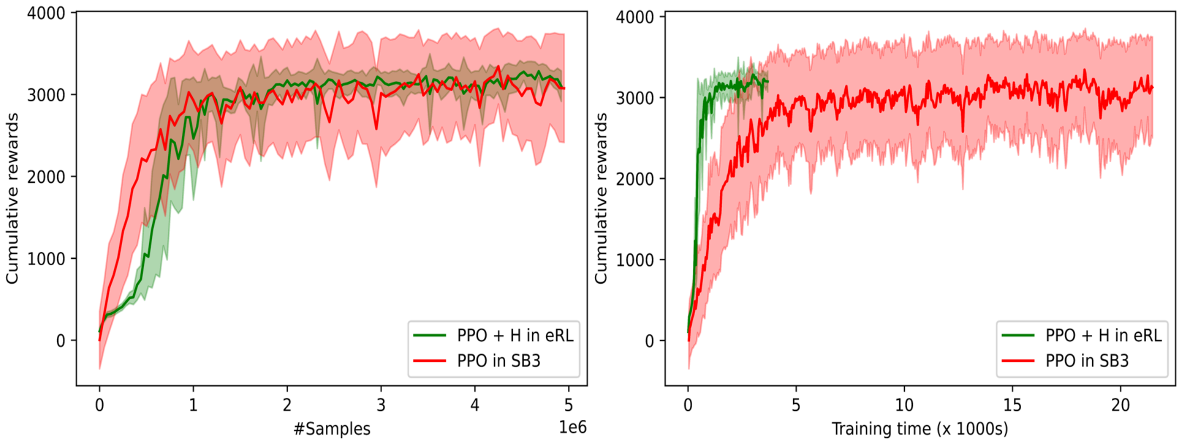
RL Frameworks with Parallel/Batch Environments

- EnvPool - a batched environment pool with pybind11 and thread pool.
    - It has high performance (~1M raw FPS with Atari games, ~3M raw FPS with Mujoco simulator on DGX-A100) and compatible APIs (supports both gym and dm_env, both sync and async, both single and multi player environment).
    - Compatible with OpenAI `gym` APIs and DeepMind `dm_env` APIs.
    - 1 Million Atari frames / 3 Million Mujoco steps per second simulation with 256 CPU cores, ~20x throughput of Python subprocess-based vector env.
    - ~3x throughput of Python subprocess-based vector env on low resource setup like 12 CPU cores.
    - Comparing with the existing GPU-based solution (Brax / Isaac-gym), EnvPool is a general solution for various kinds of speeding-up RL environment parallelization. Code examples: 15-min Atari Breakout, 5-min MuJoCo HalfCheetah
    - **CleanRL**: EnvPool is compatible with CleanRL - a Deep Reinforcement Learning library that provides high-quality single-file implementation with research-friendly features. For example, `ppo_atari.py` only has 340 lines of code but contains all implementation details on how PPO works with Atari games, so it is a great reference implementation to read for folks who do not wish to read an entire modular library.



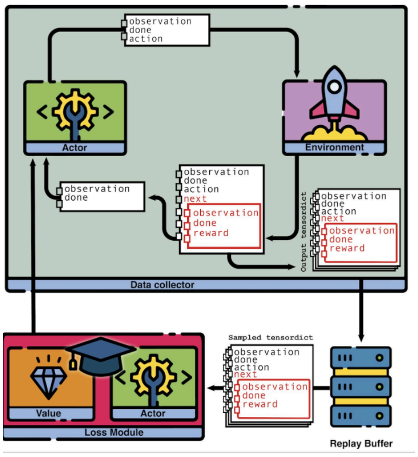
- [Stable-Baselines-3](#) - Stable Baselines3 (SB3) is a set of reliable implementations of reinforcement learning algorithms in PyTorch.
  - GitHub
  - RL Baselines3 Zoo provides a collection of pre-trained agents, scripts for training, evaluating agents, tuning hyperparameters, plotting results and recording videos.
  - Tips and Tricks with SB3 - YouTube Video
  - Vectorized Environments with Multiprocessing support - Vectorized Environments are a method for stacking multiple independent environments into a single environment.
  - SB3 with EnvPool or Isaac Gym
  - Multiprocessing Example with SB3

- ElegantRL - an open-source massively parallel framework for deep reinforcement learning (DRL) algorithms implemented in PyTorch.
  - GitHub
  - Scalability: ElegantRL fully exploits the parallelism of DRL algorithms at multiple levels, making it easily scale out to hundreds or thousands of computing nodes on a cloud platform.
  - Elastic: allows to elastically and automatically allocate computing resources on the cloud.
  - Efficient: in many testing cases (single GPU/multi-GPU/GPU cloud). "We find it more efficient than Ray RLlib".
  - Stable: "much much much more stable than Stable Baselines 3 by utilizing various ensemble methods".
  - VecEnvs - ElegantRL supports massively parallel simulation through GPU-accelerated VecEnv.
  - Supports Isaac Gym
  - PPO Example



- TorchRL - TorchRL is an open-source Reinforcement Learning (RL) library for PyTorch.
  - GitHub

- TorchRL provides pytorch and python-first, low and high level abstractions for RL that are intended to be efficient, modular, documented and properly tested.
- On the low-level end, torchrl comes with a set of highly re-usable functionals for cost functions, returns and data processing.
- Tutorials



Recommended Videos

- AMP - Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA
- Autimatic Mixed Precision (AMP) - NVIDIA - Automatic Mixed Precision Training in PyTorch
- PyTorch Performance Tuning Guide - Szymon Migacz, NVIDIA
- RL Tips and Tricks with Stable-Baselines-3
- Isaac Gym Tutorial

Credits

- Icons from Icons8.com - https://icons8.com
- Performance Tuning Guide - Szymon Migacz
- Explained Output of Nvidia-smi Utility - Shachi Kaul
- 7 Tips To Maximize PyTorch Performance - William Falcon
- Tricks for training PyTorch models to convergence more quickly - Aleksey Bilogur