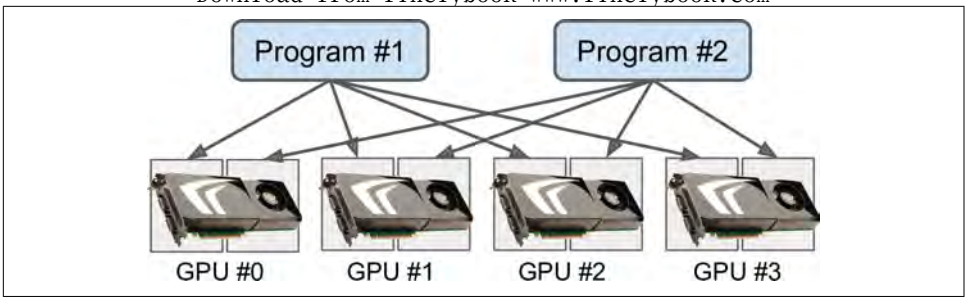

Figure 12-4. Each program gets all four GPUs, but with only 40% of the RAM each

If you run the `nvidia-smi` command while both programs are running, you should see that each process holds roughly 40% of the total RAM of each card:

```
$ nvidia-smi
[...]
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0      5231    C   python                                       1677MiB |
|    0      5262    C   python                                       1677MiB |
|    1      5231    C   python                                       1677MiB |
|    1      5262    C   python                                       1677MiB |
[...]
```

Yet another option is to tell TensorFlow to grab memory only when it needs it. To do this you must set `config.gpu_options.allow_growth` to `True`. However, TensorFlow never releases memory once it has grabbed it (to avoid memory fragmentation) so you may still run out of memory after a while. It may be harder to guarantee a deterministic behavior using this option, so in general you probably want to stick with one of the previous options.

Okay, now you have a working GPU-enabled TensorFlow installation. Let's see how to use it!

Placing Operations on Devices

The TensorFlow whitepaper[1] presents a friendly *dynamic placer* algorithm that automagically distributes operations across all available devices, taking into account things like the measured computation time in previous runs of the graph, estimations of the size of the input and output tensors to each operation, the amount of RAM available in each device, communication delay when transferring data in and out of

1 "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Google Research (2015).

devices, hints and constraints from the user, and more. Unfortunately, this sophisticated algorithm is internal to Google; it was not released in the open source version of TensorFlow. The reason it was left out seems to be that in practice a small set of placement rules specified by the user actually results in more efficient placement than what the dynamic placer is capable of. However, the TensorFlow team is working on improving the dynamic placer, and perhaps it will eventually be good enough to be released.

Until then TensorFlow relies on the *simple placer*, which (as its name suggests) is very basic.

Simple placement

Whenever you run a graph, if TensorFlow needs to evaluate a node that is not placed on a device yet, it uses the simple placer to place it, along with all other nodes that are not placed yet. The simple placer respects the following rules:

- If a node was already placed on a device in a previous run of the graph, it is left on that device.
- Else, if the user *pinned* a node to a device (described next), the placer places it on that device.
- Else, it defaults to GPU #0, or the CPU if there is no GPU.

As you can see, placing operations on the appropriate device is mostly up to you. If you don't do anything, the whole graph will be placed on the default device. To pin nodes onto a device, you must create a device block using the `device()` function. For example, the following code pins the variable `a` and the constant `b` on the CPU, but the multiplication node `c` is not pinned on any device, so it will be placed on the default device:

```
with tf.device("/cpu:0"):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)

c = a * b
```

The `"/cpu:0"` device aggregates all CPUs on a multi-CPU system. There is currently no way to pin nodes on specific CPUs or to use just a subset of all CPUs.

Logging placements

Let's check that the simple placer respects the placement constraints we have just defined. For this you can set the `log_device_placement` option to `True`; this tells the placer to log a message whenever it places a node. For example:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520,
pci bus id: 0000:00:03.0)
[...]
>>> x.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

The lines starting with `"I"` for Info are the log messages. When we create a session, TensorFlow logs a message to tell us that it has found a GPU card (in this case the Grid K520 card). Then the first time we run the graph (in this case when initializing the variable `a`), the simple placer is run and places each node on the device it was assigned to. As expected, the log messages show that all nodes are placed on `"/cpu:0"` except the multiplication node, which ends up on the default device `"/gpu:0"` (you can safely ignore the prefix `/job:localhost/replica:0/task:0` for now; we will talk about it in a moment). Notice that the second time we run the graph (to compute `c`), the placer is not used since all the nodes TensorFlow needs to compute `c` are already placed.

Dynamic placement function

When you create a device block, you can specify a function instead of a device name. TensorFlow will call this function for each operation it needs to place in the device block, and the function must return the name of the device to pin the operation on. For example, the following code pins all the variable nodes to `"/cpu:0"` (in this case just the variable `a`) and all other nodes to `"/gpu:0"`:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
```

```
b = tf.constant(4.0)
c = a * b
```

You can easily implement more complex algorithms, such as pinning variables across GPUs in a round-robin fashion.

Operations and kernels

For a TensorFlow operation to run on a device, it needs to have an implementation for that device; this is called a *kernel*. Many operations have kernels for both CPUs and GPUs, but not all of them. For example, TensorFlow does not have a GPU kernel for integer variables, so the following code will fail when TensorFlow tries to place the variable `i` on GPU #0:

```
>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device
to node 'Variable': Could not satisfy explicit device specification
```

Note that TensorFlow infers that the variable must be of type `int32` since the initialization value is an integer. If you change the initialization value to `3.0` instead of `3`, or if you explicitly set `dtype=tf.float32` when creating the variable, everything will work fine.

Soft placement

By default, if you try to pin an operation on a device for which the operation has no kernel, you get the exception shown earlier when TensorFlow tries to place the operation on the device. If you prefer TensorFlow to fall back to the CPU instead, you can set the `allow_soft_placement` configuration option to `True`:

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)

config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
sess.run(i.initializer)  # the placer runs and falls back to /cpu:0
```

So far we have discussed how to place nodes on different devices. Now let's see how TensorFlow will run these nodes in parallel.

Parallel Execution

When TensorFlow runs a graph, it starts by finding out the list of nodes that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow