TensorFlow's support of distributed computing is one of its main highlights compared to other neural network frameworks. It gives you full control over how to split (or replicate) your computation graph across devices and servers, and it lets you parallelize and synchronize operations in flexible ways so you can choose between all sorts of parallelization approaches.

We will look at some of the most popular approaches to parallelizing the execution and training of a neural network. Instead of waiting for weeks for a training algorithm to complete, you may end up waiting for just a few hours. Not only does this save an enormous amount of time, it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

Other great use cases of parallelization include exploring a much larger hyperparameter space when fine-tuning your model, and running large ensembles of neural networks efficiently.

But we must learn to walk before we can run. Let's start by parallelizing simple graphs across several GPUs on a single machine.

Multiple Devices on a Single Machine

You can often get a major performance boost simply by adding GPU cards to a single machine. In fact, in many cases this will suffice; you won't need to use multiple machines at all. For example, you can typically train a neural network just as fast using 8 GPUs on a single machine rather than 16 GPUs across multiple machines (due to the extra delay imposed by network communications in a multimachine setup).

In this section we will look at how to set up your environment so that TensorFlow can use multiple GPU cards on one machine. Then we will look at how you can distribute operations across available devices and execute them in parallel.

Installation

In order to run TensorFlow on multiple GPU cards, you first need to make sure your GPU cards have NVidia Compute Capability (greater or equal to 3.0). This includes Nvidia's Titan, Titan X, K20, and K40 cards (if you own another card, you can check its compatibility at *https://developer.nvidia.com/cuda-gpus*).

If you don't own any GPU cards, you can use a hosting service with GPU capability such as Amazon AWS. Detailed instructions to set up TensorFlow 0.9 with Python 3.5 on an Amazon AWS GPU instance are available in Žiga Avsec's helpful blog post. It should not be too hard to update it to the latest version of TensorFlow. Google also released a cloud service called *Cloud Machine Learning* to run TensorFlow graphs. In May 2016, they announced that their platform now includes servers equipped with *tensor processing units* (TPUs), processors specialized for Machine Learning that are much faster than GPUs for many ML tasks. Of course, another option is simply to buy your own GPU card. Tim Dettmers wrote a great blog post to help you choose, and he updates it fairly regularly.

You must then download and install the appropriate version of the CUDA and cuDNN libraries (CUDA 8.0 and cuDNN 5.1 if you are using the binary installation of TensorFlow 1.0.0), and set a few environment variables so TensorFlow knows where to find CUDA and cuDNN. The detailed installation instructions are likely to change fairly quickly, so it is best that you follow the instructions on TensorFlow's website.

Nvidia's *Compute Unified Device Architecture* library (CUDA) allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration). Nvidia's *CUDA Deep Neural Network* library (cuDNN) is a GPU-accelerated library of primitives for DNNs. It provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see Chapter 13). It is part of Nvidia's Deep Learning SDK (note that it requires creating an Nvidia developer account in order to download it). TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see Figure 12-2).

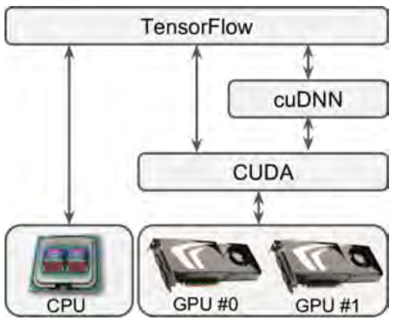

Figure 12-2. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

You can use the `nvidia-smi` command to check that CUDA is properly installed. It lists the available GPU cards, as well as processes running on each card:

```
$ nvidia-smi
Wed Sep 16 09:50:03 2016
+------------------------------------------------------+
| NVIDIA-SMI 352.63     Driver Version: 352.63         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GRID K520           Off  | 0000:00:03.0     Off |                  N/A |
| N/A   27C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Finally, you must install TensorFlow with GPU support. If you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH              # Your ML working directory (e.g., $HOME/ml)
$ source env/bin/activate
```

Then install the appropriate GPU-enabled version of TensorFlow:

```
$ pip3 install --upgrade tensorflow-gpu
```

Now you can open up a Python shell and check that TensorFlow detects and uses CUDA and cuDNN properly by importing TensorFlow and creating a session:

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0:   Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

Looks good! TensorFlow detected the CUDA and cuDNN libraries, and it used the CUDA library to detect the GPU card (in this case an Nvidia Grid K520 card).

Managing the GPU RAM

By default TensorFlow automatically grabs all the RAM in all available GPUs the first time you run a graph, so you will not be able to start a second TensorFlow program while the first one is still running. If you try, you will get the following error:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from
device: CUDA_ERROR_OUT_OF_MEMORY
```

One solution is to run each process on different GPU cards. To do this, the simplest option is to set the CUDA_VISIBLE_DEVICES environment variable so that each process only sees the appropriate GPU cards. For example, you could start two programs like this:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program #1 will only see GPU cards 0 and 1 (numbered 0 and 1, respectively), and program #2 will only see GPU cards 2 and 3 (numbered 1 and 0, respectively). Everything will work fine (see Figure 12-3).

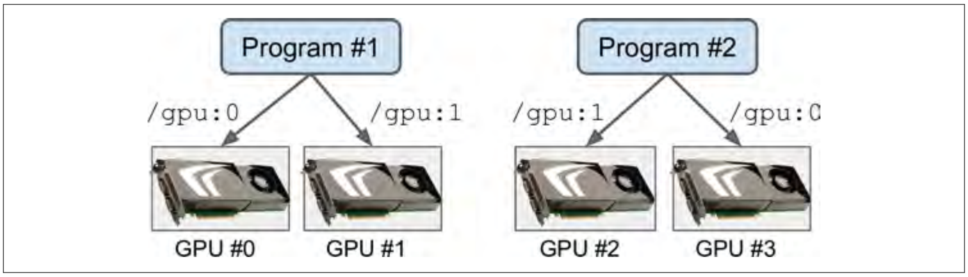

Figure 12-3. Each program gets two GPUs for itself

Another option is to tell TensorFlow to grab only a fraction of the memory. For example, to make TensorFlow grab only 40% of each GPU's memory, you must create a ConfigProto object, set its gpu_options.per_process_gpu_memory_fraction option to 0.4, and create the session using this configuration:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
```

Now two programs like this one can run in parallel using the same GPU cards (but not three, since $3 \times 0.4 > 1$). See Figure 12-4.