```
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

then starts evaluating the nodes with zero dependencies (i.e., source nodes). If these nodes are placed on separate devices, they obviously get evaluated in parallel. If they are placed on the same device, they get evaluated in different threads, so they may run in parallel too (in separate GPU threads or CPU cores).

TensorFlow manages a thread pool on each device to parallelize operations (see Figure 12-5). These are called the *inter-op thread pools*. Some operations have multithreaded kernels: they can use other thread pools (one per device) called the *intra-op thread pools*.

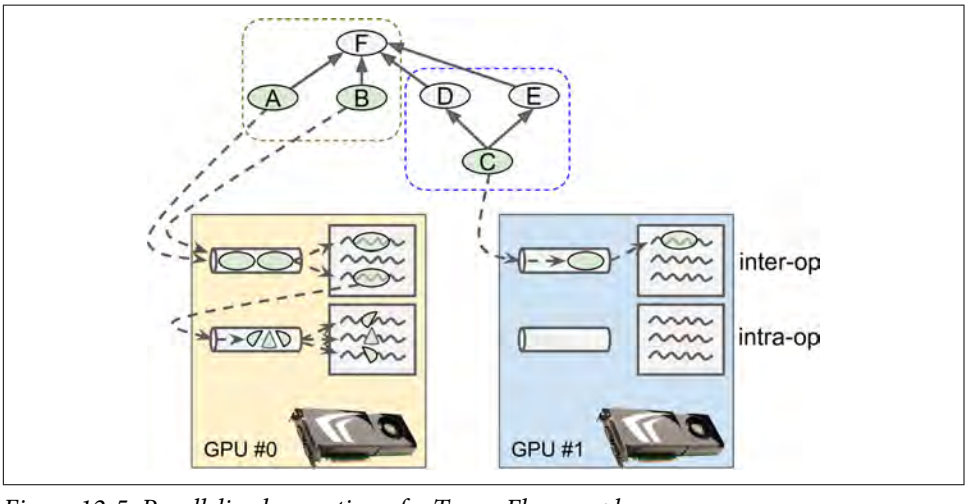

Figure 12-5. Parallelized execution of a TensorFlow graph

For example, in Figure 12-5, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on GPU #0, so they are sent to this device's inter-op thread pool, and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split in three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #1's inter-op thread pool.

As soon as operation C finishes, the dependency counters of operations D and E will be decremented and will both reach 0, so both operations will be sent to the inter-op thread pool to be executed.

You can control the number of threads per inter-op pool by setting the `inter_op_parallelism_threads` option. Note that the first session you start creates the inter-op thread pools. All other sessions will just reuse them unless you set the `use_per_session_threads` option to `True`. You can control the number of threads per intra-op pool by setting the `intra_op_parallelism_threads` option.

Control Dependencies

In some cases, it may be wise to postpone the evaluation of an operation even though all the operations it depends on have been executed. For example, if it uses a lot of memory but its value is needed only much further in the graph, it would be best to evaluate it at the last moment to avoid needlessly occupying RAM that other operations may need. Another example is a set of operations that depend on data located outside of the device. If they all run at the same time, they may saturate the device's communication bandwidth, and they will end up all waiting on I/O. Other operations that need to communicate data will also be blocked. It would be preferable to execute these communication-heavy operations sequentially, allowing the device to perform other operations in parallel.

To postpone evaluation of some nodes, a simple solution is to add *control dependencies*. For example, the following code tells TensorFlow to evaluate `x` and `y` only after `a` and `b` have been evaluated:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Obviously, since `z` depends on `x` and `y`, evaluating `z` also implies waiting for `a` and `b` to be evaluated, even though it is not explicitly in the `control_dependencies()` block. Also, since `b` depends on `a`, we could simplify the preceding code by just creating a control dependency on `[b]` instead of `[a, b]`, but in some cases "explicit is better than implicit."

Great! Now you know:

- How to place operations on multiple devices in any way you please
- How these operations get executed in parallel
- How to create control dependencies to optimize parallel execution

It's time to distribute computations across multiple servers!

Multiple Devices Across Multiple Servers

To run a graph across multiple servers, you first need to define a *cluster*. A cluster is composed of one or more TensorFlow servers, called *tasks*, typically spread across several machines (see Figure 12-6). Each task belongs to a *job*. A job is just a named group of tasks that typically have a common role, such as keeping track of the model