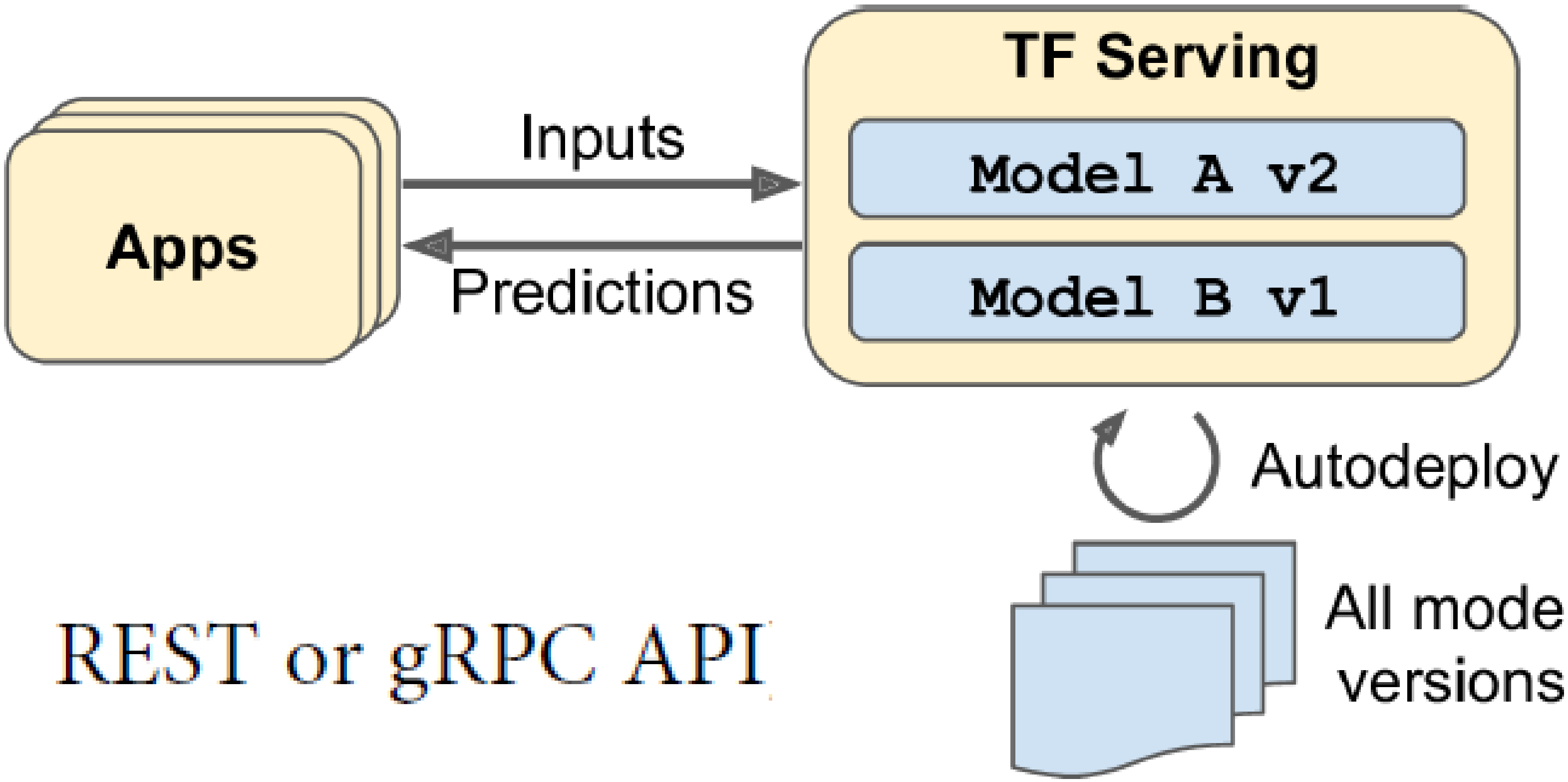
TensorFlow Models at Scale

Saeed Sharifian

Serving a TensorFlow Model



REST or gRPC API

Docker container

Exporting Saved Models

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

TF Serving through the REST API

```python
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

```python
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

```python
import requests

SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

```python
>>> y_proba = np.array(response["predictions"])
```

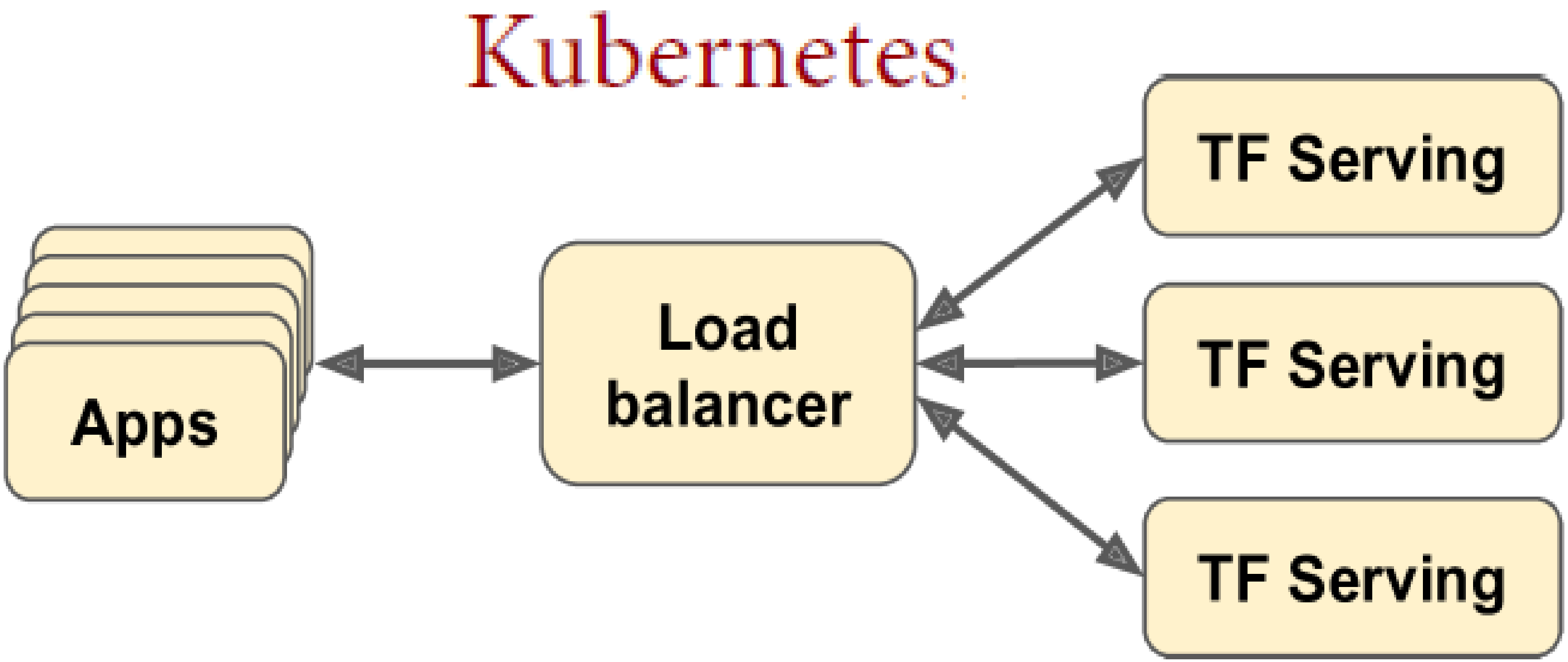
TF Serving through the gRPC API

```python
from tensorflow_serving.apis.predict_pb2 import PredictRequest
request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

```python
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc
channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

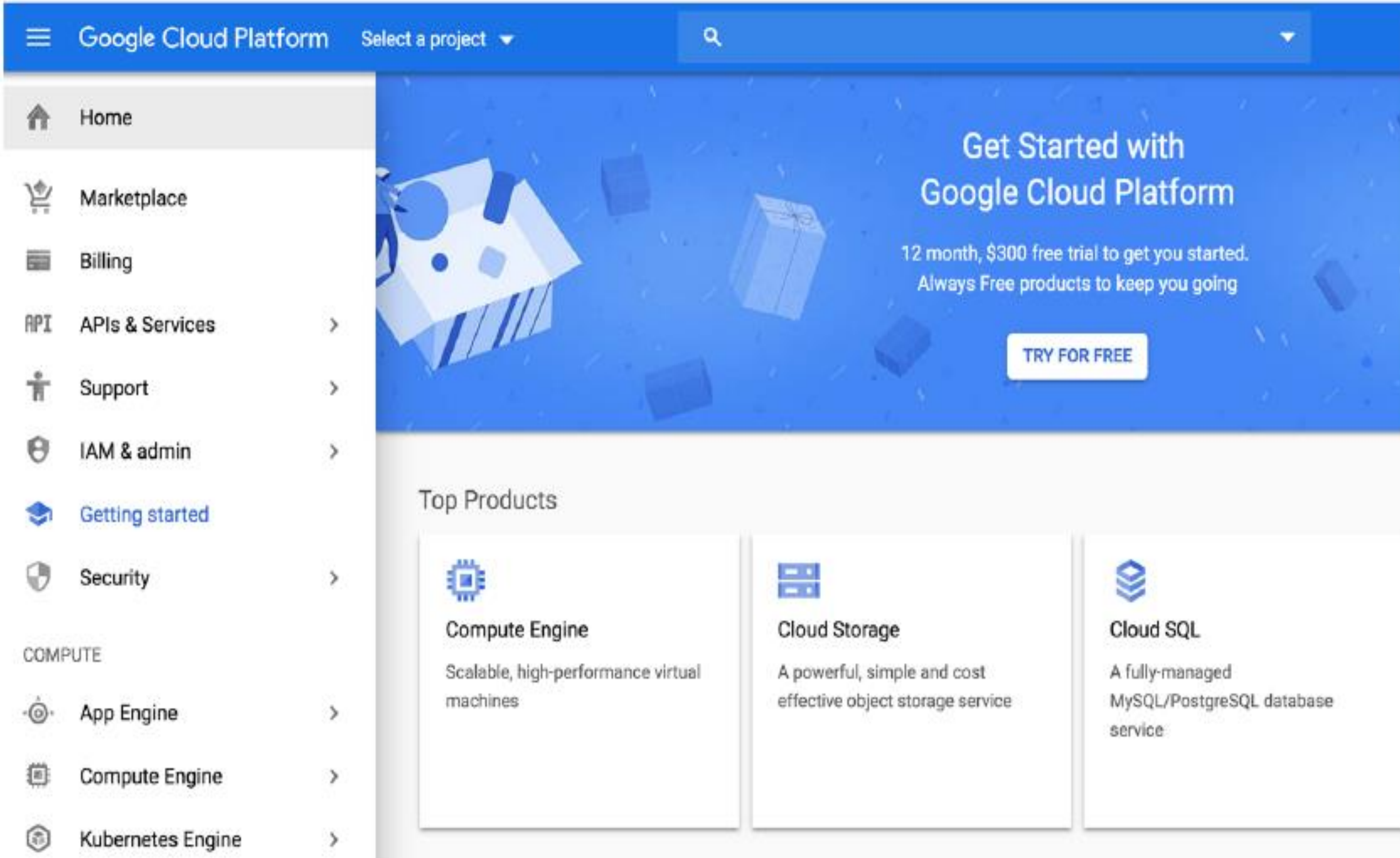
```python
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

Scaling up TF Serving



AutoML, Vision API, Natural Language API

Prediction Service on GCP AI Platform



Google API Client Library

Google Cloud Client Libraries

TFLite, Mobile or Embedded Device

- Reduce the model size, to shorten download time and reduce RAM usage.
- Reduce the amount of computations needed for each prediction, to reduce latency, battery usage, and heating.
- Adapt the model to device-specific constraints.

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```

load FlatBuffers straight to RAM

smaller bit-widths

floats (16 bits) rather than regular floats (32 bits)

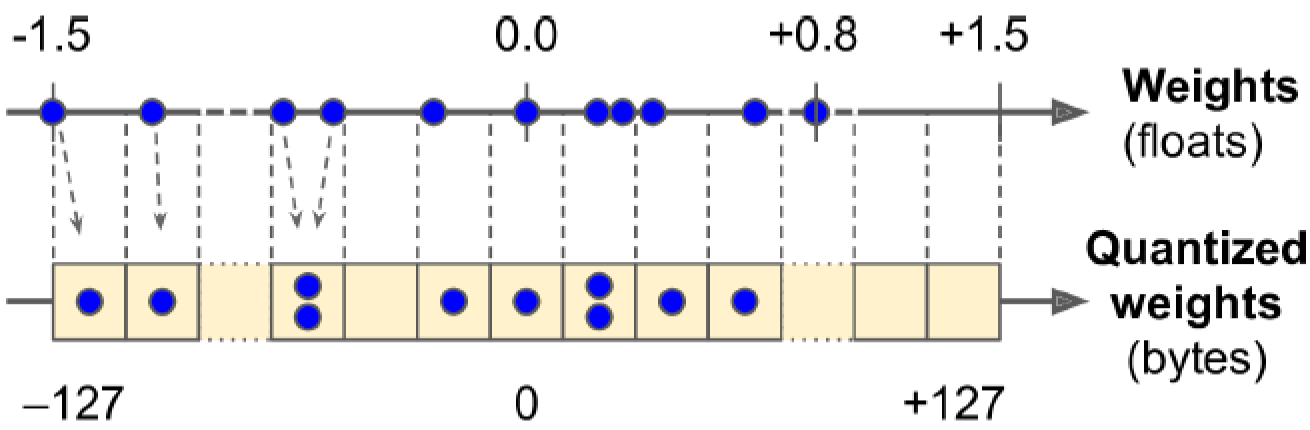
TFLite, Mobile or Embedded Device

post-training quantization

quantizing the model weights down to fixed-point, 8-bit integers

finds the maximum absolute weight value, m,

floating-point range $-m$ to $+m$ to the fixed-point (integer) range -127 to $+127$

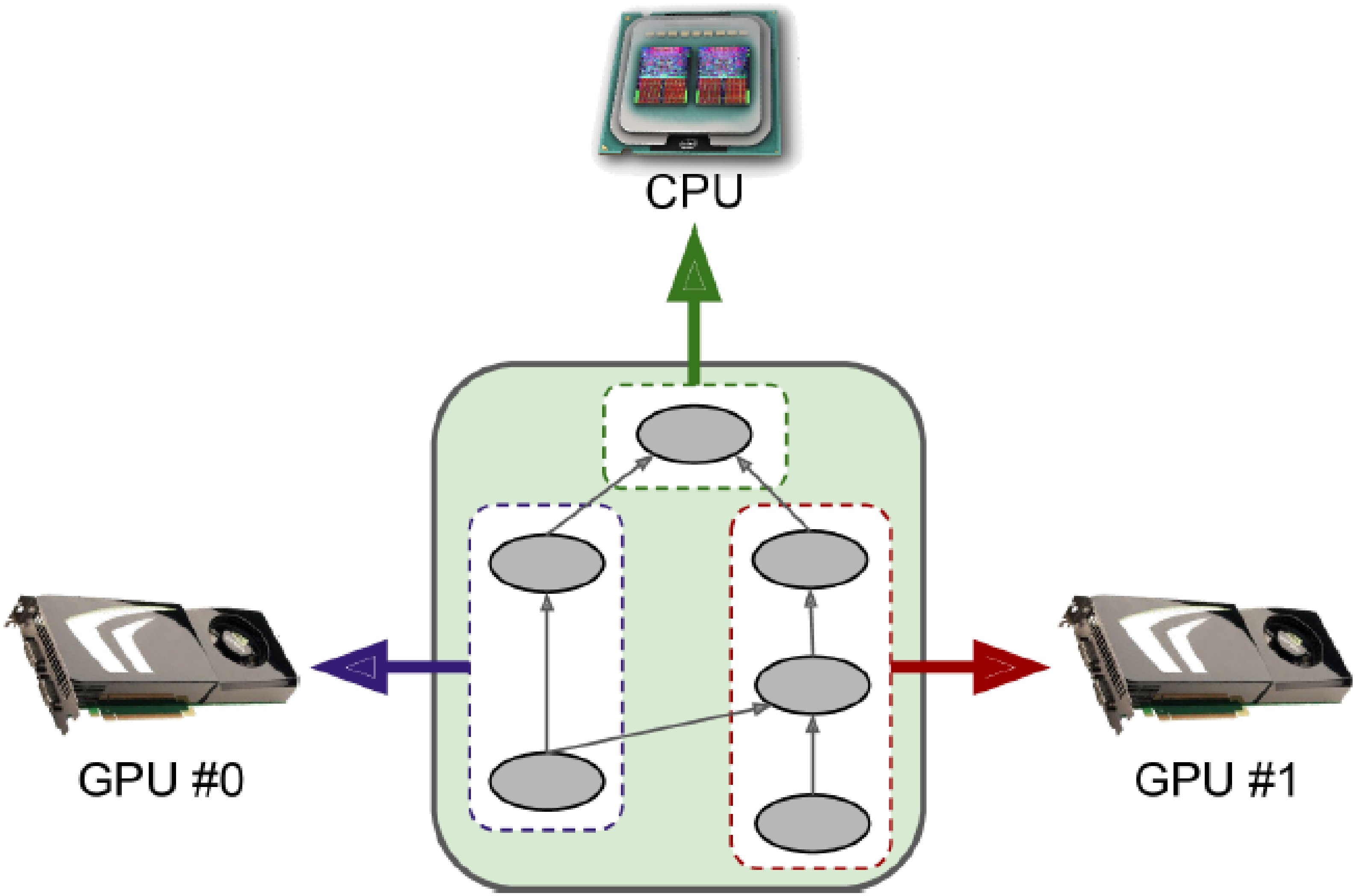



```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

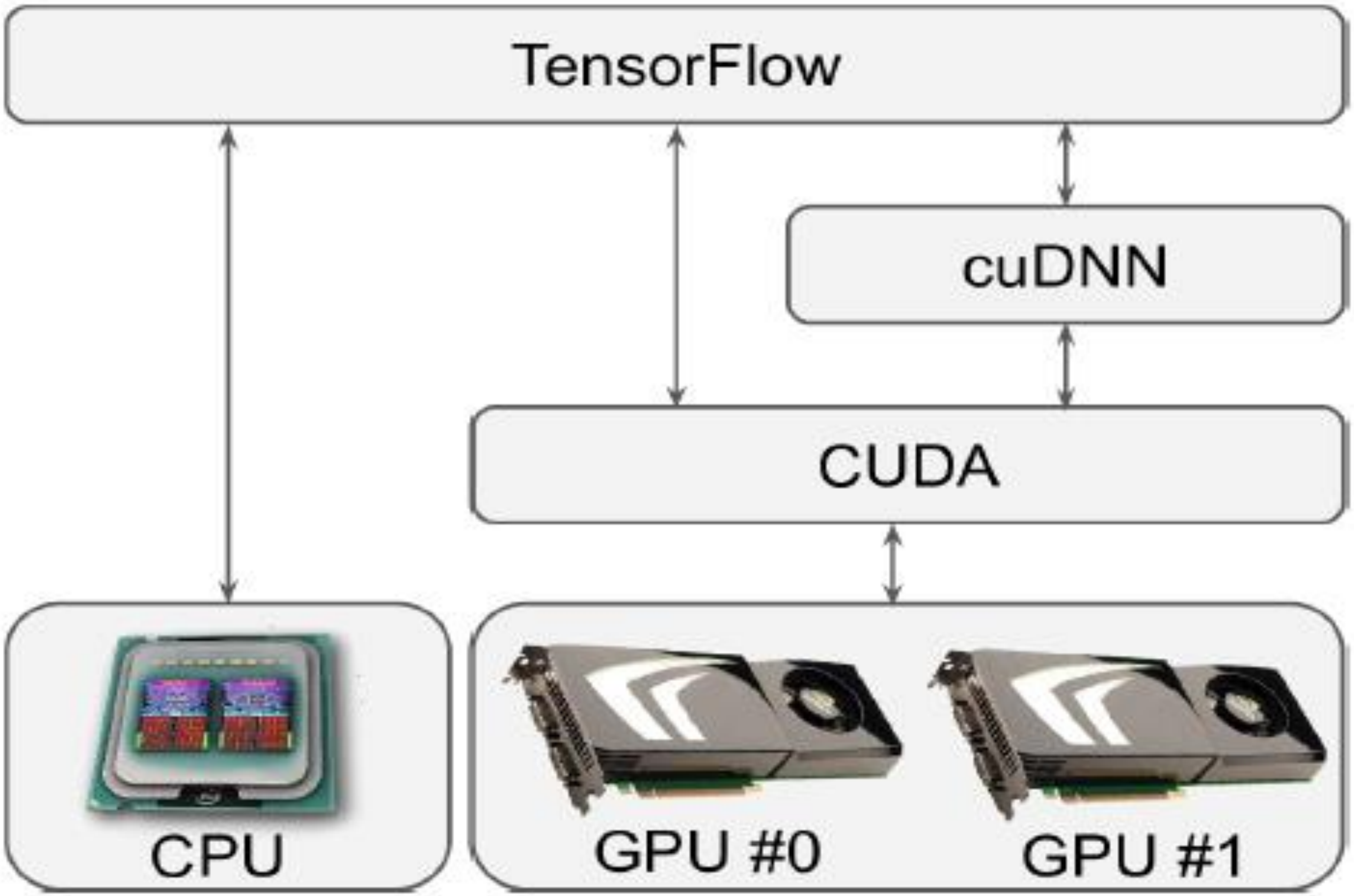
TensorFlow in the Browser

- TensorFlow.js JavaScript library
- TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers, by Pete Warden
- Practical Deep Learning for Cloud, Mobile, and Edge, by Anirudh Koul

Jsing GPUs to Speed Up Computations



TensorFlow uses CUDA and cuDNN



Nvidia cards with CUDA Compute Capability 3.5+

Compute Unified Device Architecture library (CUDA)

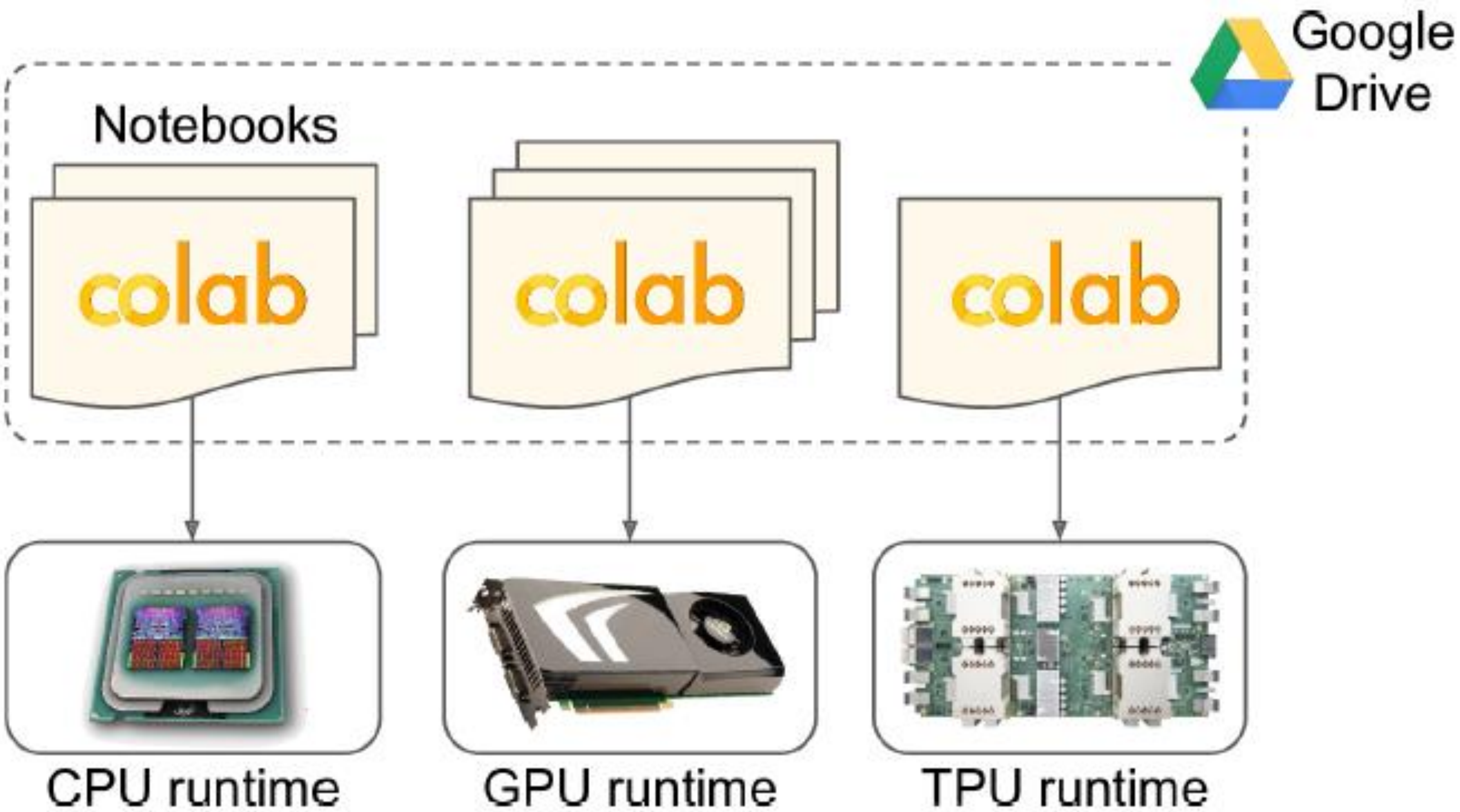
Nvidia's Deep Learning SDK

TensorFlow uses CUDA and cuDNN

```
$ nvidia-smi
Sun Jun  2 10:05:22 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 410.79       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P8    17W /  70W |      0MiB / 15079MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

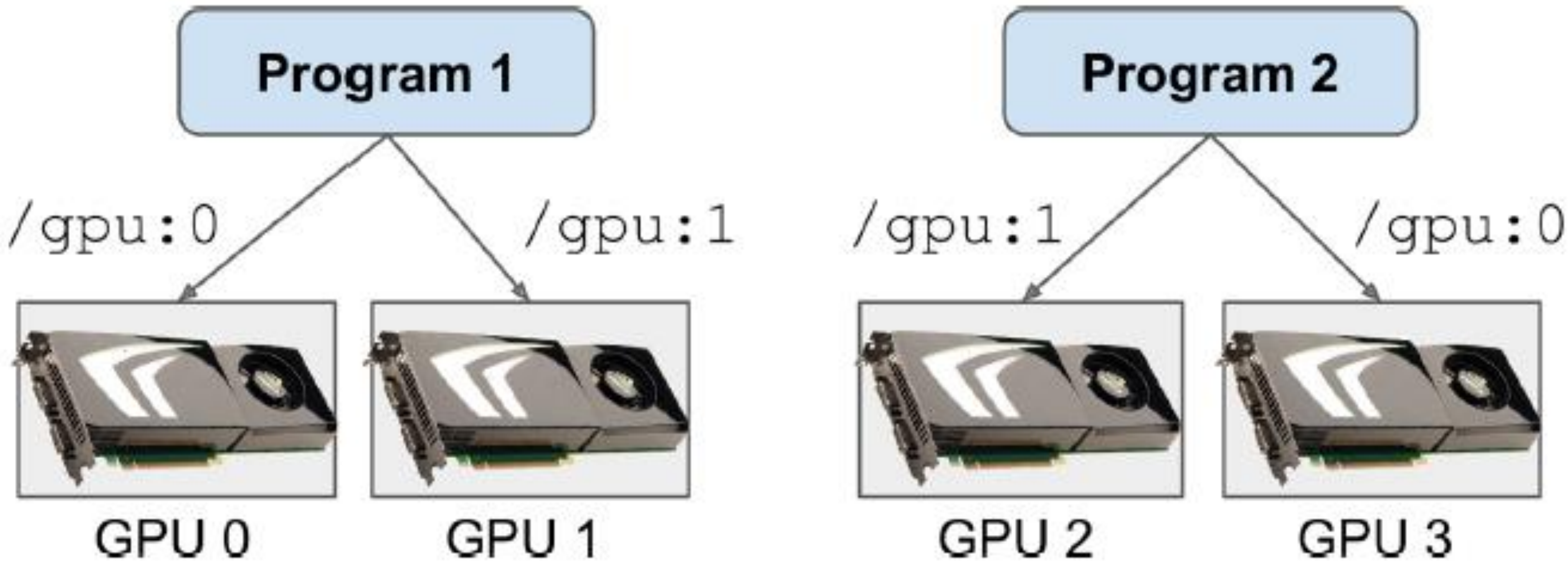
```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Colab Runtimes and notebooks



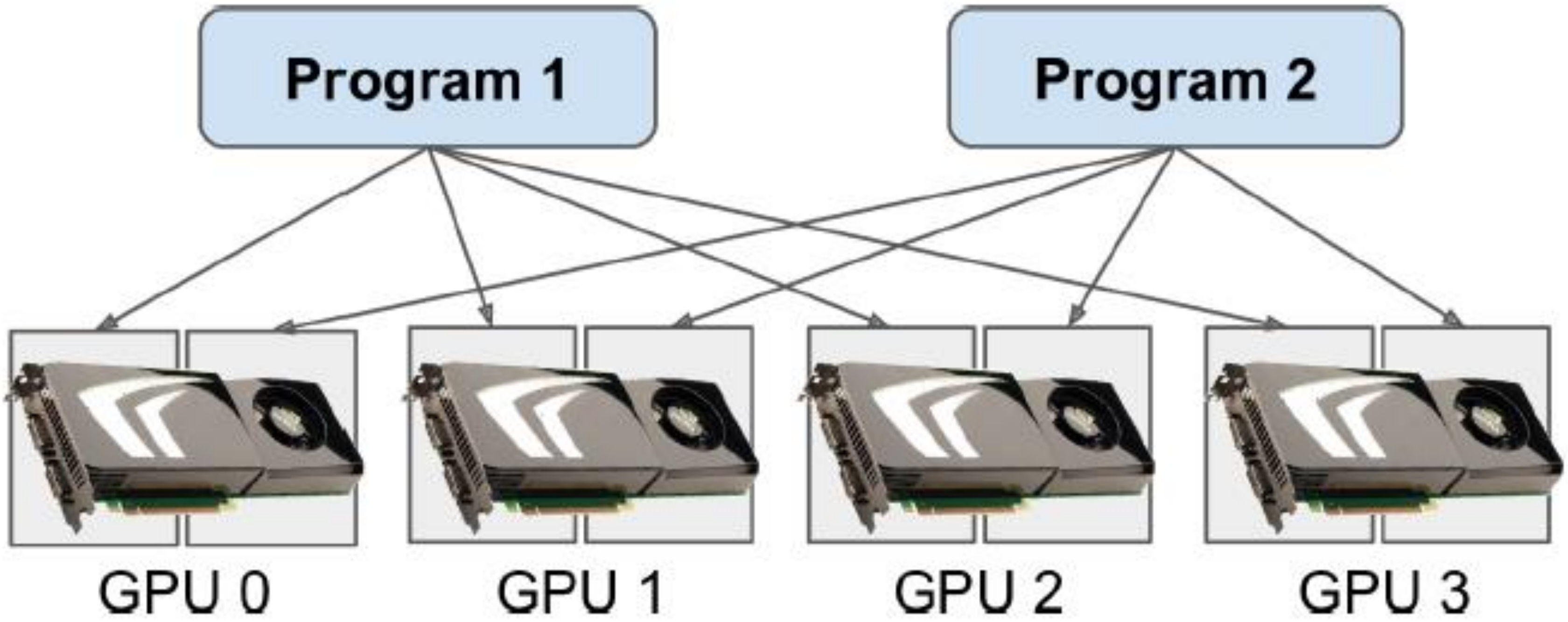
automatically shut down after 12 hours

Managing the GPU RAM




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

Grab a specific amount of GPU RAM




```python
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_virtual_device_configuration(
        gpu,
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

virtual GPU device (also called a *logical GPU device*)

Placing Operations and Variables on Devices

dynamic placer

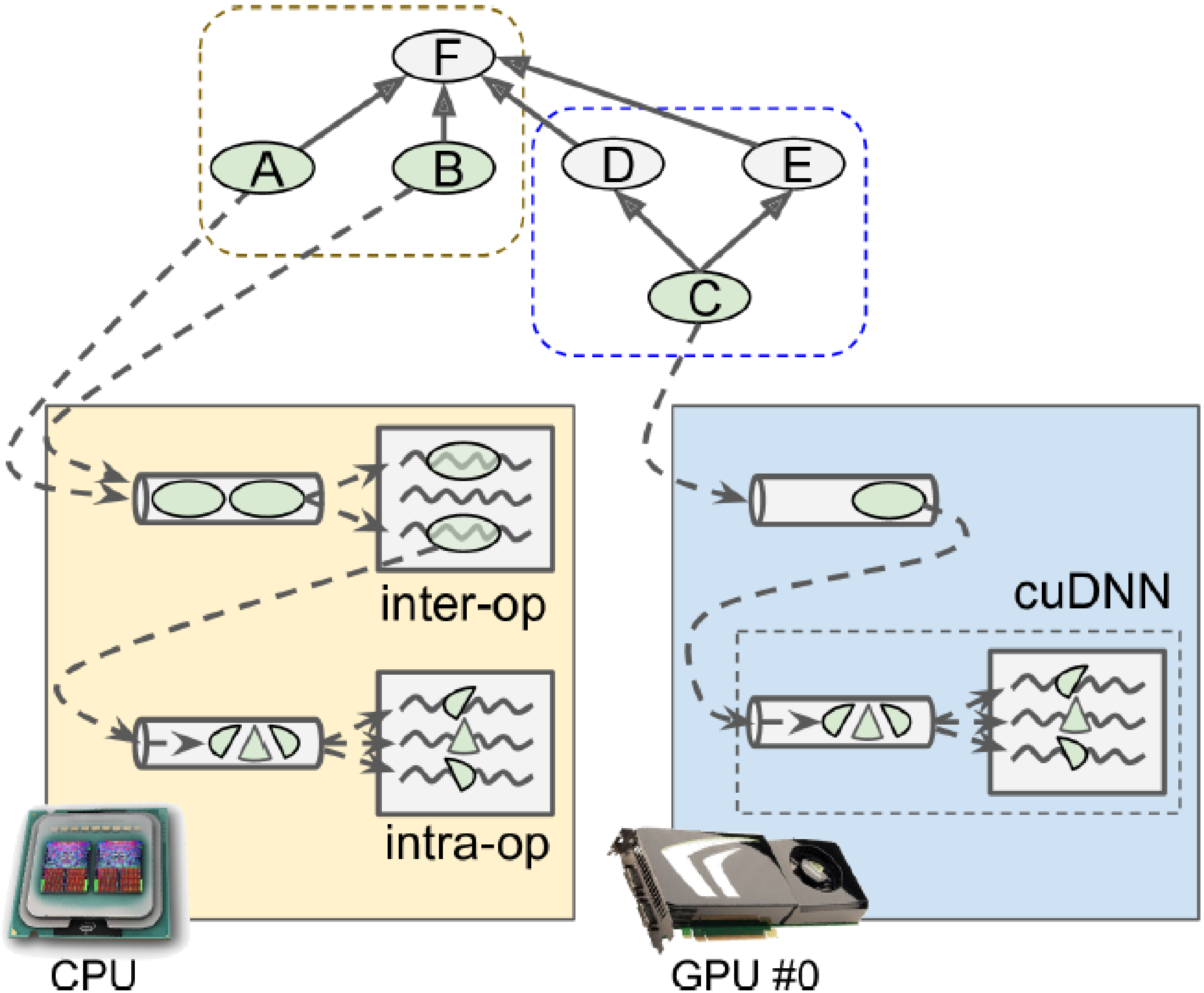
- computation time in previous runs
- available RAM in each device
- hints and constraints from the user
- size of the input and output
- communication delay

By default, all variables and all operations will be placed on the first GPU (named `/gpu:0`), except for variables and operations that don't have a GPU kernel

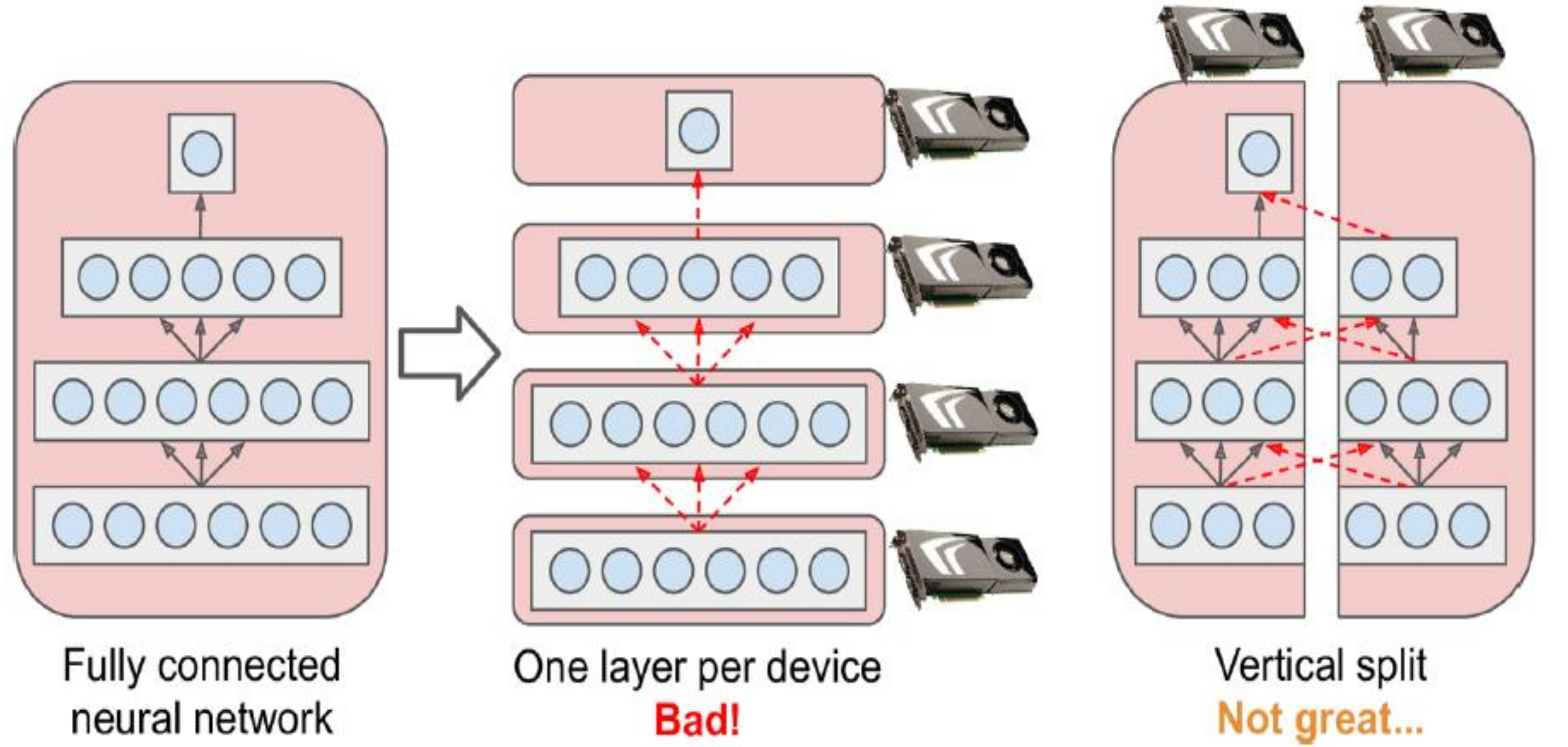
```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

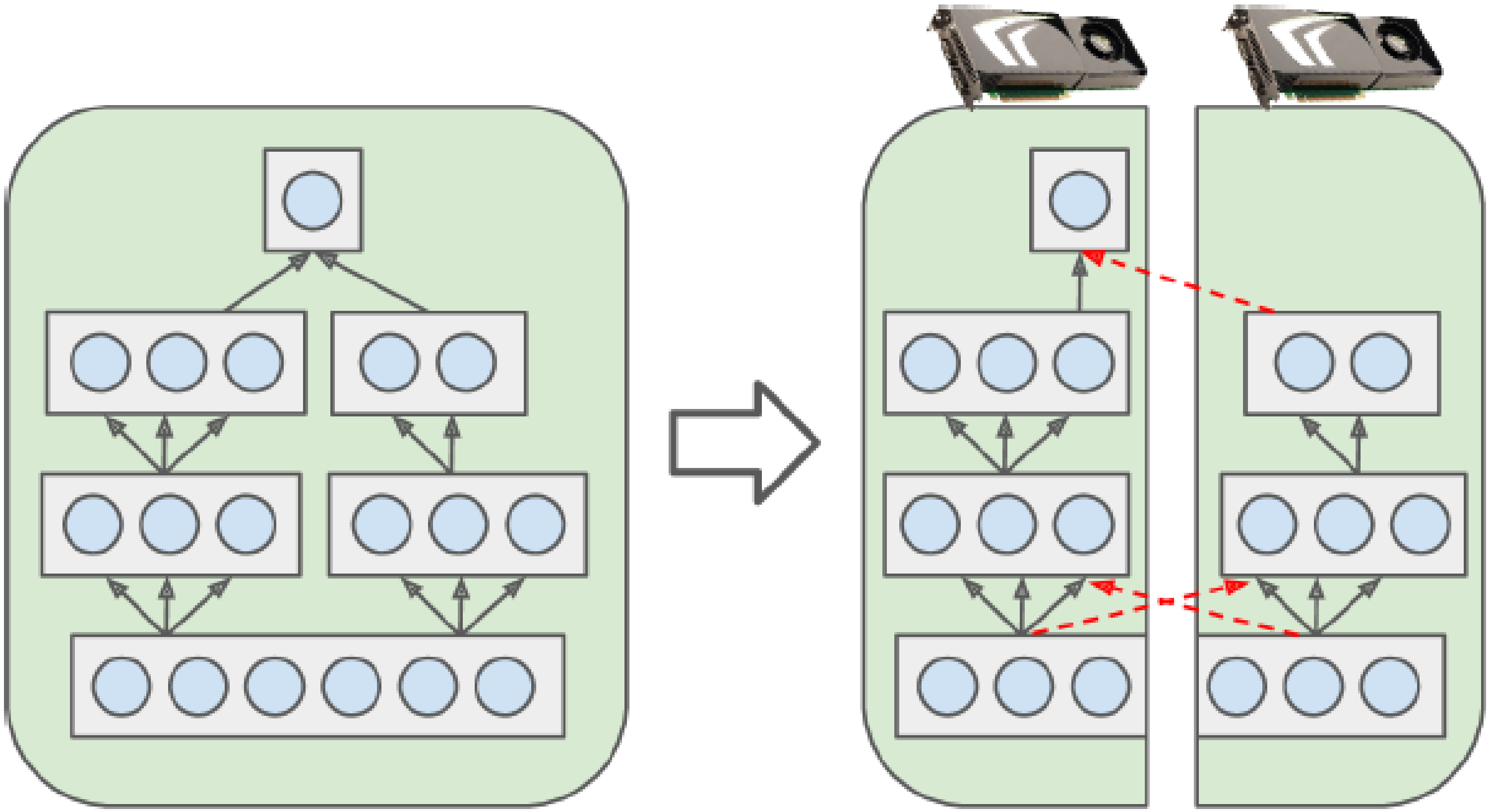
Parallelized execution of a TensorFlow graph



Training Models Across Multiple Devices
Model Parallelism



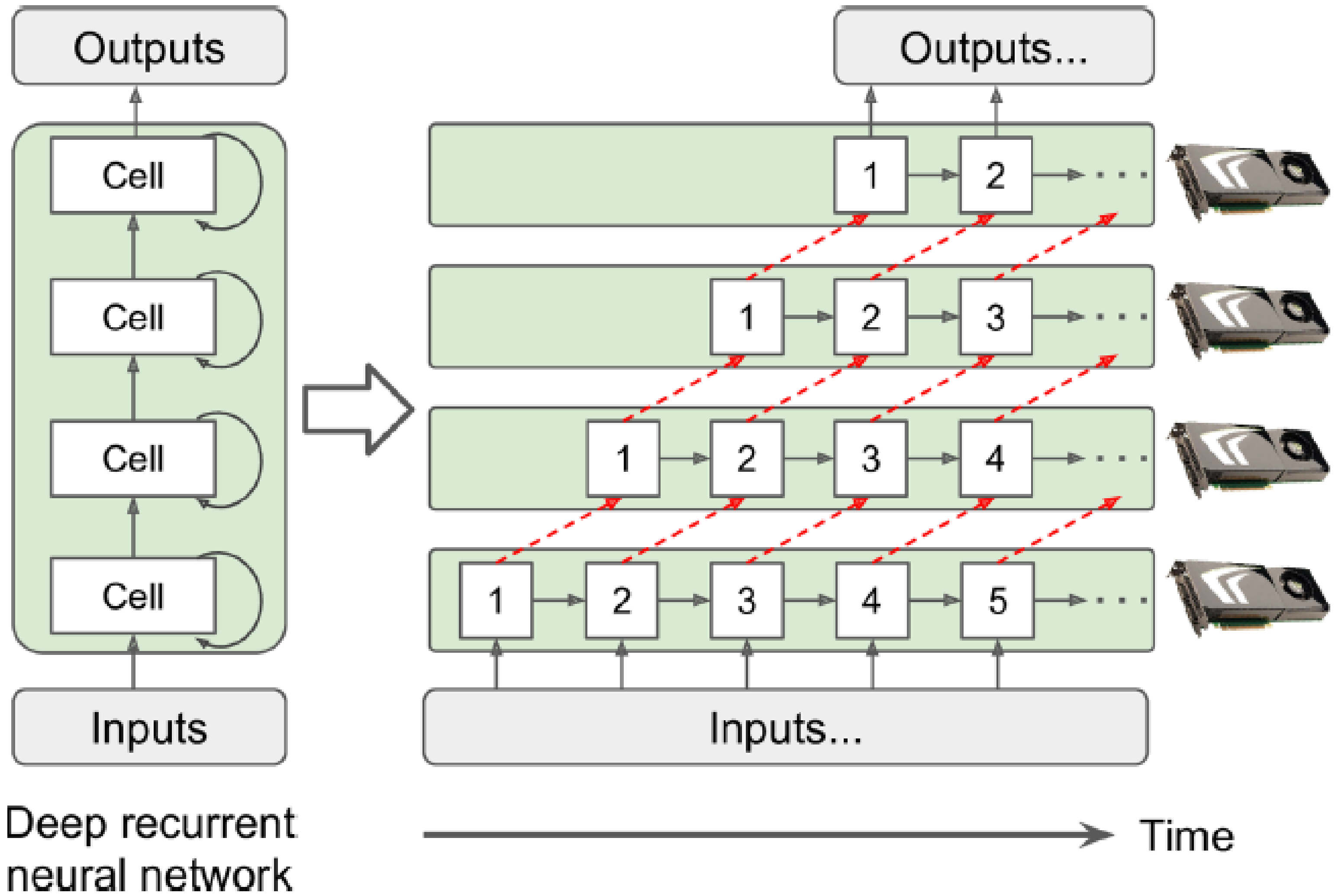
Training Models Across Multiple Devices



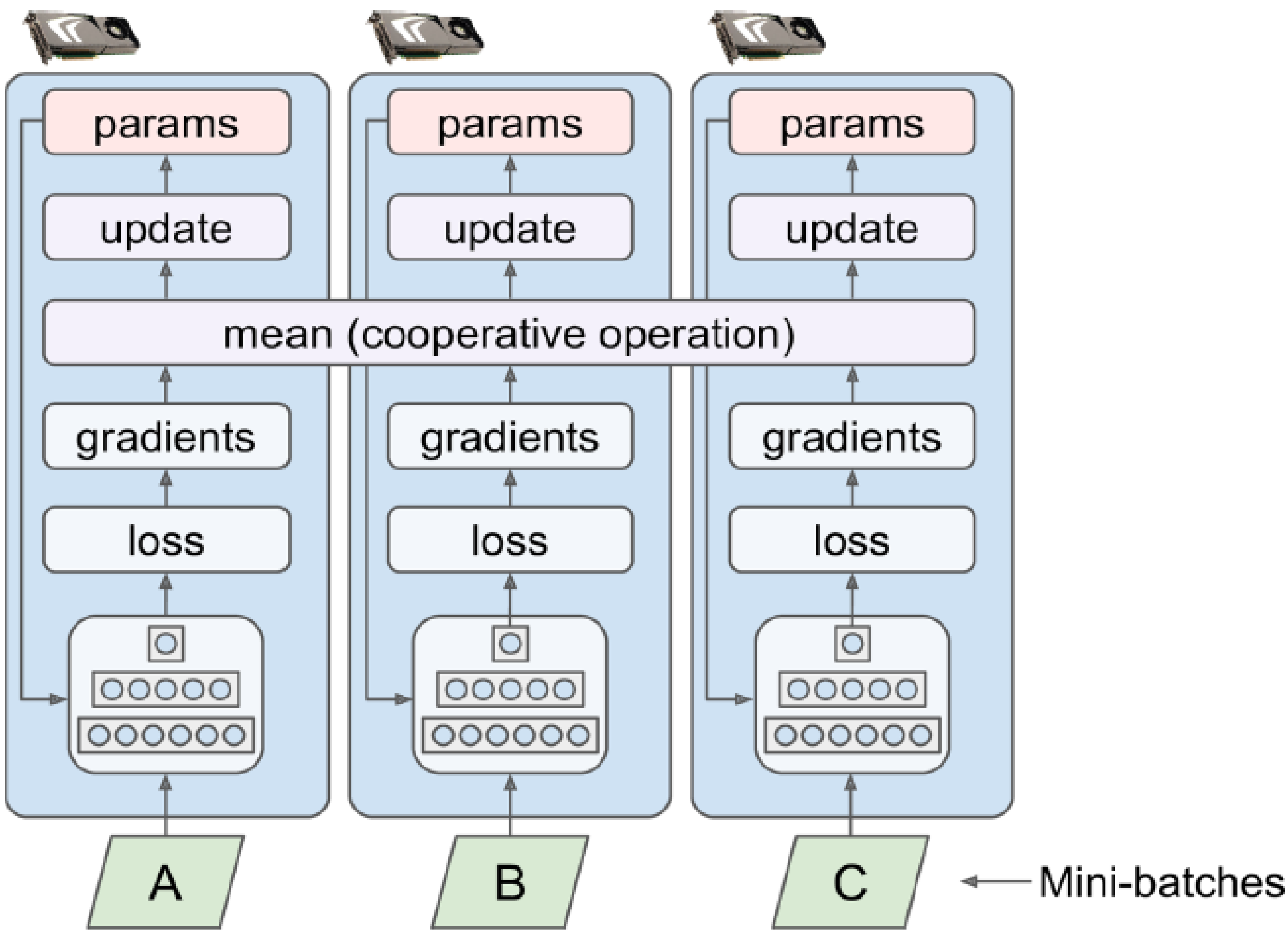
Partially connected neural network

Vertical split
Fairly good!

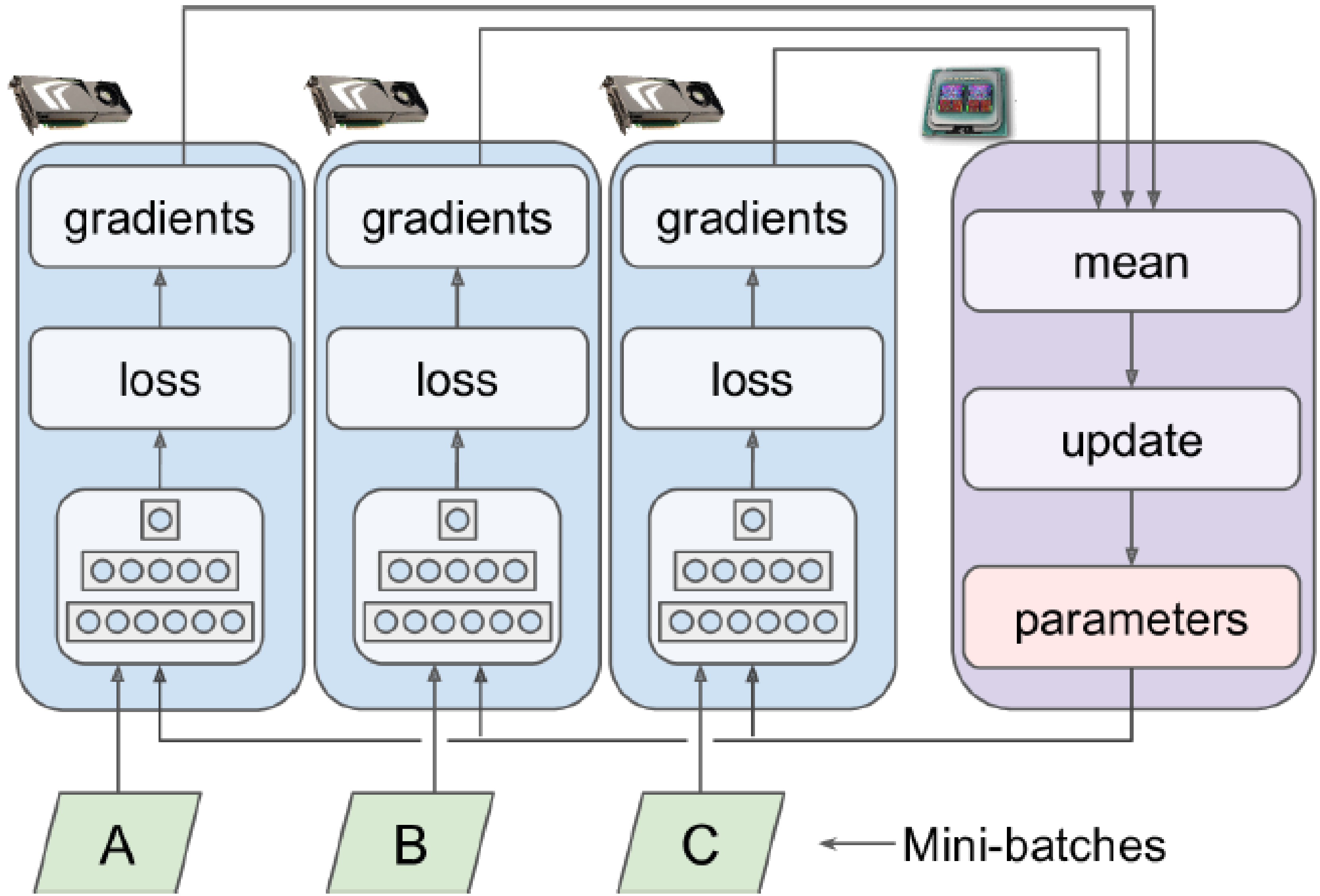
Training Models Across Multiple Devices
Model Parallelism



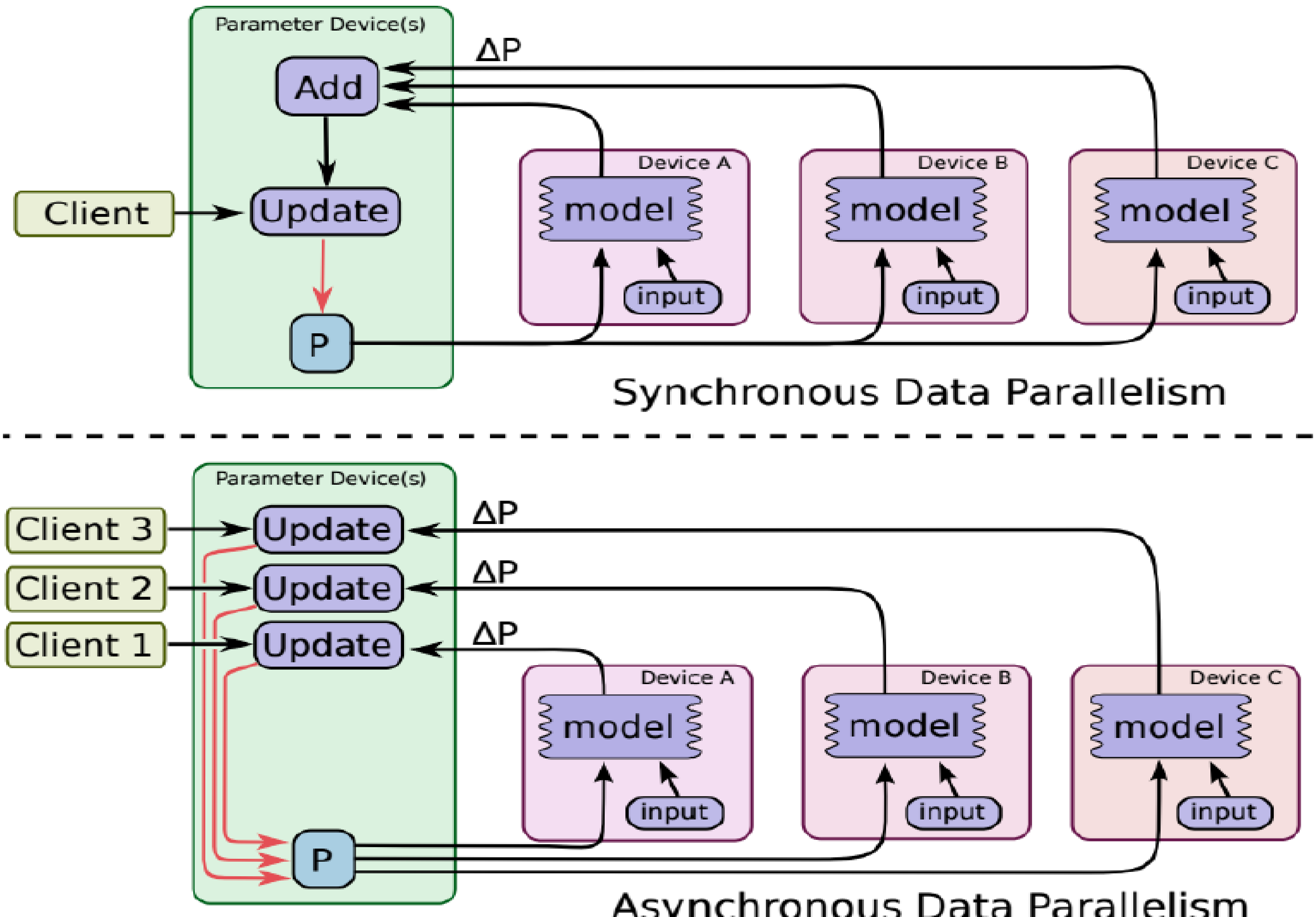
Training Models Across Multiple Devices Data parallelism -mirrored strategy



Training Models Across Multiple Devices
Data parallelism -centralized parameters



Centralized parameters



Distribution Strategies API

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = tf.keras.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

```
with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")
```

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

AllReduce mean operation

NVIDIA Collective Communications Library (NCCL)

TensorFlow Cluster

`"worker"`, `"chief"`, `"ps"` (parameter server), or `"evaluator"`

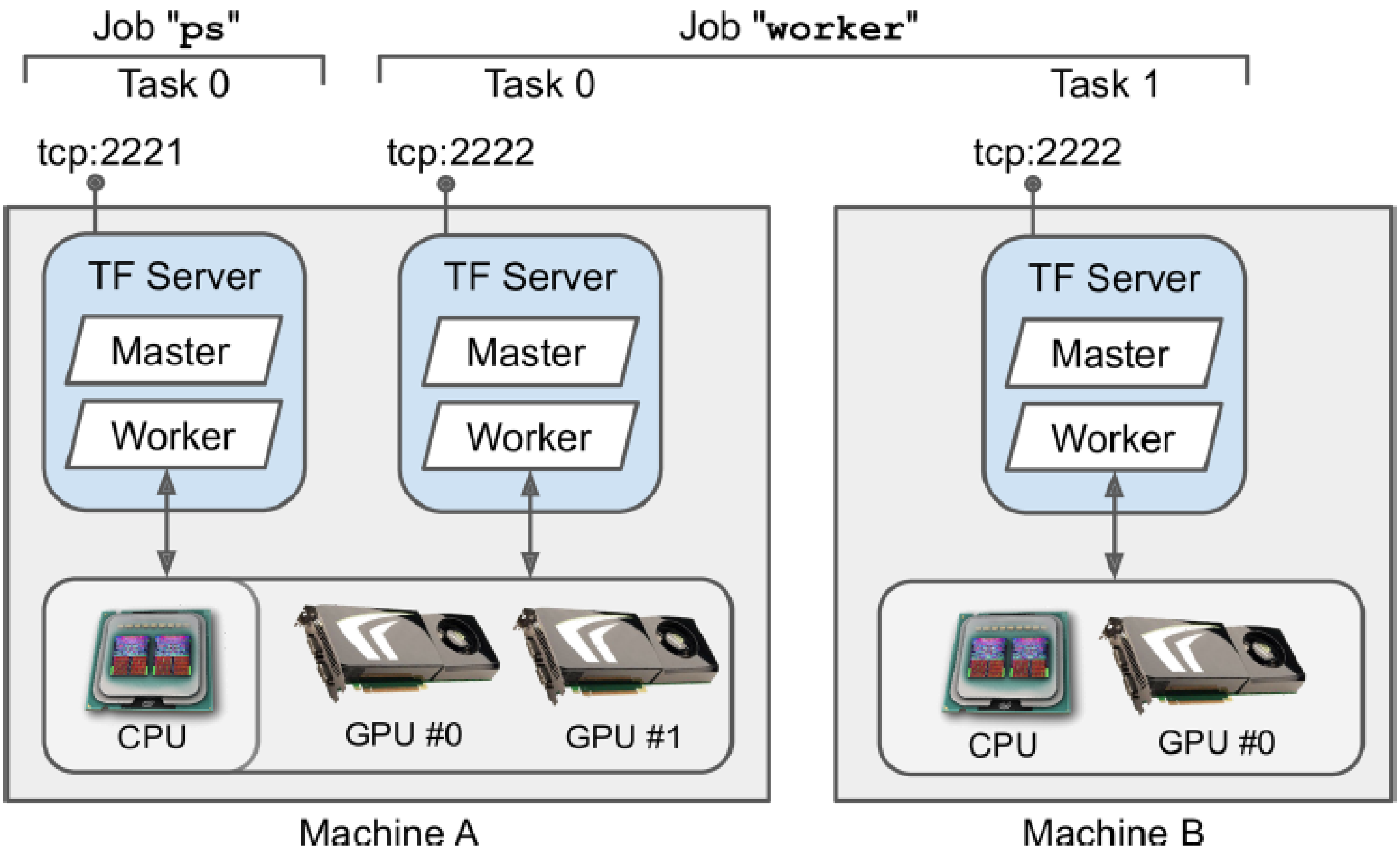
- Each *worker* performs computations, usually on a machine with one or more GPUs.

- The *chief* performs computations as well (it is a worker), but it also handles extra work such as writing TensorBoard logs or saving checkpoints. There is a single chief in a cluster. If no chief is specified, then the first worker is the chief.

- A *parameter server* only keeps track of variable values, and it is usually on a CPU-only machine. This type of task is only used with the `ParameterServerStrategy`.

- An *evaluator* obviously takes care of evaluation.

TensorFlow Cluster



cluster specification

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222"   # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
}
```

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```

Example Implementation

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = tf.keras.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

ParameterServerStrategy

TPUStrategy

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```