Using a GPU

A GPU (Graphical Processing Unit) is a component of most modern computers that is designed to perform computations needed for 3D graphics. Their most common use is to perform these actions for video games, computing where polygons go to show the game to the user. With a lot of hand waving, a GPU is basically a large array of small processors, performing highly parallelised computation. You basically have a mini-supercomputer* running right now!

While each of the “CPUs” in a GPU is quite slow, there are a lot of them and they are specialised for numerical processing. This means a GPU can perform lots of simple numerical processing tasks at the same time. In a great stroke of luck, this is *exactly* what many machine learning algorithms need to do.

Don't have a GPU?

Most modern (last 10 years) computers have some form of GPU, even if it is built into your motherboard. For the purposes of this tutorial, this will be sufficient.

You'll need to know what type of graphics card you have. Windows users can follow these instructions, and users of other systems need to consult their system's documentation.

Non-Nvidia Graphics Card Users

While other graphics cards may be supportable, this tutorial is only test on a recent NVidia Graphics card. If your graphics card is of a different type, I recommend that you seek out a NVidia graphics card to learn, either buy or borrow. If this is really hard for you to do, contact your local University or School and see if they can help. If you are still having trouble, feel free to just read through, and to the work on a standard CPU instead. You'll be able to migrate the learnings over at a later date.

* Note: not really a supercomputer, but somewhat similar in many respects.

Installing GPU-enabled TensorFlow

If you didn’t install the GPU-enabled TensorFlow earlier then we need to do that first. Our instructions in Lesson 1 don’t say to, so if you didn’t go out of your way to enable GPU support than you didn’t.

I recommend that you create a new Anaconda environment for this, rather than try to update your previous one.

Before you start

Head to the official TensorFlow installation instructions, and follow the Anaconda Installation instructions. The main difference between this, and what we did in Lesson 1, is that you need the **GPU enabled**version of TensorFlow for your system. However, **before you install TensorFlow** into this environment, you need to setup your computer to be GPU enabled with CUDA and CuDNN. The official TensorFlow documentation outline this step by step, but I recommended this tutorialif you are trying to setup a recent Ubuntu install. The main reason is that, at the time of writing (July 2016), CUDA has not yet been built for the most recent Ubuntu version, which means the process is a lot more manual.

Using your GPU

It’s quite simple really. At least, syntactically. Just change this:

```
# Setup operations

with tf.Session() as sess:
    # Run your code
```

To this:

```
with tf.device("/gpu:0"):
    # Setup operations
```

```
with tf.Session() as sess:
    # Run your code
```

This new line will create a new context manager, telling TensorFlow to perform those actions on the GPU.

Let's have a look at a concrete example. The below code creates a random matrix with a size given at the command line. We can either run the code on a CPU or GPU using command line options:

```
import sys
import numpy as np
import tensorflow as tf
from datetime import datetime

device_name = sys.argv[1]  # Choose device from cmd line. Options: gpu or cpu
shape = (int(sys.argv[2]), int(sys.argv[2]))
if device_name == "gpu":
    device_name = "/gpu:0"
else:
    device_name = "/cpu:0"

with tf.device(device_name):
    random_matrix = tf.random_uniform(shape=shape, minval=0, maxval=1)
    dot_operation = tf.matmul(random_matrix, tf.transpose(random_matrix))
    sum_operation = tf.reduce_sum(dot_operation)


startTime = datetime.now()
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as session:
        result = session.run(sum_operation)
        print(result)

# It can be hard to see the results on the terminal with lots of output -- add some newlines to improve readability.
print("\n" * 5)
print("Shape:", shape, "Device:", device_name)
print("Time taken:", datetime.now() - startTime)

print("\n" * 5)
```

You can run this at the command line with:

`python matmul.py gpu 1500`

This will use the CPU with a matrix of size 1500 squared. Use the following to do the same operation on the CPU:

`python matmul.py cpu 1500`

The first thing you'll notice when running GPU-enabled code is a large increase in output, compared to a normal TensorFlow script. Here is what my computer prints out, *before* it prints out any result from the operations.

```
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcudnn.so.5 locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I tensorflow/stream_executor/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:925] successful NUMA node read from SysFS had negative value (-1), but there must b
e at least one NUMA node, so returning NUMA node zero
I tensorflow/core/common_runtime/gpu/gpu_init.cc:102] Found device 0 with properties:
name: GeForce GTX 950M
major: 5 minor: 0 memoryClockRate (GHz) 1.124
pciBusID 0000:01:00.0
Total memory: 3.95GiB
Free memory: 3.50GiB
I tensorflow/core/common_runtime/gpu/gpu_init.cc:126] DMA: 0
I tensorflow/core/common_runtime/gpu/gpu_init.cc:136] 0:   Y
I tensorflow/core/common_runtime/gpu/gpu_device.cc:838] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX 950M, pci bus
 id: 0000:01:00.0)
```

If your code doesn't produce output similar in nature to this, you aren't running the GPU enabled Tensorflow. Alternatively, if you get an error such as `ImportError: libcudart.so.7.5: cannot open shared object file: No such file or directory`, then you haven't installed the CUDA library properly. In this case, you'll need to go back to follow the instructions for installing CUDA on your system.

Try running the above code on both the CPU and GPU, increasing the number slowly. Start with 1500, then try 3000, then 4500, and so on. You'll find that the CPU starts taking quite a long time, while the GPU is really, really fast at this operation!

If you have multiple GPUs, you can use either. GPUs are zero-indexed - the above code accesses the first GPU. Changing the device to `gpu:1` uses the second GPU, and so on. You can also send part of your computation to one GPU, and part to another GPU. In addition, you can access the CPUs of your machine in a similar way – just use `cpu:0` (or another number).

What types of operations should I send to the GPU?

In general, if the step of the process can be described such as “do this mathematical operation thousands of times”, then send it to the GPU. Examples include matrix multiplication and computing the inverse of a matrix. In fact, many basic matrix operations are prime candidates for GPUs. As an overly broad and simple rule, other operations should be performed on the CPU.

There is also a cost to changing devices and using GPUs. GPUs don’t have direct access to the rest of your computer (except, of course for the display). Due to this, if you are running a command on a GPU, you need to copy all of the data to the GPU first, then do the operation, then copy the result back to your computer’s main memory. TensorFlow handles this under the hood, so the code is simple, but the work still needs to be performed.

Not all operations can be done on GPUs. If you get the following error, you are trying to do an operation that can’t be done on a GPU:

```
Cannot assign a device to node 'PyFunc': Could not satisfy explicit device specification '/device:GPU:1' because no devices matching that
 specification are registered in this process;
```

If this is the case, you can either manually change the device to a CPU for this operation, or set TensorFlow to automatically change the device in this case. To do this, set `allow_soft_placement` tp `True` in the configuration, done as part of creating the session. The prototype looks like this:

```
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True)):
    # Run your graph here
```

I also recommend logging device placement when using GPUs, at this lets you easily debug issues relating to different device usage. This prints the usage of devices to the log, allowing you to see when devices change and how that affects the graph.

```
with tf.Session(config=tf.ConfigProto(allow_soft_placement=True, log_device_placement=True)):
    # Run your graph here
```

Exercises

1) Setup your computer to use the GPU for TensorFlow (or find a computer to lend if you don’t have a recent GPU).

2) Try running the previous exercise solutions on the GPU. Which operations can be performed on a GPU, and which cannot?

3) Build a program that uses operations on both the GPU and the CPU. Use the profiling code we saw in Lesson 5 to estimate the impact of sending data to, and retrieving data from, the GPU.

4) Send me your code! I’d love to see examples of your code, how you use Tensorflow, and any tricks you have found.