12 | Computational Performance

In deep learning, datasets are usually large and model computation is complex. Therefore, we are always very concerned about computing performance. This chapter will focus on the important factors that affect computing performance: imperative programming, symbolic programming, asynchronous programing, automatic parallel computation, and multi-GPU computation. By studying this chapter, you should be able to further improve the computing performance of the models that have been implemented in the previous chapters, for example, by reducing the model training time without affecting the accuracy of the model.

12.1 Compilers and Interpreters

So far, this book has focused on imperative programming, which makes use of statements such as `print`, `+` or `if` to change a program's state. Consider the following example of a simple imperative program.

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

```
10
```

Python is an interpreted language. When evaluating `fancy_func` it performs the operations making up the function's body *in sequence*. That is, it will evaluate `e = add(a, b)` and it will store the results as variable `e`, thereby changing the program's state. The next two statements `f = add(c, d)` and `g = add(e, f)` will be excecuted similarly, performing additions and storing the results as variables. Fig. 12.1.1 illustrates the flow of data.

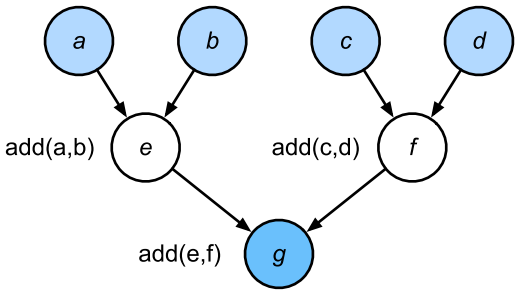


Fig. 12.1.1: Data flow in an imperative program.

Although imperative programming is convenient, it may be inefficient. On one hand, even if the `add` function is repeatedly called throughout `fancy_func`, Python will execute the three function calls individually. If these are executed, say, on a GPU (or even on multiple GPUs), the overhead arising from the Python interpreter can become overwhelming. Moreover, it will need to save the variable values of `e` and `f` until all the statements in `fancy_func` have been executed. This is because we do not know whether the variables `e` and `f` will be used by other parts of the program after the statements `e = add(a, b)` and `f = add(c, d)` have been executed.

12.1.1 Symbolic Programming

Consider the alternative, symbolic programming where computation is usually performed only once the process has been fully defined. This strategy is used by multiple deep learning frameworks, including Theano, Keras and TensorFlow (the latter two have since acquired imperative extensions). It usually involves the following steps:

1. Define the operations to be executed.
2. Compile the operations into an executable program.
3. Provide the required inputs and call the compiled program for execution.

This allows for a significant amount of optimization. First off, we can skip the Python interpreter in many cases, thus removing a performance bottleneck that can become significant on multiple fast GPUs paired with a single Python thread on a CPU. Secondly, a compiler might optimize and rewrite the above code into `print((1 + 2) + (3 + 4))` or even `print(10)`. This is possible since a compiler gets to see the full code before turning it into machine instructions. For instance, it can release memory (or never allocate it) whenever a variable is no longer needed. Or it can transform the code entirely into an equivalent piece. To get a better idea consider the following simulation of imperative programming (it's Python after all) below.

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
```

(continues on next page)

(continued from previous page)

```
    g = add(e, f)
    return g
'''

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

The differences between imperative (interpreted) programming and symbolic programming are as follows:

- Imperative programming is easier. When imperative programming is used in Python, the majority of the code is straightforward and easy to write. It is also easier to debug imperative programming code. This is because it is easier to obtain and print all relevant intermediate variable values, or use Python's built-in debugging tools.
- Symbolic programming is more efficient and easier to port. It makes it easier to optimize the code during compilation, while also having the ability to port the program into a format independent of Python. This allows the program to be run in a non-Python environment, thus avoiding any potential performance issues related to the Python interpreter.

12.1.2 Hybrid Programming

Historically most deep learning frameworks choose between an imperative or a symbolic approach. For example, Theano, TensorFlow (inspired by the latter), Keras and CNTK formulate models symbolically. Conversely Chainer and PyTorch take an imperative approach. An imperative mode was added TensorFlow 2.0 (via Eiger) and Keras in later revisions. When designing Gluon, developers considered whether it would be possible to combine the benefits of both programming models. This led to a hybrid model that lets users develop and debug using pure imperative programming, while having the ability to convert most programs into symbolic programs to be run when product-level computing performance and deployment are required.

In practice this means that we build models using either the `HybridBlock` or the `HybridSequential` and `HybridConcurrent` classes. By default, they are executed in the same way `Block` or `Sequential` and `Concurrent` classes are executed in imperative programming. `HybridSequential` is a subclass of `HybridBlock` (just like `Sequential` subclasses `Block`). When the `hybridize` function is called,

Gluon compiles the model into the form used in symbolic programming. This allows one to optimize the compute-intensive components without sacrifices in the way a model is implemented. We will illustrate the benefits below, focusing on sequential models and blocks only (the concurrent composition works analogously).

12.1.3 HybridSequential

The easiest way to get a feel for how hybridization works is to consider deep networks with multiple layers. Conventionally the Python interpreter will need to execute the code for all layers to generate an instruction that can then be forwarded to a CPU or a GPU. For a single (fast) compute device this doesn't cause any major issues. On the other hand, if we use an advanced 8-GPU server such as an AWS P3dn.24xlarge instance Python will struggle to keep all GPUs busy. The single-threaded Python interpreter becomes the bottleneck here. Let's see how we can address this for significant parts of the code by replacing `Sequential` by `HybridSequential`. We begin by defining a simple MLP.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# factory for networks
def get_net():
    net = nn.HybridSequential()
    net.add(nn.Dense(256, activation='relu'),
            nn.Dense(128, activation='relu'),
            nn.Dense(2))
    net.initialize()
    return net

x = np.random.normal(size=(1, 512))
net = get_net()
net(x)
```

```
array([[ 0.16526176, -0.14005631]])
```

By calling the `hybridize` function, we are able to compile and optimize the computation in the MLP. The model's computation result remains unchanged.

```
net.hybridize()
net(x)
```

```
array([[ 0.16526176, -0.14005631]])
```

This seems almost too good to be true: simply designate a block to be `HybridSequential`, write the same code as before and invoke `hybridize`. Once this happens the network is optimized (we will benchmark the performance below). Unfortunately this doesn't work magically for every layer. That said, the blocks provided by Gluon are by default subclasses of `HybridBlock` and thus hybridizable. A layer will not be optimized if it inherits from the `Block` instead.

Acceleration by Hybridization

To demonstrate the performance improvement gained by compilation we compare the time needed to evaluate `net(x)` before and after hybridization. Let's define a function to measure this time first. It will come handy throughout the chapter as we set out to measure (and improve) performance.

```
# Saved in the d2l package for later use
class benchmark:
    def __init__(self, description = 'Done in %.4f sec'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(self.description % self.timer.stop())
```

Now we can invoke the network twice, once with and once without hybridization.

```
net = get_net()
with benchmark('Without hybridization: %.4f sec'):
    for i in range(1000): net(x)
    npx.waitall()

net.hybridize()
with benchmark('With    hybridization: %.4f sec'):
    for i in range(1000): net(x)
    npx.waitall()
```

```
Without hybridization: 0.5402 sec
With    hybridization: 0.2650 sec
```

As is observed in the above results, after a HybridSequential instance calls the `hybridize` function, computing performance is improved through the use of symbolic programming.

Serialization

One of the benefits of compiling the models is that we can serialize (save) the model and its parameters to disk. This allows us to store a model in a manner that is independent of the front-end language of choice. This allows us to deploy trained models to other devices and easily use other front-end programming languages. At the same time the code is often faster than what can be achieved in imperative programming. Let's see the `export` method in action.

```
net.export('my_mlp')
!ls -lh my_mlp*
```

```
-rw-r--r-- 1 jenkins jenkins 643K Dec 17 22:22 my_mlp-0000.params
-rw-r--r-- 1 jenkins jenkins 3.0K Dec 17 22:22 my_mlp-symbol.json
```

The model is decomposed into a (large binary) parameter file and a JSON description of the program required to execute to compute the model. The files can be read by other front-end languages supported by Python or MXNet, such as C++, R, Scala, and Perl. Let's have a look at the model description.

```
!head my_mlp-symbol.json
```

```
{
  "nodes": [
    {
      "op": "null",
      "name": "data",
      "inputs": []
    },
    {
      "op": "null",
      "name": "dense3_weight",
```

Things are slightly more tricky when it comes to models that resemble code more closely. Basically hybridization needs to deal with control flow and Python overhead in a much more immediate manner. Moreover,

Contrary to the Block instance, which needs to use the `forward` function, for a HybridBlock instance we need to use the `hybrid_forward` function.

Earlier, we demonstrated that, after calling the `hybridize` method, the model is able to achieve superior computing performance and portability. Note, though that hybridization can affect model flexibility, in particular in terms of control flow. We will illustrate how to design more general models and also how compilation will remove spurious Python elements.

```
class HybridNet(nn.HybridBlock):
    def __init__(self, **kwargs):
        super(HybridNet, self).__init__(**kwargs)
        self.hidden = nn.Dense(4)
        self.output = nn.Dense(2)

    def hybrid_forward(self, F, x):
        print('module F: ', F)
        print('value  x: ', x)
        x = F.npx.relu(self.hidden(x))
        print('result  : ', x)
        return self.output(x)
```

The code above implements a simple network with 4 hidden units and 2 outputs. `hybrid_forward` takes an additional argument - the module `F`. This is needed since, depending on whether the code has been hybridized or not, it will use a slightly different library (`ndarray` or `symbol`) for processing. Both classes perform very similar functions and MXNet automatically determines the argument. To understand what is going on we print the arguments as part of the function invocation.

```
net = HybridNet()
net.initialize()
x = np.random.normal(size=(1, 3))
net(x)
```

```
module F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/d2l-en-1/lib/
↪python3.7/site-packages/mxnet/ndarray/__init__.py'>
value  x:  [[-0.6338663   0.40156594  0.46456942]]
result  :  [[0.01641375 0.         0.         0.        ]]
```

```
array([[0.00097611, 0.00019453]])
```

Repeating the forward computation will lead to the same output (we omit details). Now let's see what happens if we invoke the `hybridize` method.

```
net.hybridize()
net(x)
```

```
module F:  <module 'mxnet.symbol' from '/var/lib/jenkins/miniconda3/envs/d2l-en-1/lib/
↪python3.7/site-packages/mxnet/symbol/__init__.py'>
value  x:  <_Symbol data>
result  :  <_Symbol hybridnet0_relu0>
```

```
array([[0.00097611, 0.00019453]])
```

Instead of using `ndarray` we now use the `symbol` module for `F`. Moreover, even though the input is of `ndarray` type, the data flowing through the network is now converted to `symbol` type as part of the compilation process. Repeating the function call leads to a surprising outcome:

```
net(x)
```

```
array([[0.00097611, 0.00019453]])
```

This is quite different from what we saw previously. All print statements, as defined in `hybrid_forward` are omitted. Indeed, after hybridization the execution of `net(x)` does not involve the Python interpreter any longer. This means that any spurious Python code is omitted (such as print statements) in favor of a much more streamlined execution and better performance. Instead, MXNet directly calls the C++ backend. Also note that some functions are not supported in the `symbol` module (like `asnumpy`) and operations in-place like `a += b` and `a[:] = a + b` must be rewritten as `a = a + b`. Nonetheless, compilation of models is worth the effort whenever speed matters. The benefit can range from small percentage points to more than twice the speed, depending on the complexity of the model, the speed of the CPU and the speed and number of GPUs.

Summary

- Imperative programming makes it easy to design new models since it is possible to write code with control flow and the ability to use a large amount of the Python software ecosystem.
- Symbolic programming requires that we specify the program and compile it before executing it. The benefit is improved performance.
- MXNet is able to combine the advantages of both approaches as needed.

- Models constructed by the `HybridSequential` and `HybridBlock` classes are able to convert imperative programs into symbolic programs by calling the `hybridize` method.

Exercises

1. Design a network using the `HybridConcurrent` class. Alternatively look at *Networks with Parallel Concatenations (GoogLeNet)* (page 273) for a network to compose.
2. Add `x.asnumpy()` to the first line of the `hybrid_forward` function of the HybridNet class in this section. Execute the code and observe the errors you encounter. Why do they happen?
3. What happens if we add control flow, i.e. the Python statements `if` and `for` in the `hybrid_forward` function?
4. Review the models that interest you in the previous chapters and use the HybridBlock class or HybridSequential class to implement them.

12.2 Asynchronous Computation

Today's computers are highly parallel systems, consisting of multiple CPU cores (often multiple threads per core), multiple processing elements per GPU and often multiple GPUs per device. In short, we can process many different things at the same time, often on different devices. Unfortunately Python is not a great way of writing parallel and asynchronous code, at least not with some extra help. After all, Python is single-threaded and this is unlikely to change in the future. Deep learning frameworks such as MXNet and TensorFlow utilize an asynchronous programming model to improve performance (PyTorch uses Python's own scheduler leading to a different performance trade-off). Hence, understanding how asynchronous programming works helps us to develop more efficient programs, by proactively reducing computational requirements and mutual dependencies. This allows us to reduce memory overhead and increase processor utilization. We begin by importing the necessary libraries.

```
import d2l, numpy, os, subprocess
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

12.2.1 Asynchrony via Backend

For a warmup consider the following toy problem - we want to generate a random matrix and multiply it. Let's do that both in NumPy and in MXNet NP to see the difference.

```
with d2l.benchmark('numpy   : %.4f sec'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.benchmark('mxnet.np: %.4f sec'):
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
```

```
numpy   : 0.5854 sec
mxnet.np: 0.0031 sec
```

This is orders of magnitude faster. At least it seems to be so. Since both are executed on the same processor something else must be going on. Forcing MXNet to finish all computation prior to returning shows what happened previously: computation is being executed by the backend while the frontend returns control to Python.

```
with d2l.benchmark():
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
    npx.waitall()
```

```
Done in 0.6616 sec
```

Broadly speaking, MXNet has a frontend for direct interaction with the users, e.g., via Python, as well as a backend used by the system to perform the computation. The backend possesses its own threads that continuously collect and execute queued tasks. Note that for this to work the backend must be able to keep track of the dependencies between various steps in the computational graph. Hence it is ony possible to parallelize operations that do not depend on each other.

As shown in Fig. 12.2.1, users can write MXNet programs in various frontend languages, such as Python, R, Scala and C++. Regardless of the front-end programming language used, the execution of MXNet programs occurs primarily in the back-end of C++ implementations. Operations issued by the frontend language are passed on to the backend for execution. The backend manages its own threads that continuously collect and execute queued tasks. Note that for this to work the backend must be able to keep track of the dependencies between various steps in the computational graph. That is, it is not possible to parallelize operations that depend on each other.

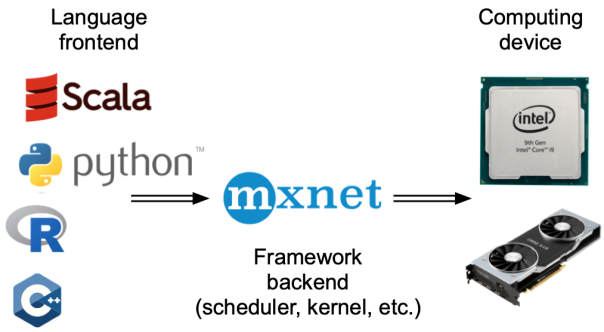


Fig. 12.2.1: Programming Frontends.

Let's look at another toy example to understand the dependency graph a bit better.

```
x = np.ones((1, 2))
y = np.ones((1, 2))
z = x * y + 2
z
```

```
array([[3., 3.]])
```

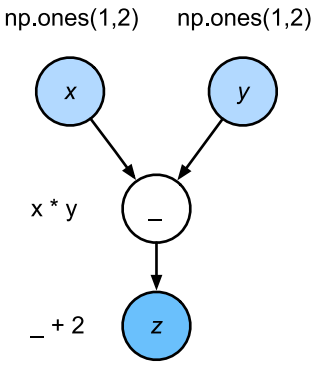


Fig. 12.2.2: Dependencies.

The code snippet above is also illustrated in Fig. 12.2.2. Whenever the Python frontend thread executes one of the first three statements, it simply returns the task to the backend queue. When the last statement's results need to be printed, the Python frontend thread will wait for the C++ backend thread to finish computing result of the variable z. One benefit of this design is that the Python frontend thread does not need to perform actual computations. Thus, there is little impact on the program's overall performance, regardless of Python's performance. Fig. 12.2.3 illustrates how frontend and backend interact.

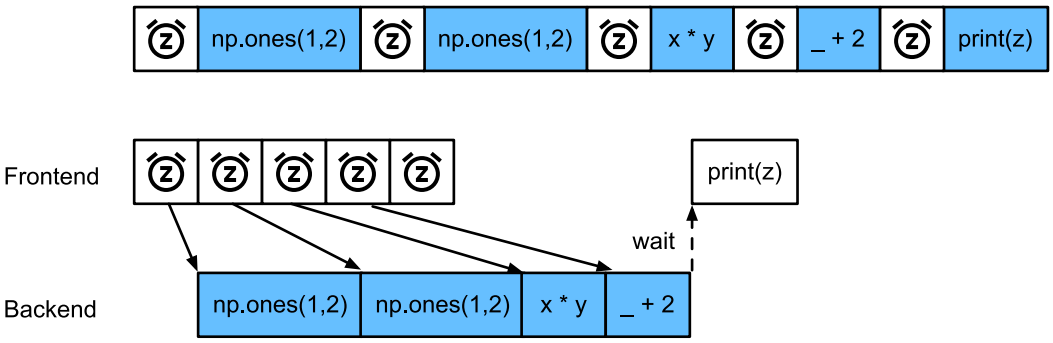


Fig. 12.2.3: Frontend and Backend.

12.2.2 Barriers and Blockers

There are a number of operations that will force Python to wait for completion: * Most obviously `npx.waitall()` waits until all computation has completed, regardless of when the compute instructions were issued. In practice it is a bad idea to use this operator unless absolutely necessary since it can lead to poor performance. * If we just want to wait until a specific variable is available we can call `z.wait_to_read()`. In this case MXNet blocks return to Python until the variable z has been computed. Other computation may well continue afterwards.

Let's see how this works in practice:

```
with d2l.benchmark('waitall     : %.4f sec'):
    b = np.dot(a, a)
    npx.waitall()

with d2l.benchmark('wait_to_read: %.4f sec'):
    b = np.dot(a, a)
    b.wait_to_read()
```

```
waitall     : 0.0105 sec
wait_to_read: 0.0042 sec
```

Both operations take approximately the same time to complete. Besides the obvious blocking operations we recommend that the reader is aware of *implicit* blockers. Printing a variable clearly requires the variable to be avaialable and is thus a blocker. Lastly, conversions to NumPy via `z.asnumpy()` and conversions to scalars via `z.item()` are blocking, since NumPy has no notion of asynchrony. It needs access to the values just like the `print` function. Copying small amounts of data frequently from MXNet's scope to NumPy and back can destroy performance of an otherwise efficient code, since each such operation requires the compute graph to evaluate all intermediate results needed to get the relevant term *before* anything else can be done.

```
with d2l.benchmark('numpy  conversion: %.4f sec'):
    b = np.dot(a, a)
    b.asnumpy()

with d2l.benchmark('scalar conversion: %.4f sec'):
    b = np.dot(a, a)
    b.sum().item()
```

```
numpy  conversion: 0.0064 sec
scalar conversion: 0.0134 sec
```

12.2.3 Improving Computation

On a heavily multithreaded system (even regular laptops have 4 threads or more and on multi-socket servers this number can exceed 256) the overhead of scheduling operations can become significant. This is why it's highly desirable to have computation and scheduling occur asynchronously and in parallel. To illustrate the benefit of doing this let's see what happens if we increment a variable by 1 multiple times, both in sequence or asynchronously. We simulate synchronous execution by inserting a `wait_to_read()` barrier in between each addition.

```
with d2l.benchmark('Synchronous : %.4f sec'):
    for _ in range(1000):
        y = x + 1
        y.wait_to_read()

with d2l.benchmark('Asynchronous: %.4f sec'):
    for _ in range(1000):
        y = x + 1
    y.wait_to_read()
```

```
Synchronous : 0.0553 sec
Asynchronous: 0.0537 sec
```

A slightly simplified interaction between the Python front-end thread and the C++ back-end thread can be summarized as follows:

1. The front-end orders the back-end to insert the calculation task `y = x + 1` into the queue.
2. The back-end then receives the computation tasks from the queue and performs the actual computations.
3. The back-end then returns the computation results to the front-end.

Assume that the durations of these three stages are t_1, t_2 and t_3, respectively. If we do not use asynchronous programming, the total time taken to perform 1000 computations is approximately $1000(t_1 + t_2 + t_3)$. If asynchronous programming is used, the total time taken to perform 1000 computations can be reduced to $t_1 + 1000t_2 + t_3$ (assuming $1000t_2 > 999t_1$), since the front-end does not have to wait for the back-end to return computation results for each loop.

12.2.4 Improving Memory Footprint

Imagine a situation where we keep on inserting operations into the backend by executing Python code on the frontend. For instance, the frontend might insert a large number of minibatch tasks within a very short time. After all, if no meaningful computation happens in Python this can be done quite quickly. If each of these tasks can be launched quickly at the same time this may cause a spike in memory usage. Given a finite amount of memory available on GPUs (and even on CPUs) this can lead to resource contention or even program crashes. Some readers might have noticed that previous training routines made use of synchronization methods such as `item` or even `asnumpy`.

We recommend to use these operations carefully, e.g., for each minibatch, such as to balance computational efficiency and memory footprint. To illustrate what happens let's implement a simple training loop for a deep network and measure its memory consumption and timing. Below is the mock data generator and deep network.

```
def data_iter():
    timer = d2l.Timer()
    num_batches, batch_size = 150, 1024
    for i in range(num_batches):
        X = np.random.normal(size=(batch_size, 512))
        y = np.ones((batch_size,))
        yield X, y
        if (i + 1) % 50 == 0:
            print('batch %d, time %.4f sec' % (i + 1, timer.stop()))

net = nn.Sequential()
net.add(nn.Dense(2048, activation='relu'),
        nn.Dense(512, activation='relu'), nn.Dense(1))
net.initialize()
trainer = gluon.Trainer(net.collect_params(), 'sgd')
loss = gluon.loss.L2Loss()
```

Next we need a tool to measure the memory footprint of our code. We use a relatively primitive ps call to accomplish this (note that the latter only works on Linux and MacOS). For a much more detailed analysis of what is going on here use e.g., Nvidia's Nsight[168] or Intel's vTune[169].

```
def get_mem():
    res = subprocess.check_output(['ps', 'u', '-p', str(os.getpid())])
    return int(str(res).split()[15]) / 1e3
```

Before we can begin testing we need to initialize the parameters of the network and process one batch. Otherwise it would be tricky to see what the additional memory consumption is. See Section 5.3 for further details related to initialization.

```
for X, y in data_iter():
    break
loss(y, net(X)).wait_to_read()
```

To ensure that we don't overflow the task buffer on the backend we insert a `wait_to_read` call for the loss function at the end of each loop. This forces the forward pass to complete before a new forward pass is commenced. Note that a (possibly more elegant) alternative would have been to track the loss in a scalar variable and to force a barrier via the `item` call.

```
mem = get_mem()
with d2l.benchmark('Time per epoch: %.4f sec'):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
        l.backward()
        trainer.step(X.shape[0])
        l.wait_to_read() # barrier before new batch
```

(continues on next page)

[168] https://developer.nvidia.com/nsight-compute-2019_5
[169] https://software.intel.com/en-us/vtune

(continued from previous page)

```
    npx.waitall()
print('increased memory: %f MB' % (get_mem() - mem))
```

```
batch 50, time 2.3893 sec
batch 100, time 4.7343 sec
batch 150, time 7.2224 sec
Time per epoch: 7.2520 sec
increased memory: 7.216000 MB
```

As we see, the timing of the minibatches lines up quite nicely with the overall runtime of the optimization code. Moreover, memory footprint only increases slightly. Now let's see what happens if we drop the barrier at the end of each minibatch.

```
mem = get_mem()
with d2l.benchmark('Time per epoch: %.4f sec'):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
        l.backward()
        trainer.step(X.shape[0])
    npx.waitall()
print('increased memory: %f MB' % (get_mem() - mem))
```

```
batch 50, time 0.1010 sec
batch 100, time 0.1986 sec
batch 150, time 0.2984 sec
Time per epoch: 7.8668 sec
increased memory: 0.020000 MB
```

Even though the time to issue instructions for the backend is an order of magnitude smaller, we still need to perform computation. Consequently a large amount of intermediate results cannot be released and may pile up in memory. While this didn't cause any issues in the toy example above, it might well have resulted in out of memory situations when left unchecked in real world scenarios.

Summary

- MXNet decouples the Python frontend from an execution backend. This allows for fast asynchronous insertion of commands into the backend and associated parallelism.
- Asynchrony leads to a rather responsive frontend. However, use caution not to overfill the task queue since it may lead to excessive memory consumption.
- It is recommended to synchronize for each minibatch to keep frontend and backend approximately synchronized.
- Be aware of the fact that conversions from MXNet's memory management to Python will force the backend to wait until the specific variable is ready. `print`, `asnumpy` and `item` all have this effect. This can be desirable but a carless use of synchronization can ruin performance.
- Chip vendors offer sophisticated performance analysis tools to obtain a much more fine-grained insight into the efficiency of deep learning.

Exercises

1. We mentioned above that using asynchronous computation can reduce the total amount of time needed to perform 1000 computations to $t_1 + 1000t_2 + t_3$. Why do we have to assume $1000t_2 > 999t_1$ here?
2. How would you need to modify the training loop if you wanted to have an overlap of one minibatch each? I.e., if you wanted to ensure that batch b_t finishes before batch b_{t+2} commences?
3. What might happen if we want to execute code on CPUs and GPUs simultaneously? Should you still insist on synchronizing after every minibatch has been issued?
4. Measure the difference between `waitall` and `wait_to_read`. Hint - perform a number of instructions and synchronize for an intermediate result.

12.3 Automatic Parallelism

MXNet automatically constructs computational graphs at the backend. Using a computational graph, the system is aware of all the dependencies, and can selectively execute multiple non-interdependent tasks in parallel to improve speed. For instance, Fig. 12.2.2 in Section 12.2 initializes two variables independently. Consequently the system can choose to execute them in parallel.

Typically, a single operator will use all the computational resources on all CPUs or on a single GPU. For example, the `dot` operator will use all cores (and threads) on all CPUs, even if there are multiple CPU processors on a single machine. The same applies to a single GPU. Hence parallelization isn't quite so useful single-device computers. With multiple devices things matter more. While parallelization is typically most relevant between multiple GPUs, adding the local CPU will increase performance slightly. See e.g., (Hadjis et al., 2016) for a paper that focuses on training computer vision models combining a GPU and a CPU. With the convenience of an automatically parallelizing framework we can accomplish the same goal in a few lines of Python code. More broadly, our discussion of automatic parallel computation focuses on parallel computation using both CPUs and GPUs, as well as the parallelization of computation and communication. We begin by importing the required packages and modules. Note that we need at least one GPU to run the experiments in this section.

```
import d2l
from mxnet import np, npx
npx.set_np()
```

12.3.1 Parallel Computation on CPUs and GPUs

Let's start by defining a reference workload to test - the `run` function below performs 10 matrix-matrix multiplications on the device of our choosing using data allocated into two variables, `x_cpu` and `x_gpu`.

```
def run(x):
    return [x.dot(x) for _ in range(10)]

x_cpu = np.random.uniform(size=(2000, 2000))
x_gpu = np.random.uniform(size=(6000, 6000), ctx=d2l.try_gpu())
```

Now we apply the function to the data. To ensure that caching doesn't play a role in the results we warm up the devices by performing a single pass on each of them prior to measuring.

```
run(x_cpu)  # Warm-up both devices
run(x_gpu)
npx.waitall()

with d2l.benchmark('CPU time: %.4f sec'):
    run(x_cpu)
    npx.waitall()

with d2l.benchmark('GPU time: %.4f sec'):
    run(x_gpu)
    npx.waitall()
```

```
CPU time: 0.3072 sec
GPU time: 0.3030 sec
```

If we remove the `waitall()` between both tasks the system is free to parallelize computation on both devices automatically.

```
with d2l.benchmark('CPU&GPU : %.4f sec'):
    run(x_cpu)
    run(x_gpu)
    npx.waitall()
```

```
CPU&GPU : 0.3074 sec
```

In the above case the total execution time is less than the sum of its parts, since MXNet automatically schedules computation on both CPU and GPU devices without the need for sophisticated code on behalf of the user.

12.3.2 Parallel Computation and Communication

In many cases we need to move data between different devices, say between CPU and GPU, or between different GPUs. This occurs e.g., when we want to perform distributed optimization where we need to aggregate the gradients over multiple accelerator cards. Let's simulate this by computing on the GPU and then copying the results back to the CPU.

```
def copy_to_cpu(x):
    return [y.copyto(npx.cpu()) for y in x]

with d2l.benchmark('Run  on GPU: %.4f sec'):
    y = run(x_gpu)
    npx.waitall()

with d2l.benchmark('Copy to CPU: %.4f sec'):
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

```
Run  on GPU: 0.3125 sec
Copy to CPU: 1.0149 sec
```

This is somewhat inefficient. Note that we could already start copying parts of `y` to the CPU while the remainder of the list is still being computed. This situatio occurs, e.g., when we compute the (backprop) gradient on a minibatch. The gradients of some of the parameters will be available earlier than that of others. Hence it works to our advantage to start using PCI-Express bus bandwidth while the GPU is still running. Removing `waitall` between both parts allows us to simulate this scenario.

```
with d2l.benchmark('Run on GPU and copy to CPU: %.4f sec'):
    y = run(x_gpu)
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

```
Run on GPU and copy to CPU: 1.0901 sec
```

The total time required for both operations is (as expected) significantly less than the sum of their parts. Note that this task is different from parallel computation as it uses a different resource: the bus between CPU and GPUs. In fact, we could compute on both devices and communicate, all at the same time. As noted above, there is a dependency between computation and communication: `y[i]` must be computed before it can be copied to the CPU. Fortunately, the system can copy `y[i-1]` while computing `y[i]` to reduce the total running time.

We conclude with an illustration of the computational graph and its dependencies for a simple two-layer MLP when training on a CPU and two GPUs, as depicted in Fig. 12.3.1. It would be quite painful to schedule the parallel program resulting from this manually. This is where it is advantageous to have a graph based compute backend for optimization.

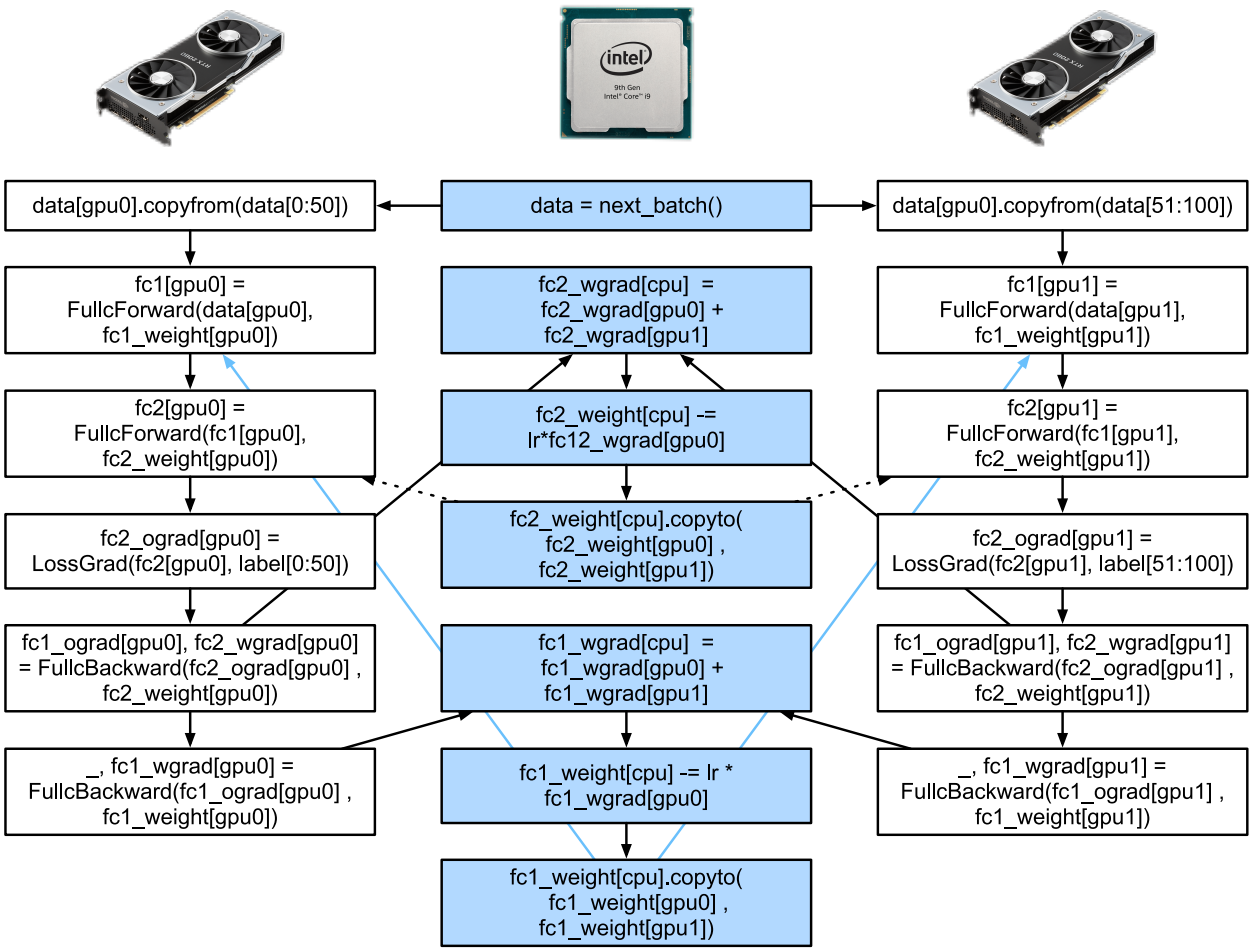


Fig. 12.3.1: Two layer MLP on a CPU and 2 GPUs.

Summary

- Modern systems have a variety of devices, such as multiple GPUs and CPUs. They can be used in parallel, asynchronously.
- Modern systems also have a variety of resources for communication, such as PCI Express, storage (typically SSD or via network), and network bandwidth. They can be used in parallel for peak efficiency.
- The backend can improve performance through through automatic parallel computation and communication.

Exercises

1. 10 operations were performed in the `run` function defined in this section. There are no dependencies between them. Design an experiment to see if MXNet will automatically execute them in parallel.
2. When the workload of an individual operator is sufficiently small, parallelization can help even on a single CPU or GPU. Design an experiment to verify this.
3. Design an experiment that uses parallel computation on CPU, GPU and communication between both devices.
4. Use a debugger such as NVIDIA's Nsight to verify that your code is efficient.
5. Designing computation tasks that include more complex data dependencies, and run experiments to see if you can obtain the correct results while improving performance.

12.4 Hardware

Building systems with great performance requires a good understanding of the algorithms and models to capture the statistical aspects of the problem. At the same time it is also indispensable to have at least a modicum of knowledge of the underlying hardware. The current section is no substitute for a proper course on hardware and systems design. Instead, it might serve as a starting point for understanding why some algorithms are more efficient than others and how to achieve good throughput. Good design can easily make a difference of an order of magnitude and, in turn, this can make the difference between being able to train a network (e.g., in a week) or not at all (in 3 months, thus missing the deadline). We'll start by looking at computers. Then we will zoom in to look more carefully at CPUs and GPUs. Lastly we zoom out to review how multiple computers are connected in a server center or in the cloud. This is not a GPU purchase guide. For this review Section 18.5. An introduction to cloud computing with AWS can be found in Section 18.3.

Impatient readers may be able to get by with Fig. 12.4.1. It is taken from Colin Scott's interactive post[172] which gives a good overview of the progress over the past decade. The original numbers are due to Jeff Dean's Stanford talk from 2010[173]. The discussion below explains some of the rationale for these numbers and how they can guide us in designing algorithms. The discussion below is very high level and cursory. It is clearly *no substitute* for a proper course but rather just meant to provide enough information for a statistical modeler to make suitable design decisions. For an in-depth overview of computer architecture we refer the reader to (Hennessy & Patterson, 2011) or a recent course on the subject, such as the one by Arste Asanovic[174].

[172] https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html
[173] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf
[174] http://inst.eecs.berkeley.edu/~cs152/sp19/

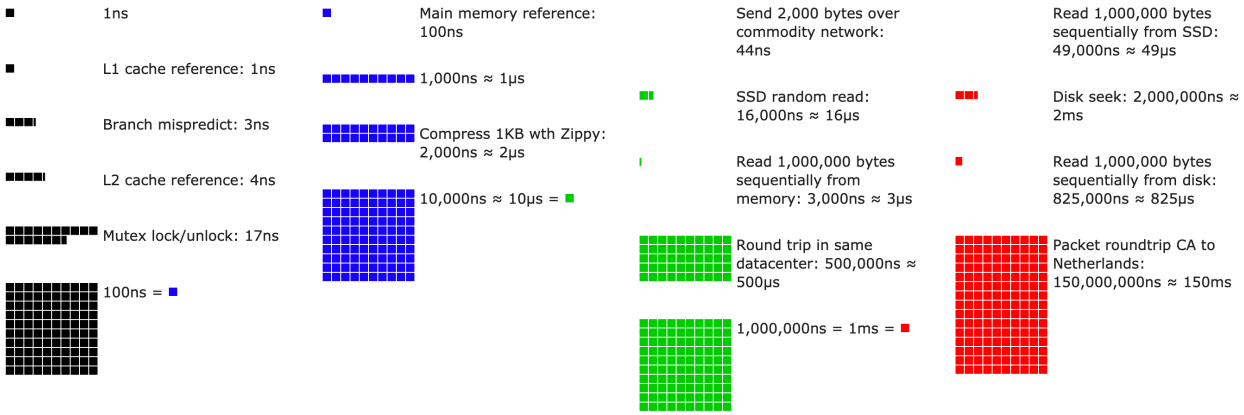


Fig. 12.4.1: Latency Numbers every Programmer should know.

12.4.1 Computers

Most deep learning researchers have access to a computer with a fair amount of memory, compute, some form of an accelerator such as a GPU, or multiples thereof. It consists of several key components:

- A processor, also referred to as CPU which is able to execute the programs we give it (in addition to running an operating system and many other things), typically consisting of 8 or more cores.
- Memory (RAM) to store and retrieve the results from computation, such as weight vectors, activations, often training data.
- An Ethernet network connection (sometimes multiple) with speeds ranging from 1Gbit/s to 100Gbit/s (on high end servers more advanced interconnects can be found).
- A high speed expansion bus (PCIe) to connect the system to one or more GPUs. Severs have up to 8 accelerators, often connected in an advanced topology, desktop systems have 1-2, depending on the budget of the user and the size of the power supply.
- Durable storage, such as a magnetic harddrive (HDD), solid state (SSD), in many cases connected using the PCIe bus, provides efficient transfer of training data to the system and storage of intermediate checkpoints as needed.

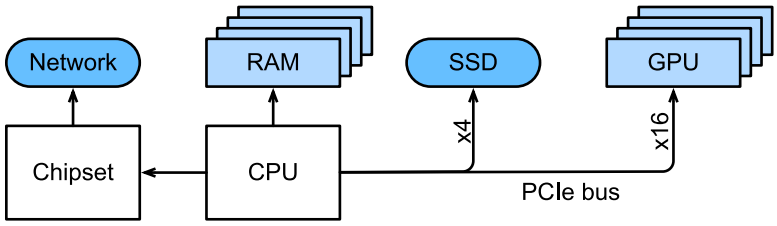


Fig. 12.4.2: Connectivity of components

As Fig. 12.4.2 indicates, most components (network, GPU, storage) are connected to the CPU across the PCI Express bus. It consists of multiple lanes that are directly attached to the CPU. For instance AMD's Threadripper 3 has 64 PCIe 4.0 lanes, each of which is capable 16 Gbit/s data transfer in both directions. The memory is directly attached to the CPU with a total bandwidth of up to 100 GB/s.

When we run code on a computer we need to shuffle data to the processors (CPU or GPU), perform computation and then move the results off the processor back to RAM and durable storage. Hence, in order to get good performance we need to make sure that this works seamlessly without any one of the systems becoming a major bottleneck. For instance, if we cannot load images quickly enough the processor won't have any work to do. Likewise, if we cannot move matrices quickly enough to the CPU (or GPU), its processing elements will starve. Finally, if we want to synchronize multiple computers across the network, the latter shouldn't slow down computation. One option is to interleave communication and computation. Let's have a look at the various components in more detail.

12.4.2 Memory

At its most basic memory is used to store data that needs to be readily accessible. At present CPU RAM is typically of the DDR4[175] variety, offering 20-25GB/s bandwidth per module. Each module has a 64 bit wide bus. Typically pairs of memory modules are used to allow for multiple channels. CPUs have between 2 and 4 memory channels, i.e., they have between 40GB/s and 100GB/s peak memory bandwidth. Often there are two banks per channel. For instance AMD's Zen 3 Threadripper has 8 slots.

While these numbers are impressive, indeed, they only tell part of the story. When we want to read a portion from memory we first need to tell the memory module where the information can be found. That is, we first need to send the *address* to RAM. Once this accomplished we can choose to read just a single 64bit record or a long sequence of records. The latter is called *burst read*. In a nutshell, sending an address to memory and setting up the transfer takes approximately 100ns (details depend on the specific timing coefficients of the memory chips used), every subsequent transfer takes only 0.2ns. In short, the first read is 500 times as expensive as subsequent ones! We could perform up to $10,000,000$ random reads per second. This suggests that we avoid random memory access as far as possible and use burst reads (and writes) instead.

Matters are a bit more complex when we take into account that we have multiple banks. Each bank can read memory largely independently. This means two things: the effective number of random reads is up to 4x higher, provided that they are spread evenly across memory. It also means that it's still a bad idea to perform random reads since burst reads are 4x faster, too. Secondly, due to memory alignment to 64 bit boundaries it is a good idea to align any datastructures with the same boundaries. Compilers do this pretty much automatically[176] when the appropriate flags are set. Curious readers are encouraged to review a lecture on DRAMs such as the one by Zeshan Chishti[177].

GPU memory is subject to even higher bandwidth requirements since they have many more processing elements than CPUs. By and large there are two options to address them. One is to make the memory bus significantly wider. For instance NVIDIA's RTX 2080 Ti has a 352 bit wide bus. This allows for much more information to be transferred at the same time. Secondly, GPUs use specific high-performance memory. Consumer grade devices, such as NVIDIA's RTX and Titan series typically use GDDR6[178] chips with over 500 GB/s aggregate bandwidth. An alternative is to use HBM (high bandwidth memory) modules. They use a very different interface and connect directly with GPUs on a dedicated silicon wafer. This makes them very expensive and their use is typically limited to high end server chips, such as the NVIDIA Volta V100 series of accelerators. Quite unsurprisingly GPU memory is *much* smaller than CPU memory due to its higher cost. For

[175] https://en.wikipedia.org/wiki/DDR4_SDRAM
[176] https://en.wikipedia.org/wiki/Data_structure_alignment
[177] http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf
[178] https://en.wikipedia.org/wiki/GDDR6_SDRAM

our purposes, by and large their performance characteristics are similar, just a lot faster. We can safely ignore the details for the purpose of this book. They only matter when tuning GPU kernels for high throughput.

12.4.3 Storage

We saw that some of the key characteristics of RAM were *bandwidth* and *latency*. The same is true for storage devices, just that the differences can be even more extreme.

Hard Disks have been in use for over half a century. In a nutshell they contain a number of spinning platters with heads that can be positioned to read / write at any given track. High end end disks hold up to 16TB on 9 platters. One of the key benefits of HDDs is that they are relatively inexpensive. One of their many downsides are their typically catastrophic failure modes and their relatively high read latency.

To understand the latter, consider the fact that HDDs spin at around 7,200 RPM. If they were much faster they would shatter due to the centrifugal force exerted on the platters. This has a major downside when it comes to accessing a specific sector on the disk: we need to wait until the platter has rotated in position (we can move the heads but not accelerate the actual disks). Hence it can take over 8ms until the requested data is available. A common way this is expressed is to say that HDDs can operate at approximately 100 IOPs. This number has essentially remained unchanged for the past two decades. Worse still, it is equally difficult to increase bandwidth (it is in the order of 100-200 MB/s). After all, each head reads a track of bits, hence the bit rate only scales with the square root of the information density. As a result HDDs are quickly becoming relegated to archival storage and low-grade storage for very large datasets.

Solid State Drives use Flash memory to store information persistently. This allows for *much faster* access to stored records. Modern SSDs can operate at 100,000 to 500,000 IOPs, i.e., up to 3 orders of magnitude faster than HDDs. Furthermore, their bandwidth can reach 1-3GB/s, i.e., one order of magnitude faster than HDDs. These improvements sound almost too good to be true. Indeed, they come with a number of caveats, due to the way SSDs are designed.

- SSDs store information in blocks (256 KB or larger). They can only be written as a whole, which takes significant time. Consequently bit-wise random writes on SSD have very poor performance. Likewise, writing data in general takes significant time since the block has to be read, erased and then rewritten with new information. By now SSD controllers and firmware have developed algorithms to mitigate this. Nonetheless writes can be much slower, in particular for QLC (quad level cell) SSDs. The key for improved performance is to maintain a *queue* of operations, to prefer reads and to write in large blocks if possible.
- The memory cells in SSDs wear out relatively quickly (often already after a few thousand writes). Wear-level protection algorithms are able to spread the degradation over many cells. That said, it is not recommended to use SSDs for swap files or for large aggregations of log-files.
- Lastly, the massive increase in bandwidth has forced computer designers to attach SSDs directly to the PCIe bus. The drives capable of handling this, referred to as NVMe (Non Volatile Memory enhanced), can use up to 4 PCIe lanes. This amounts to up to 8GB/s on PCIe 4.0.

Cloud Storage provides a configurable range of performance. That is, the assignment of storage to virtual machines is dynamic, both in terms of quantity and in terms speed, as chosen by the user. We recommend that the user increase the provisioned number of IOPs whenever latency is too high, e.g., during training with many small records.

12.4.4 CPUs

Central Processing Units (CPUs) are the centerpiece of any computer (as before we give a very high level description focusing primarily on what matters for efficient deep learning models). They consist of a number of key components: processor cores which are able to execute machine code, a bus connecting them (the specific topology differs significantly between processor models, generations and vendors), and caches to allow for higher bandwidth and lower latency memory access than what is possible by reads from main memory. Lastly, almost all modern CPUs contain vector processing units to aid with high performance linear algebra and convolutions, as they are common in media processing and machine learning.

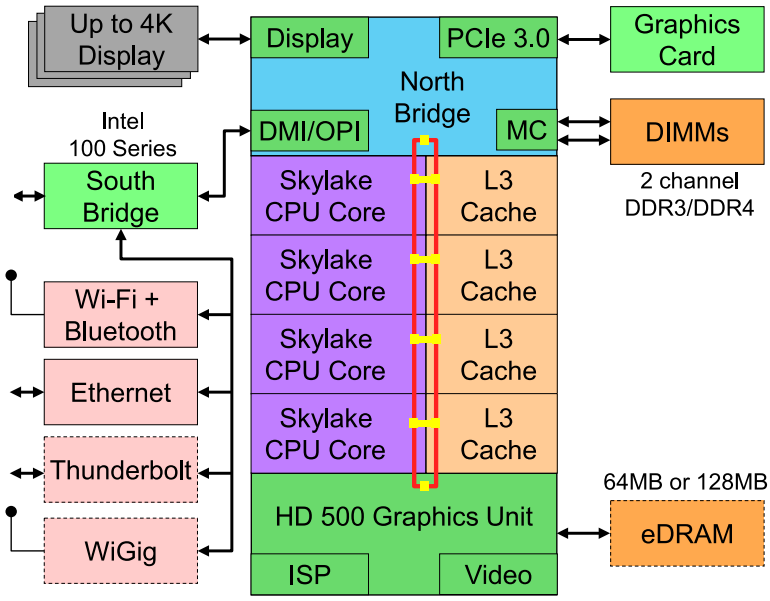


Fig. 12.4.3: Intel Skylake consumer quad-core CPU

Fig. 12.4.3 depicts an Intel Skylake consumer grade quad-core CPU. It has an integrated GPU, caches, and a ringbus connecting the four cores. Peripherals (Ethernet, WiFi, Bluetooth, SSD controller, USB, etc.) are either part of the chipset or directly attached (PCIe) to the CPU.

Microarchitecture

Each of the processor cores consists of a rather sophisticated set of components. While details differ between generations and vendors, the basic functionality is pretty much standard. The front end loads instructions and tries to predict which path will be taken (e.g., for control flow). Instructions are then decoded from assembly code to microinstructions. Assembly code is often not the lowest level code that a processor executes. Instead, complex instructions may be decoded into a set of more lower level operations. These are then processed by the actual execution core. Often the latter is capable of performing many operations simultaneously. For instance, the ARM Cortex A77 core of Fig. 12.4.4 is able to perform up to 8 operations simultaneously.

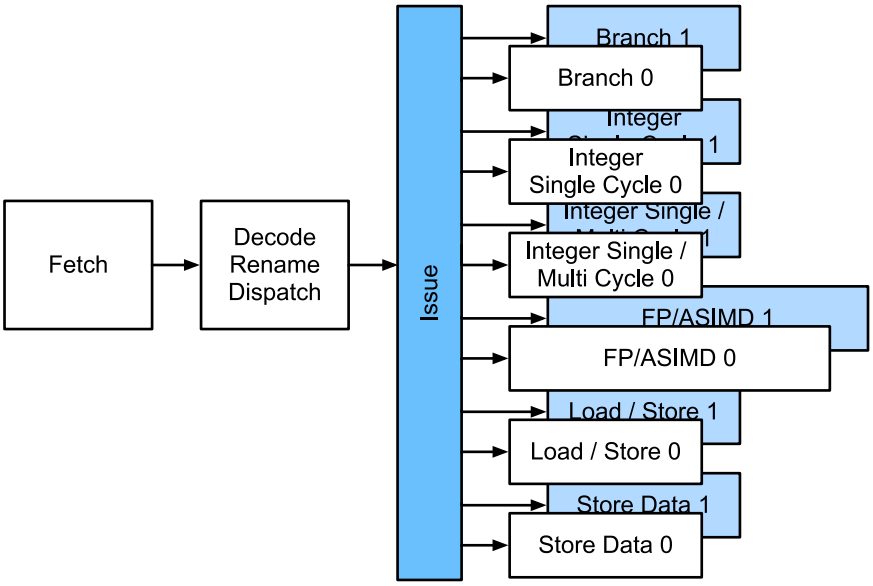


Fig. 12.4.4: ARM Cortex A77 Microarchitecture Overview

This means that efficient programs might be able to perform more than one instruction per clock cycle, *provided that* they can be carried out independently. Not all units are created equal. Some specialize in integer instructions whereas others are optimized for floating point performcne. To increase throughput the processor might also follow multiple codepaths simultaneously in a branching instruction and then discard the results of the branch not taken. This is why branch prediction units matter (on the frontend) such that only the most promising paths are pursued.

Vectorization

Deep learning is extremely compute hungry. Hence, to make CPUs suitable for machine learning one needs to perform many operations in one clock cycle. This is achieved via vector units. They have different names: on ARM they're called NEON, on x86 the latest generation is referred to as AVX2[179] units. A common aspect is that they are able to perform SIMD (single instruction multiple data) operations. Fig. 12.4.5 shows how 8 short integers can be added in one clock cycle on ARM.

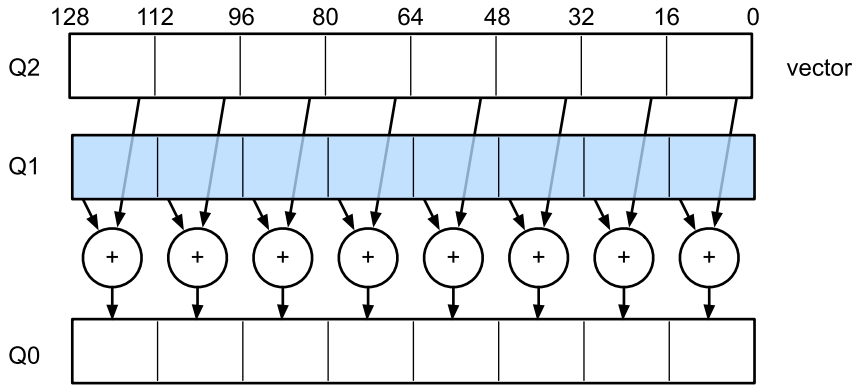


Fig. 12.4.5: 128 bit NEON vectorization

[179] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

Depending on architecture choices such registers are up to 512 bit long, allowing for the combination of up to 64 pairs of numbers. For instance, we might be multiplying two numbers and adding them to a third, which is also known as a fused multiply-add. Intel's OpenVino[180] uses these to achieve respectable throughput for deep learning on server grade CPUs. Note, though, that this number is entirely dwarved by what GPUs are capable of achieving. For instance, NVIDIA's RTX 2080 Ti has 4,352 CUDA cores, each of which is capable of processing such an operation at any time.

Cache

Consider the following situation: we have a modest CPU core with 4 cores as depicted in Fig. 12.4.3 above, running at 2GHz frequency. Moreover, let's assume that we have an IPC (instructions per clock) count of 1 and that the units have AVX2 with 256bit width enabled. Let's furthermore assume that at least one of the registers used for AVX2 operations needs to be retrieved from memory. This means that the CPU consumes 4x256bit = 1kbit of data per clock cycle. Unless we are able to transfer $2 \cdot 10^9 \cdot 128 = 256 \cdot 10^9$ bytes to the processor per second the processing elements are going to starve. Unfortunately the memory interface of such a chip only supports 20-40 GB/s data transfer, i.e., one order of magnitude less. The fix is to avoid loading *new* data from memory as far as possible and rather to cache it locally on the CPU. This is where caches come in handy (see this Wikipedia article[181] for a primer). Commonly the following names / concepts are used:

- **Registers** are strictly speaking not part of the cache. They help stage instructions. That said, CPU registers are memory locations that a CPU can access at clock speed without any delay penalty. CPUs have tens of registers. It is up to the compiler (or programmer) to use registers efficiently. For instance the C programming language has a `register` keyword.
- **L1** caches are the first line of defense against high memory bandwidth requirements. L1 caches are tiny (typical sizes might be 32-64kB) and often split into data and instructions caches. When data is found in the L1 cache access is very fast. If it cannot be found there, the search progresses down the cache hierarchy.
- **L2** caches are the next stop. Depending on architecture design and processor size they might be exclusive. They might be accessible only by a given core or shared between multiple cores. L2 caches are larger (typically 256-512kB per core) and slower than L1. Furthermore, to access something in L2 we first need to check to realize that the data isn't in L1, which adds a small amount of extra latency.
- **L3** caches are shared between multiple cores and can be quite large. AMD's Epyc 3 server CPUs have a whopping 256MB of cache spread across multiple chiplets. More typical numbers are in the 4-8MB range.

Predicting which memory elements will be needed next is one of the key optimization parameters in chip design. For instance, it is advisable to traverse memory in a *forward* direction since most caching algorithms will try to *read ahead* rather than backwards. Likewise, keeping memory access patterns local is agood way of improving performance. Adding caches is a double-edge sword. On one hand they ensure that the processor cores don't starve of data. At the same time they increase chip size, using up area that otherwise could have been spent on increasing processing power. Moreover, *cache misses* can be expensive. Consider the worst case scenario, depicted in Fig. 12.4.6. A memory location is cached on processor 0 when a thread on processor 1 requests the data. To obtain it, processor 0 needs to stop what it's doing, write the information back to main

[180] https://01.org/openvinotoolkit
[181] https://en.wikipedia.org/wiki/Cache_hierarchy

memory and then let processor 1 read it from memoruy. During this operation both processors wait. Quite potentially such code runs *more slowly* on multiple processors when compared to an efficient single-processor implementation. This is one more reason for why there is a practical limit to cache sizes (besides their physical size).

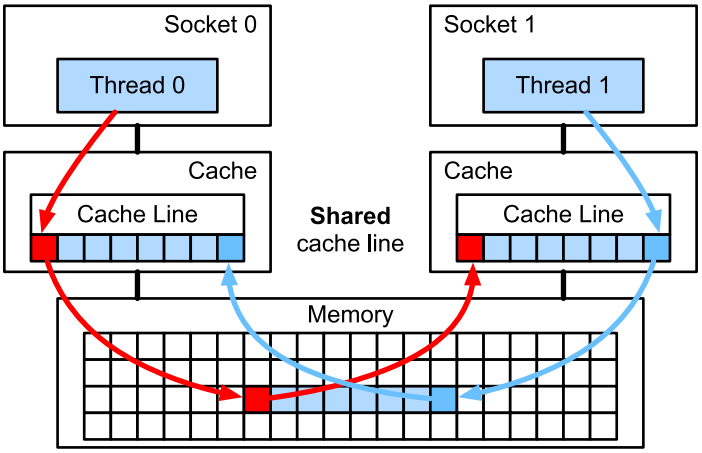


Fig. 12.4.6: False sharing (image courtesy of Intel)

12.4.5 GPUs and other Accelerators

It is not an exaggeration to claim that deep learning would not have been successful without GPUs. By the same token, it is quite reasonable to argue that GPU manufacturers' fortunes have been increased significantly due to deep learning. This co-evolution of hardware and algorithms has led to a situation where for better or worse deep learning is the preferable statistical modeling paradigm. Hence it pays to understand the specific benefits that GPUs and related accelerators such as the TPU (Jouppi et al., 2017) offer.

Of note is a distinction that is often made in practice: accelerators are optimized either for training or inference. For the latter we only need to compute the forward pass in a network. No storage of intermediate data is needed for backpropagation. Moreover, we may not need very precise computation (FP16 or INT8 typically suffice). On the other hand, during training all intermediate results need storing to compute gradients. Moreover, accumulating gradients requires higher precision to avoid numerical underflow (or overflow). This means that FP16 (or mixed precision with FP32) is the minimum required. All of this necessitates faster and larger memory (HBM2 vs. GDDR6) and more processing power. For instance, NVIDIA's Turing[182] T4 GPUs are optimized for inference whereas the V100 GPUs are preferable for training.

Recall Fig. 12.4.5. Adding vector units to a processor core allowed us to increase throughput significantly (in the example in the figure we were able to perform 16 operations simultaneously). What if we added operations that optimized not just operations between vectors but also between matrices? This strategy led to Tensor Cores (more on this shortly). Secondly, what if we added many more cores? In a nutshell, these two strategies summarize the design decisions in GPUs. Fig. 12.4.7 gives an overview over a basic processing block. It contains 16 integer and 16 floating point units. In addition to that, two Tensor Cores accelerate a narrow subset of additional operations relevant for deep learning. Each Streaming Multiprocessor (SM) consists of four such blocks.

[182] https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/

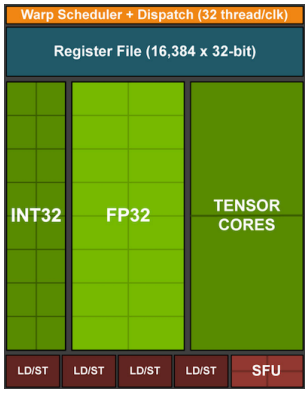


Fig. 12.4.7: NVIDIA Turing Processing Block (image courtesy of NVIDIA)

12 streaming multiprocessors are then grouped into graphics processing clusters which make up the high-end TU102 processors. Ample memory channels and an L2 cache complement the setup. Fig. 12.4.8 has the relevant details. One of the reasons for designing such a device is that individual blocks can be added or removed as needed to allow for more compact chips and to deal with yield issues (faulty modules might not be activated). Fortunately programming such devices is well hidden from the casual deep learning researcher beneath layers of CUDA and framework code. In particular, more than one of the programs might well be executed simultaneously on the GPU, provided that there are available resources. Nonetheless it pays to be aware of the limitations of the devices to avoid picking models that do not fit into device memory.

Fig. 12.4.8: NVIDIA Turing Architecture (image courtesy of NVIDIA)

A last aspect that is worth mentioning in more detail are TensorCores. They are an example of a recent trend of adding more optimized circuits that are specifically effective for deep learning. For instance, the TPU added a systolic array (Kung, 1988) for fast matrix multiplication. There the design was to support a very small number (one for the first generation of TPUs) of large operations. TensorCores are at the other end. They are optimized for small operations involving between 4x4 and 16x16 matrices, depending on their numerical precision. Fig. 12.4.9 gives an overview of the optimizations.

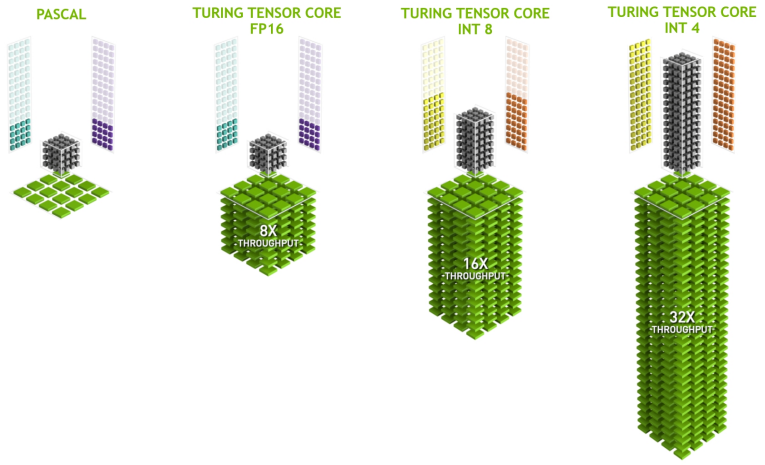


Fig. 12.4.9: NVIDIA TensorCores in Turing (image courtesy of NVIDIA)

Obviously when optimizing for computation we end up making certain compomises. One of them is that GPUs are not very good at handling interrupts and sparse data. While there are notable exceptions, such as Gunrock[183] (Wang et al., 2016), the access pattern of sparse matrices and vectors do not go well with the high bandwidth burst read operations where GPUs excel. Matching both goals is an area of active research. See e.g., DGL[184], a library tuned for deep learning on graphs.

12.4.6 Networks and Buses

Whenever a single device is insufficient for optimization we need to transfer data to and from it to synchronize processing. This is where networks and buses come in handy. We have a number of design parameters: bandwidth, cost, distance and flexibility. On one end we have WiFi which has a pretty good range, is very easy to use (no wires, after all), cheap but it offers comparatively mediocre bandwidth and latency. No machine learning researcher within their right mind would use it to build a cluster of servers. In what follows we focus on interconnects that are suitable for deep learning.

- **PCIe** is a dedicated bus for very high bandwidth point to point connections (up to 16 Gbs on PCIe 4.0) per lane. Latency is in the order of single-digit microseconds (5 µs). PCIe links are precious. Processors only have a limited number of them: AMD's EPYC 3 has 128 lanes, Intel's Xeon has up to 48 lanes per chip; on desktop grade CPUs the numbers are 20 (Ryzen 9) and 16 (Core i9) respectively. Since GPUs have typically 16 lanes this limits the number of GPUs that can connect to the CPU at full bandwidth. After all, they need to share the links with other high bandwidth peripherals such as storage and Ethernet. Just like with RAM access, large bulk transfers are preferable due to reduced packet overhead.
- **Ethernet** is the most commonly used way of connecting computers. While it is significantly slower than PCIe, it is very cheap and resilient to install and covers much longer distances. Typical bandwidth for low-grade servers is 1 GBit/s. Higher end devices (e.g., C5 instances[185] in the cloud) offer between 10 and 100 GBit/s bandwidth. As in all previous cases data transmission has significant overheads. Note that we almost never use raw Ethernet directly but rather a protocol that is executed on top of the physical interconnect (such as UDP or TCP/IP).

[183] https://github.com/gunrock/gunrock
[184] http://dgl.ai
[185] https://aws.amazon.com/ec2/instance-types/c5/

This adds further overhead. Like PCIe, Ethernet is designed to connect two devices, e.g., a computer and a switch.

- **Switches** allow us to connect multiple devices in a manner where any pair of them can carry out a (typically full bandwidth) point to point connection simultaneously. For instance, Ethernet switches might connect 40 servers at high cross-sectional bandwidth. Note that switches are not unique to traditional computer networks. Even PCIe lanes can be switched[186]. This occurs e.g., to connect a large number of GPUs to a host processor, as is the case for the P2 instances[187].
- **NVLink** is an alternative to PCIe when it comes to very high bandwidth interconnects. It offers up to 300 Gbit/s data transfer rate per link. Server GPUs (Volta V100) have 6 links whereas consumer grade GPUs (RTX 2080 Ti) have only one link, operating at a reduced 100 Gbit/s rate. We recommend to use NCCL[188] to achieve high data transfer between GPUs.

Summary

- Devices have overheads for operations. Hence it is important to aim for a small number of large transfers rather than many small ones. This applies to RAM, SSDs, Networks and GPUs.
- Vectorization is key for performance. Make sure you're aware of the specific abilities of your accelerator. E.g., some Intel Xeon CPUs are particularly good for INT8 operations, NVIDIA Volta GPUs excel at FP16 matrix-matrix operations and NVIDIA Turing shines at FP16, INT8 and INT4 operations.
- Numerical overflow due to small datatypes can be a problem during training (and to a lesser extent during inference).
- Aliasing can significantly degrade performance. For instance, memory alignment on 64 bit CPUs should be done with respect to 64 bit boundaries. On GPUs it's a good idea to keep convolution sizes aligned e.g., to TensorCores.
- Match your algorithms to the hardware (memory footprint, bandwidth, etc.). Great speedup (orders of magnitude) can be achieved when fitting the parameters into caches.
- We recommend that you sketch out the performance of a novel algorithm on paper before verifying the experimental results. Discrepancies of an order-of-magnitude or more are reasons for concern.
- Use profilers to debug performance bottlenecks.
- Training and inference hardware have different sweet spots in terms of price / performance.

[186] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches
[187] https://aws.amazon.com/ec2/instance-types/p2/
[188] https://github.com/NVIDIA/nccl

12.4.7 More Latency Numbers

The summary in Table 12.4.1 and Table 12.4.2 are due to Eliot Eshelman[189] who maintains an updated version of the numbers as a GitHub Gist[190].

Table 12.4.1: Common Latency Numbers.

Action	Time	Notes
L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
Branch mispredict	6 ns	15 ~ 20 cycles
L3 cache hit (unshared cache)	16 ns	42 cycles
L3 cache hit (shared in another core)	25 ns	65 cycles
Mutex lock/unlock	25 ns	
L3 cache hit (modified in another core)	29 ns	75 cycles
L3 cache hit (on a remote CPU socket)	40 ns	100 ~ 300 cycles (40 ~ 116 ns)
QPI hop to a another CPU (per hop)	40 ns	
64MB memory ref. (local CPU)	46 ns	TinyMemBench on Broadwell E5-2690v4
64MB memory ref. (remote CPU)	70 ns	TinyMemBench on Broadwell E5-2690v4
256MB memory ref. (local CPU)	75 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random write	94 ns	UCSD Non-Volatile Systems Lab
256MB memory ref. (remote CPU)	120 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random read	305 ns	UCSD Non-Volatile Systems Lab
Send 4KB over 100 Gbps HPC fabric	1 µs	MVAPICH2 over Intel Omni-Path
Compress 1KB with Google Snappy	3 µs	
Send 4KB over 10 Gbps ethernet	10 µs	
Write 4KB randomly to NVMe SSD	30 µs	DC P3608 NVMe SSD (QOS 99% is 500µs)
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCIe 3.0 x16 link
Read 4KB randomly from NVMe SSD	120 µs	DC P3608 NVMe SSD (QOS 99%)
Read 1MB sequentially from NVMe SSD	208 µs	~4.8GB/s DC P3608 NVMe SSD
Write 4KB randomly to SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Read 4KB randomly from SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Round trip within same datacenter	500 µs	One-way ping is ~250µs
Read 1MB sequentially from SATA SSD	2 ms	~550MB/s DC S3510 SATA SSD
Read 1MB sequentially from disk	5 ms	~200MB/s server HDD
Random Disk Access (seek+rotation)	10 ms	
Send packet CA->Netherlands->CA	150 ms	

Table 12.4.2: Latency Numbers for NVIDIA Tesla GPUs.

Action	Time	Notes
GPU Shared Memory access	30 ns	30~90 cycles (bank conflicts add latency)
GPU Global Memory access	200 ns	200~800 cycles
Launch CUDA kernel on GPU	10 µs	Host CPU instructs GPU to start kernel
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCI-Express x16 link

[189] https://gist.github.com/eshelman
[190] https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646

Exercises

1. Write C code to test whether there is any difference in speed between accessing memory aligned or misaligned relative to the external memory interface. Hint - be careful of caching effects.
2. Test the difference in speed between accessing memory in sequence or with a given stride.
3. How could you measure the cache sizes on a CPU?
4. How would you lay out data across multiple memory channels for maximum bandwidth? How would you lay it out if you had many small threads?
5. An enterprise class HDD is spinning at 10,000 rpm. What is the absolutely minimum time an HDD needs to spend worst case before it can read data (you can assume that heads move almost instantaneously)? Why are 2.5" HDDs becoming popular for commercial servers (relative to 3.5" and 5.25" drives)?
6. Assume that an HDD manufacturer increases the storage density from 1 Tbit per square inch to 5 Tbit per square inch. How much information can you store on a ring on a 2.5" HDD? Is there a difference between the inner and outer tracks?
7. The AWS P2 instances have 16 K80 Kepler GPUs. Use `lspci` on a p2.16xlarge and a p2.8xlarge instance to understand how the GPUs are connected to the CPUs. Hint - keep your eye out for PCI PLX bridges.
8. Going from 8 bit to 16 bit datatypes increases the amount of silicon approximately by 4x. Why? Why might NVIDIA have added INT4 operations to their Turing GPUs.
9. Given 6 high speed links between GPUs (such as for the Volta V100 GPUs), how would you connect 8 of them? Look up the connectivity used in the P3.16xlarge servers.
10. How much faster is it to read forward through memory vs. reading backwards? Does this number differ between different computers and CPU vendors? Why? Write C code and experiment with it.
11. Can you measure the cache size of your disk? What is it for a typical HDD? Do SSDs need a cache?
12. Measure the packet overhead when sending messages across the Ethernet. Look up the difference between UDP and TCP/IP connections.
13. Direct Memory Access allows devices other than the CPU to write (and read) directly to (from) memory. Why is this a good idea?
14. Look at the performance numbers for the Turing T4 GPU. Why does the performance 'only' double as you go from FP16 to INT8 and INT4?
15. What is the shortest time it should take for a packet on a roundtrip between San Francisco and Amsterdam? Hint - you can assume that the distance is 10,000km.

12.5 Training on Multiple GPUs

So far we discussed how to train models efficiently on CPUs and GPUs. We even showed how deep learning frameworks such as MXNet (and TensorFlow) allow one to parallelize computation and communication automatically between them in Section 12.3. Lastly, we showed in Section 5.6 how to list all available GPUs on a computer using `nvidia-smi`. What we did *not* discuss is how to actually parallelize deep learning training (we omit any discussion of *inference* on multiple GPUs here as it's a rather rarely used and advanced topic that goes beyond the scope of this book). Instead, we implied in passing that one would somehow split the data across multiple devices and make it work. The present section fills in the details and shows how to train a network in parallel when starting from scratch. Details on how to take advantage of functionality in Gluon is relegated to Section 12.6. We assume that the reader is familiar with minibatch SGD algorithms such as the ones described in Section 11.5.

12.5.1 Splitting the Problem

Let's start with a simple computer vision problem and a slightly archaic network, e.g., with multiple layers of convolutions, pooling, and possibly a few dense layers in the end. That is, let's start with a network that looks quite similar to LeNet (LeCun et al., 1998) or AlexNet (Krizhevsky et al., 2012). Given multiple GPUs (2 if it's a desktop server, 4 on a g4dn.12xlarge, 8 on an AWS p3.16xlarge, or 16 on a p2.16xlarge), we want to partition training in a manner as to achieve good speedup while simultaneously benefitting from simple and reproducible design choices. Multiple GPUs, after all, increase both *memory* and *compute* ability. In a nutshell, we have a number of choices, given a minibatch of training data that we want to classify.

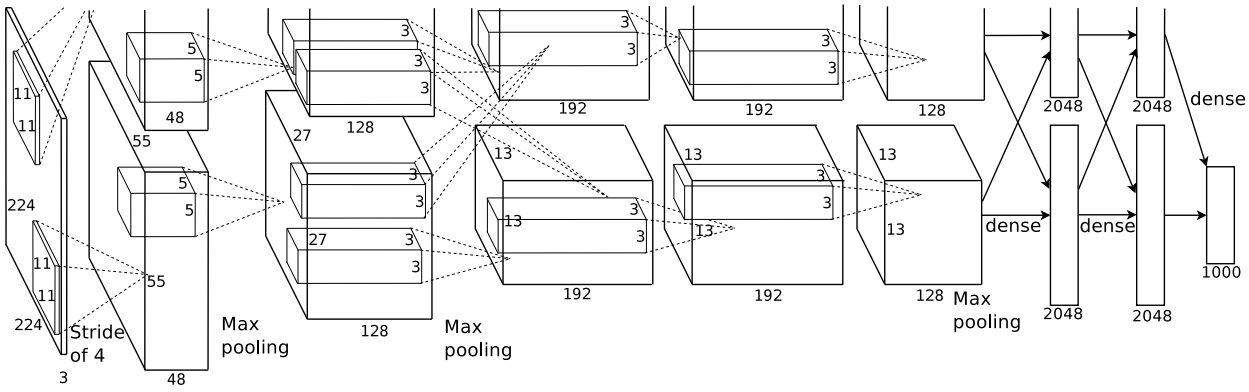


Fig. 12.5.1: Model parallelism in the original AlexNet design due to limited GPU memory.

- We could partition the network layers across multiple GPUs. That is, each GPU takes as input the data flowing into a particular layer, processes data across a number of subsequent layers and then sends the data to the next GPU.
  - This allows us to process data with larger networks when compared to what a single GPU could handle.
  - Memory footprint per GPU can be well controlled (it's a fraction of the total network footprint)
  - The interface between layers (and thus GPUs) requires tight synchronization. This can be tricky, in particular if the computational workloads are not properly matched between layers. The problem is exacerbated for large numbers of GPUs.

  - The interface between layers requires large amounts of data transfer (activations, gradients). This may overwhelm the bandwidth of the GPU buses.
  - Compute intensive, yet sequential operations are nontrivial to partition. See e.g., (Mirhoseini et al., 2017) for a best effort in this regard. It remains a difficult problem and it is unclear whether it is possible to achieve good (linear) scaling on nontrivial problems. We do not recommend it unless there is excellent framework / OS support for chaining together multiple GPUs.

- We could split the work required by individual layers. For instance, rather than computing 64 channels on a single GPU we could split up the problem across 4 GPUs, each of which generate data for 16 channels. Likewise, for a dense layer we could split the number of output neurons. Fig. 12.5.1 illustrates this design. The figure is taken from (Krizhevsky et al., 2012) where this strategy was used to deal with GPUs that had a very small memory footprint (2GB at the time).
  - This allows for good scaling in terms of computation, provided that the number of channels (or neurons) is not too small.
  - Multiple GPUs can process increasingly larger networks since the memory available scales linearly.
  - We need a *very large* number of synchronization / barrier operations since each layer depends on the results from all other layers.
  - The amount of data that needs to be transferred is potentially even larger than when distributing layers across GPUs. We do not recommend this approach due to its bandwidth cost and complexity.

- Lastly we could partition data across multiple GPUs. This way all GPUs perform the same type of work, albeit on different observations. Gradients are aggregated between GPUs after each minibatch.
  - This is the simplest approach and it can be applied in any situation.
  - Adding more GPUs does not allow us to train larger models.
  - We only need to synchronize after each minibatch. That said, it's highly desirable to start exchanging gradients parameters already while others are still being computed.
  - Large numbers of GPUs lead to very large minibatch sizes, thus reducing training efficiency.

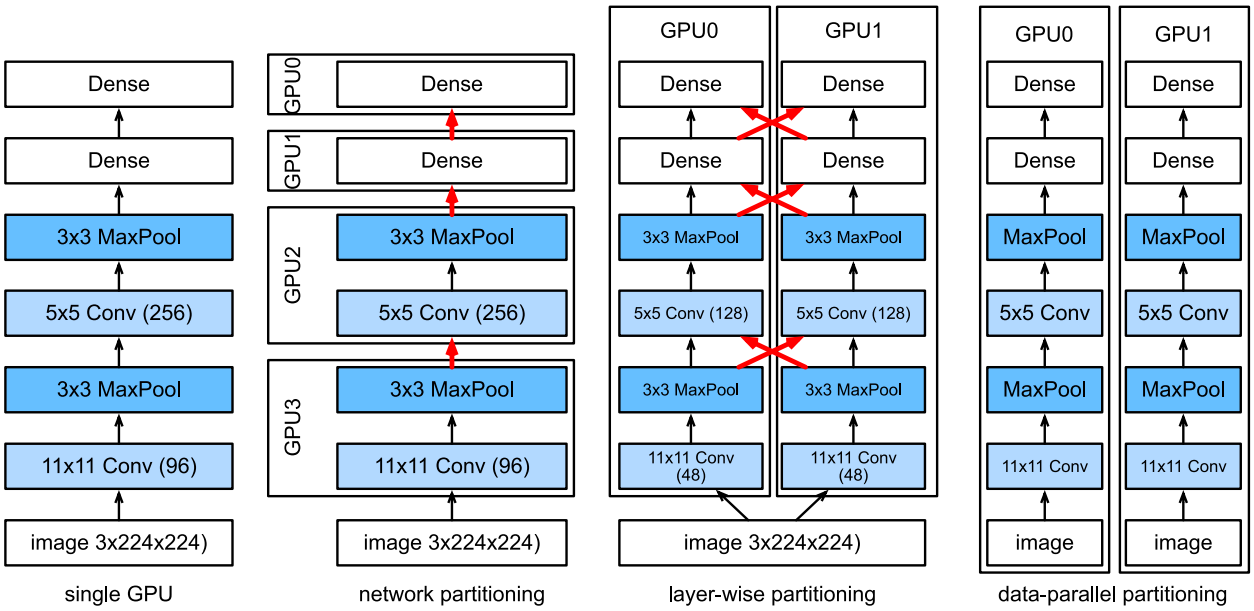


Fig. 12.5.2: Parallelization on multiple GPUs. From left to right - original problem, network partitioning, layer partitioning, data parallelism.

By and large, data parallelism is the most convenient way to proceed, provided that we have access to GPUs with sufficiently large memory. See also (Li et al., 2014) for a detailed description of partitioning for distributed training. GPU memory used to be a problem in the early days of deep learning. By now this issue has been resolved for all but the most unusual cases. We focus on data parallelism in what follows.

12.5.2 Data Parallelism

Assume that there are k GPUs on a machine. Given the model to be trained, each GPU will maintain a complete set of model parameters independently. Training proceeds as follows (see Fig. 12.5.3 for details on data parallel training on two GPUs):

- In any iteration of training, given a random minibatch, we split the examples in the batch into k portions and distribute them evenly across the GPUs.
- Each GPU calculates loss and gradient of the model parameters based on the minibatch subset it was assigned and the model parameters it maintains.
- The local gradients of each of the k GPUs are aggregated to obtain the current minibatch stochastic gradient.
- The aggregate gradient is re-distributed to each GPU.
- Each GPU uses this minibatch stochastic gradient to update the complete set of model parameters that it maintains.

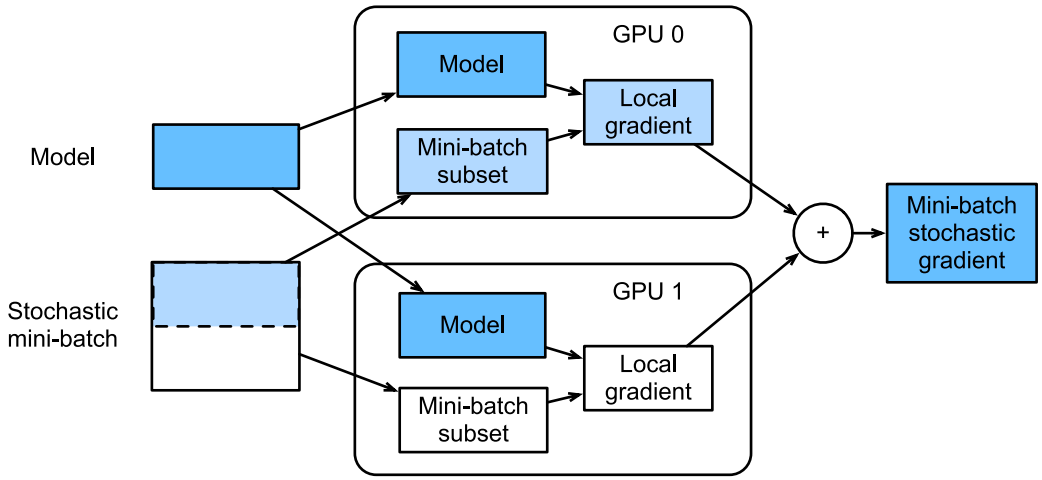


Fig. 12.5.3: Calculation of minibatch stochastic gradient using data parallelism and two GPUs.

A comparison of different ways of parallelization on multiple GPUs is depicted in Fig. 12.5.2. Note that in practice we *increase* the minibatch size k-fold when training on k GPUs such that each GPU has the same amount of work to do as if we were training on a single GPU only. On a 16 GPU server this can increase the minibatch size considerably and we may have to increase the learning rate accordingly. Also note that Section 7.5 needs to be adjusted (e.g., by keeping a separate batch norm coefficient per GPU). In what follows we will use Section 6.6 as the toy network to illustrate multi-GPU training. As always we begin by importing the relevant packages and modules.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()
```

12.5.3 A Toy Network

We use LeNet as introduced in Section 6.6. We define it from scratch to illustrate parameter exchange and synchronization in detail.

```
# Initialize model parameters
scale = 0.01
W1 = np.random.normal(scale=scale, size=(20, 1, 3, 3))
b1 = np.zeros(20)
W2 = np.random.normal(scale=scale, size=(50, 20, 5, 5))
b2 = np.zeros(50)
W3 = np.random.normal(scale=scale, size=(800, 128))
b3 = np.zeros(128)
W4 = np.random.normal(scale=scale, size=(128, 10))
b4 = np.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
    h1_conv = npx.convolution(data=X, weight=params[0], bias=params[1],
                              kernel=(3, 3), num_filter=20)
    h1_activation = npx.relu(h1_conv)
```

(continues on next page)

(continued from previous page)

```
    h1 = npx.pooling(data=h1_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
    h2_conv = npx.convolution(data=h1, weight=params[2], bias=params[3],
                              kernel=(5, 5), num_filter=50)
    h2_activation = npx.relu(h2_conv)
    h2 = npx.pooling(data=h2_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = np.dot(h2, params[4]) + params[5]
    h3 = npx.relu(h3_linear)
    y_hat = np.dot(h3, params[6]) + params[7]
    return y_hat

# Cross-entropy loss function
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

12.5.4 Data Synchronization

For efficient multi-GPU training we need two basic operations: firstly we need to have the ability to distribute a list of parameters to multiple devices and to attach gradients (`get_params`). Without parameters it's impossible to evaluate the network on a GPU. Secondly, we need the ability to sum parameters across multiple devices, i.e., we need an `allreduce` function.

```
def get_params(params, ctx):
    new_params = [p.copyto(ctx) for p in params]
    for p in new_params:
        p.attach_grad()
    return new_params
```

Let's try it out by copying the model parameters of lenet to gpu(0).

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 weight: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] @gpu(0)
b1 grad: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] @gpu(0)
```

Since we didn't perform any computation yet, the gradient with regard to the bias weights is still 0. Now let's assume that we have a vector distributed across multiple GPUs. The following allreduce function adds up all vectors and broadcasts the result back to all GPUs. Note that for this to work we need to copy the data to the device accumulating the results.

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].copyto(data[0].context)
    for i in range(1, len(data)):
        data[0].copyto(data[i])
```

Let's test this by creating vectors with different values on different devices and aggregate them.

```
data = [np.ones((1, 2), ctx=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
before allreduce:
 [[1. 1.]] @gpu(0)
 [[2. 2.]] @gpu(1)
after allreduce:
 [[3. 3.]] @gpu(0)
 [[3. 3.]] @gpu(1)
```

12.5.5 Distributing Data

We need a simple utility function to distribute a minibatch evenly across multiple GPUs. For instance, on 2 GPUs we'd like to have half of the data to be copied to each of the GPUs. Since it's more convenient and more concise, we use the built-in split and load function in Gluon (to try it out on a 4×5 matrix).

```
data = np.arange(20).reshape(4, 5)
ctx = [npx.gpu(0), npx.gpu(1)]
split = gluon.utils.split_and_load(data, ctx)
print('input :', data)
print('load into', ctx)
print('output:', split)
```

```
input : [[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]]
load into [gpu(0), gpu(1)]
output: [array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]], ctx=gpu(0)), array([[10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.]], ctx=gpu(1))]
```

For later reuse we define a `split_batch` function which splits both data and labels.

```
# Saved in the d2l package for later use
def split_batch(X, y, ctx_list):
    """Split X and y into multiple devices specified by ctx."""
    assert X.shape[0] == y.shape[0]
    return (gluon.utils.split_and_load(X, ctx_list),
            gluon.utils.split_and_load(y, ctx_list))
```

12.5.6 Training

Now we can implement multi-GPU training on a single minibatch. Its implementation is primarily based on the data parallelism approach described in this section. We will use the auxiliary functions we just discussed, `allreduce` and `split_and_load`, to synchronize the data among multiple GPUs. Note that we don't need to write any specific code to achieve parallelism. Since the compute graph doesn't have any dependencies across devices within a minibatch, it is executed in parallel *automatically*.

```
def train_batch(X, y, gpu_params, ctx_list, lr):
    gpu_Xs, gpu_ys = split_batch(X, y, ctx_list)
    with autograd.record():  # Loss is calculated separately on each GPU
        losses = [loss(lenet(gpu_X, gpu_W), gpu_y)
                  for gpu_X, gpu_y, gpu_W in zip(gpu_Xs, gpu_ys, gpu_params)]
    for l in losses:  # Back Propagation is performed separately on each GPU
        l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    for i in range(len(gpu_params[0])):
        allreduce([gpu_params[c][i].grad for c in range(len(ctx_list))])
    # The model parameters are updated separately on each GPU
    for param in gpu_params:
        d2l.sgd(param, lr, X.shape[0])  # Here, we use a full-size batch
```

Now, we can define the training function. It is slightly different from the ones used in the previous chapters: we need to allocate the GPUs and copy all the model parameters to all devices. Obviously each batch is processed using `train_batch` to deal with multiple GPUs. For convenience (and conciseness of code) we compute the accuracy on a single GPU (this is *inefficient* since the other GPUs are idle).

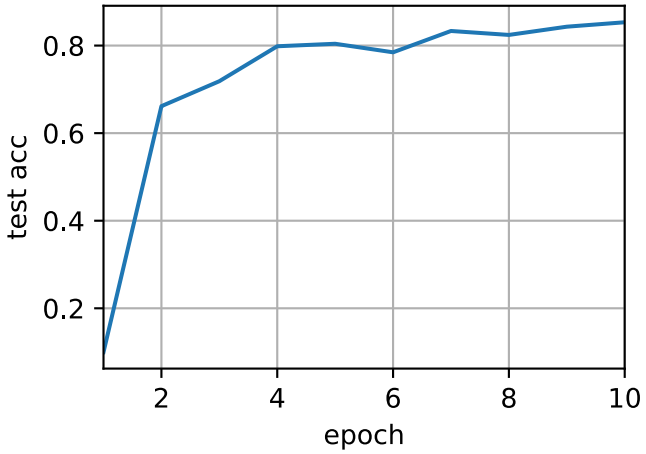
```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    ctx_list = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Copy model parameters to num_gpus GPUs
    gpu_params = [get_params(params, c) for c in ctx_list]
    # num_epochs, times, acces = 10, [], []
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, gpu_params, ctx_list, lr)
            npx.waitall()
        timer.stop()
        # Verify the model on GPU 0
        animator.add(epoch+1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, gpu_params[0]), test_iter, ctx[0]),))
    print('test acc: %.2f, %.1f sec/epoch on %s' % (
        animator.Y[0][-1], timer.avg(), ctx_list))
```

12.5.7 Experiment

Let's see how well this works on a single GPU. We use a batch size of 256 and a learning rate of 0.2.

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

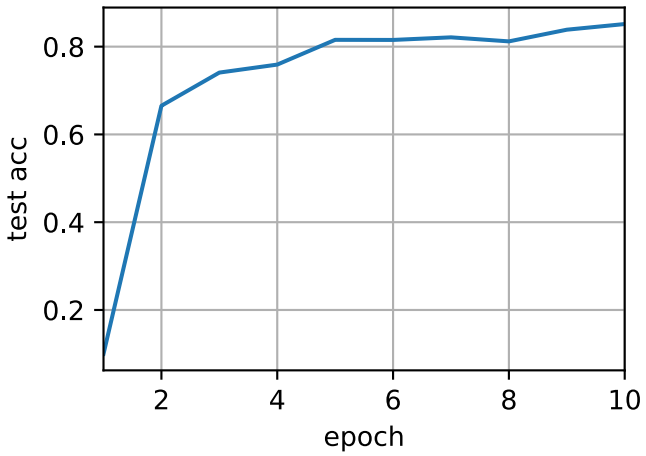
```
test acc: 0.85, 2.0 sec/epoch on [gpu(0)]
```



By keeping the batch size and learning rate unchanged and changing the number of GPUs to 2, we can see that the improvement in test accuracy is roughly the same as in the results from the previous experiment. In terms of the optimization algorithms, they are identical. Unfortunately there's no meaningful speedup to be gained here: the model is simply too small; moreover we only have a small dataset, where our slightly unsophisticated approach to implementing multi-GPU training suffered from significant Python overhead. We will encounter more complex models and more sophisticated ways of parallelization going forward. Let's see what happens nonetheless for MNIST.

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

```
test acc: 0.85, 3.9 sec/epoch on [gpu(0), gpu(1)]
```



Summary

- There are multiple ways to split deep network training over multiple GPUs. We could split them between layers, across layers, or across data. The former two require tightly choreographed data transfers. Data parallelism is the simplest strategy.
- Data parallel training is straightforward. However, it increases the effective minibatch size to be efficient.
- Data is split across multiple GPUs, each GPU executes its own forward and backward operation and subsequently gradients are aggregated and results broadcast back to the GPUs.
- Large minibatches may require a slightly increased learning rate.

Exercises

1. When training on multiple GPUs, change the minibatch size from b to $k \cdot b$, i.e., scale it up by the number of GPUs.
2. Compare accuracy for different learning rates. How does it scale with the number of GPUs.
3. Implement a more efficient allreduce that aggregates different parameters on different GPUs (why is this more efficient in the first place).
4. Implement multi-GPU test accuracy computation.

12.6 Concise Implementation for Multiple GPUs

Implementing parallelism from scratch for every new model is no fun. Moreover, there's significant benefit in optimizing synchronization tools for high performance. In the following we'll show how to do this using Gluon. The math and the algorithms are the same as in Section 12.5. As before we begin by importing the required modules (quite unsurprisingly you'll need at least two GPUs to run this notebook).

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

12.6.1 A Toy Network

Let's use a slightly more meaningful network than LeNet from the previous section that's still sufficiently easy and quick to train. We pick a ResNet-18 variant (He et al., 2016a). Since the input images are tiny we modify it slightly. In particular, the difference to Section 7.6 is that we use a smaller convolution kernel, stride, and padding at the beginning. Moreover, we remove the max-pooling layer.

```
# Saved in the d2l package for later use
def resnet18(num_classes):
    """A slightly modified ResNet-18 model."""
    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.Sequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(d2l.Residual(
                    num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(d2l.Residual(num_channels))
        return blk

    net = nn.Sequential()
    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the maximum pooling layer
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))
    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

12.6.2 Parameter Initialization and Logistics

The `initialize` method allows us to set initial defaults for parameters on a device of our choice. For a refresher see Section 4.8. What is particularly convenient is that it also lets us initialize the network on *multiple* devices simultaneously. Let's try how this works in practice.

```
net = resnet18(10)
# get a list of GPUs
ctx = d2l.try_all_gpus()
# initialize the network on all of them
net.initialize(init=init.Normal(sigma=0.01), ctx=ctx)
```

Using the `split_and_load` function introduced in the previous section we can divide a minibatch of data and copy portions to the list of devices provided by the context variable. The network object *automatically* uses the appropriate GPU to compute the value of the forward pass. As before we generate 4 observations and split them over the GPUs.

```
x = np.random.uniform(size=(4, 1, 28, 28))
gpu_x = gluon.utils.split_and_load(x, ctx)
net(gpu_x[0]), net(gpu_x[1])
```

```
(array([[ 2.2610193e-06,  2.2045974e-06, -5.4046782e-06,  1.2869954e-06,
          5.1373149e-06, -3.8298003e-06,  1.4339014e-07,  5.4683451e-06,
         -2.8279194e-06, -3.9651113e-06],
        [ 2.0698667e-06,  2.0084665e-06, -5.6382501e-06,  1.0498469e-06,
          5.5506416e-06, -4.1065468e-06,  6.0830143e-07,  5.4521765e-06,
         -3.7365030e-06, -4.1891640e-06]], ctx=gpu(0)),
 array([[ 2.4629794e-06,  2.6015521e-06, -5.4362622e-06,  1.2938231e-06,
          5.6387898e-06, -4.1360104e-06,  3.5758922e-07,  5.5125238e-06,
         -3.1957329e-06, -4.2976321e-06],
        [ 1.9431675e-06,  2.2600425e-06, -5.2698206e-06,  1.4807410e-06,
          5.4830930e-06, -3.9678889e-06,  7.5752268e-08,  5.6764361e-06,
         -3.2530238e-06, -4.0943960e-06]], ctx=gpu(1)))
```

Once data passes through the network, the corresponding parameters are initialized *on the device the data passed through*. This means that initialization happens on a per-device basis. Since we picked GPU 0 and GPU 1 for initialization, the network is initialized only there, and not on the CPU. In fact, the parameters don't even exist on the device. We can verify this by printing out the parameters and observing any errors that might arise.

```
weight = net[0].params.get('weight')

try:
    weight.data()
except RuntimeError:
    print('not initialized on cpu')
weight.data(ctx[0])[0], weight.data(ctx[1])[0]
```

```
not initialized on cpu
```

```
(array([[[ 0.01382882, -0.01183044,  0.01417866],
         [-0.00319718,  0.00439528,  0.02562625],
         [-0.00835081,  0.01387452, -0.01035946]]], ctx=gpu(0)),
 array([[[ 0.01382882, -0.01183044,  0.01417866],
         [-0.00319718,  0.00439528,  0.02562625],
         [-0.00835081,  0.01387452, -0.01035946]]], ctx=gpu(1)))
```

Lastly let's replace the code to evaluate the accuracy by one that works in parallel across multiple devices. This serves as a replacement of the `evaluate_accuracy_gpu` function from Section 6.6. The main difference is that we split a batch before invoking the network. All else is essentially identical.

```
# Saved in the d2l package for later use
def evaluate_accuracy_gpus(net, data_iter, split_f=d2l.split_batch):
    # Query the list of devices
    ctx = list(net.collect_params().values())[0].list_ctx()
    metric = d2l.Accumulator(2)  # num_corrected_examples, num_examples
    for features, labels in data_iter:
        Xs, ys = split_f(features, labels, ctx)
        pys = [net(X) for X in Xs]  # Run in parallel
        metric.add(sum(float(d2l.accuracy(py, y)) for py, y in zip(pys, ys)),
                   labels.size)
    return metric[0]/metric[1]
```

12.6.3 Training

As before, the training code needs to perform a number of basic functions for efficient parallelism:

- Network parameters need to be initialized across all devices.
- While iterating over the dataset minibatches are to be divided across all devices.
- We compute the loss and its gradient in parallel across devices.
- Losses are aggregated (by the trainer method) and parameters are updated accordingly.

In the end we compute the accuracy (again in parallel) to report the final value of the network. The training routine is quite similar to implementations in previous chapters, except that we need to split and aggregate data.

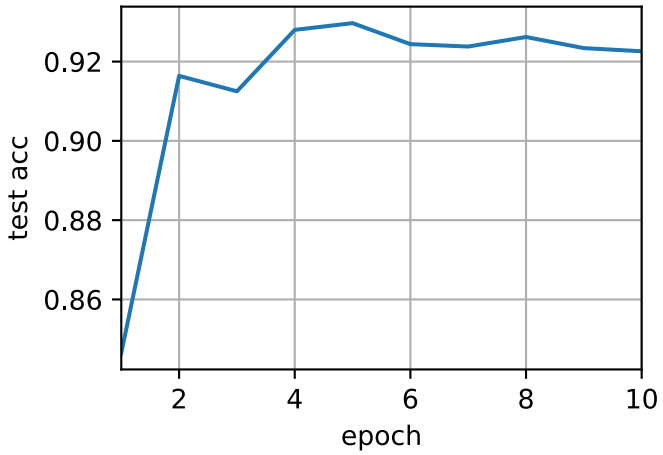
```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    ctx = [d2l.try_gpu(i) for i in range(num_gpus)]
    net.initialize(init=init.Normal(sigma=0.01), ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        timer.start()
        for features, labels in train_iter:
            Xs, ys = d2l.split_batch(features, labels, ctx)
            with autograd.record():
                losses = [loss(net(X), y) for X, y in zip(Xs, ys)]
            for l in losses:
                l.backward()
            trainer.step(batch_size)
        npx.waitall()
        timer.stop()
        animator.add(epoch+1, (evaluate_accuracy_gpus(net, test_iter),))
    print('test acc: %.2f, %.1f sec/epoch on %s' % (
        animator.Y[0][-1], timer.avg(), ctx))
```

12.6.4 Experiments

Let's see how this works in practice. As a warmup we train the network on a single GPU.

```
train(num_gpus=1, batch_size=256, lr=0.1)
```

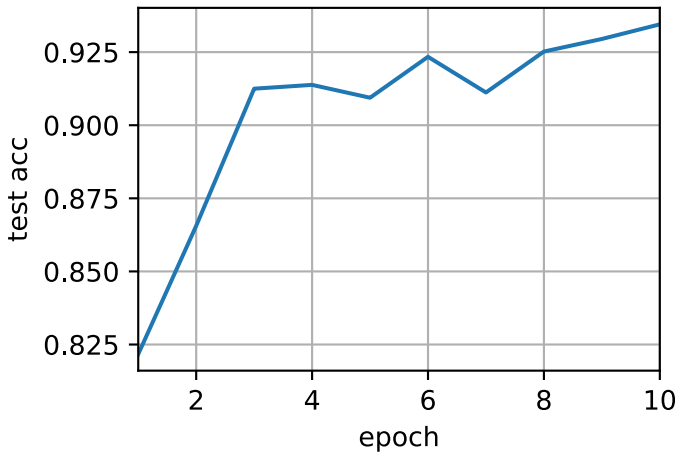
```
test acc: 0.92, 13.1 sec/epoch on [gpu(0)]
```



Next we use 2 GPUs for training. Compared to LeNet the model for ResNet-18 is considerably more complex. This is where parallelization shows its advantage. The time for computation is meaningfully larger than the time for synchronizing parameters. This improves scalability since the overhead for parallelization is less relevant.

```
train(num_gpus=2, batch_size=512, lr=0.2)
```

```
test acc: 0.93, 6.7 sec/epoch on [gpu(0), gpu(1)]
```



Summary

- Gluon provides primitives for model initialization across multiple devices by providing a context list.
- Data is automatically evaluated on the devices where the data can be found.
- Take care to initialize the networks on each device before trying to access the parameters on that device. Otherwise you will encounter an error.
- The optimization algorithms automatically aggregate over multiple GPUs.

Exercises

1. This section uses ResNet-18. Try different epochs, batch sizes, and learning rates. Use more GPUs for computation. What happens if you try this on a p2.16xlarge instance with 16 GPUs?
2. Sometimes, different devices provide different computing power. We could use the GPUs and the CPU at the same time. How should we divide the work? Is it worth the effort? Why? Why not?
3. What happens if we drop `npx.waitall()`? How would you modify training such that you have an overlap of up to two steps for parallelism?

12.7 Parameter Servers

As we move from single GPUs to multiple GPUs and then to multiple servers containing multiple GPUs, possibly all spread out across multiple racks and network switches our algorithms for distributed and parallel training need to become much more sophisticated. Details matter since different interconnects have very different bandwidth (e.g. NVLink can offer up to 100GB/s across 6 links in an appropriate setting, PCIe 3.0 16x lanes offer 16GB/s while even high speed 100 GbE Ethernet only amounts to 10GB/s). At the same time it's unreasonable to expect that a statistical modeler be an expert in networking and systems.

The core idea of the parameter server was introduced in (Smola & Narayanamurthy, 2010) in the context of distributed latent variable models. A description of the push and pull semantics then followed in (Ahmed et al., 2012) and a description of the system and an open source library followed in (Li et al., 2014). In the following we will motivate the components needed for efficiency.

12.7.1 Data Parallel Training

Let's review the data parallel training approach to distributed training. We will use this to the exclusion of all others in this section since it's significantly simpler to implement in practice. There are virtually no use cases (besides deep learning on graphs) where any other strategy for parallelism is preferred since GPUs have plenty of memory nowadays. Fig. 12.7.1 describes the variant of data parallelism that we implemented in the previous section. The key aspect in it is that the aggregation of gradients occurs on GPU0 before the updated parameters are rebroadcast to all GPUs.

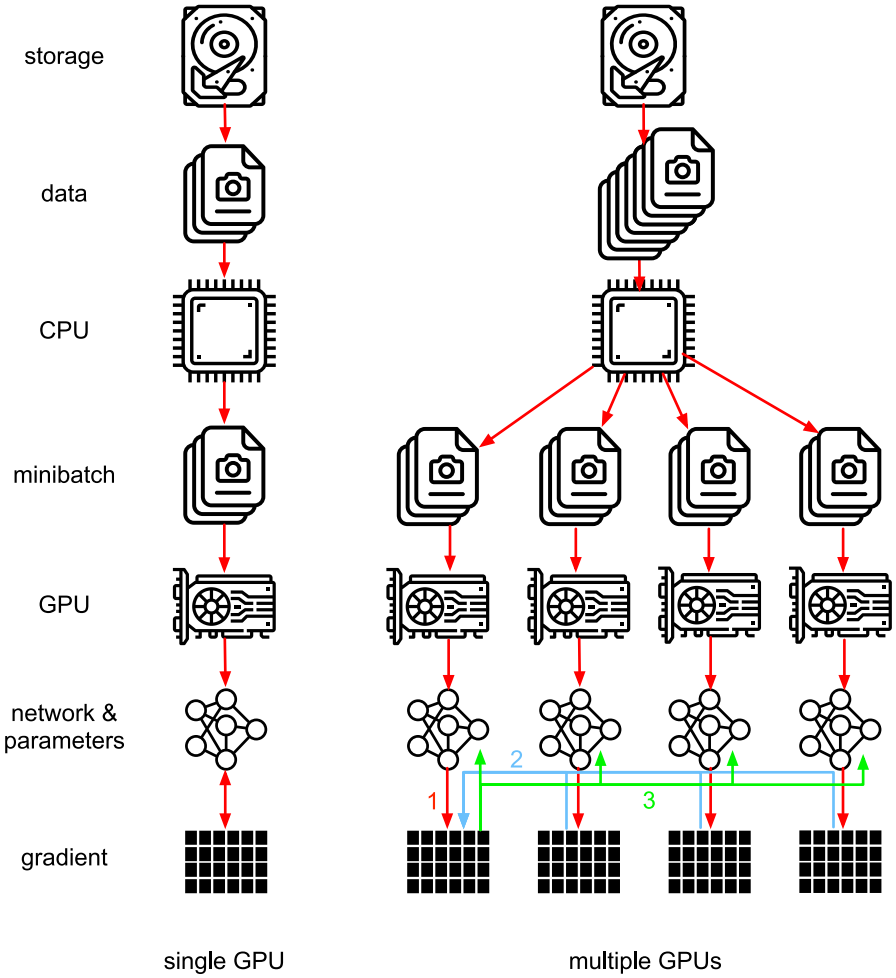


Fig. 12.7.1: Left: single GPU training; Right: a variant of multi-GPU training. It proceeds as follows. (1) we compute loss and gradient, (2) all gradients are aggregated on one GPU, (3) parameter update happens and the parameters are re-distributed to all GPUs.

In retrospect, the decision to aggregate on GPU0 seems rather ad-hoc. After all, we might just as well aggregate on the CPU. In fact, we could even decide to aggregate some of the parameters on one GPU and some others on another. Provided that the optimization algorithm supports this, there's no real reason for why we couldn't. For instance, if we have four parameter vectors $\mathbf{v}_1, \ldots \mathbf{v}_4$ with associated gradients $\mathbf{g}_1, \ldots \mathbf{g}_4$ we could aggregate the gradients on one GPU each.

$$\mathbf{g}_i = \sum_{j \in \text{GPUs}} \mathbf{g}_{ij} \tag{12.7.1}$$

This reasoning seems arbitrary and frivolous. After all, the math is the same throughout. However, we are dealing with real physical hardware where different buses have different bandwidth as discussed in Section 12.4. Consider a real 4-way GPU server as described in Fig. 12.7.2. If it's particularly well connected, it might have a 100 GbE network card. More typical numbers are in the 1-10 GbE range with an effective bandwidth of 100MB/s to 1GB/s. Since the CPUs have too few PCIe lanes to connect to all GPUs directly (e.g. consumer grade Intel CPUs have 24 lanes) we need a multiplexer[194]. The bandwidth from the CPU on a 16x Gen3 link is 16GB/s. This is also the speed at which *each* of the GPUs is connected to the switch. This means that it's more effective to communicate between the

[194] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches

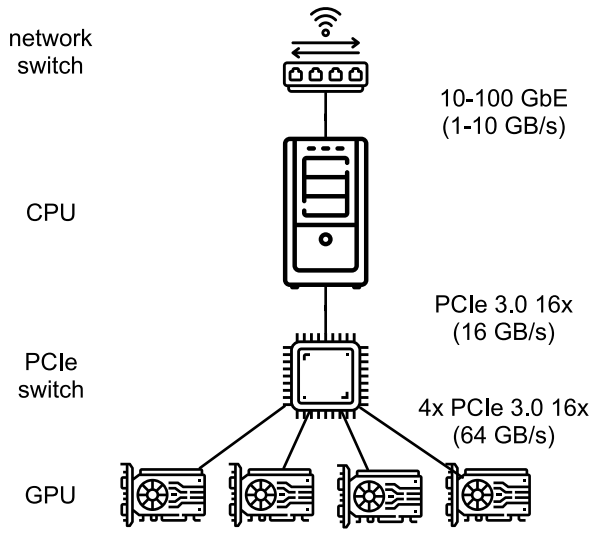


Fig. 12.7.2: Foo.

For the sake of the argument let's assume that the gradients 'weigh' 160MB. In this case it takes 30ms to send the gradients from all 3 remaining GPUs to the fourth one (each transfer takes 10ms = 160MB / 16 GB/s). Add another 30ms to transmit the weight vectors back we arrive at a total of 60ms. If we send all data to the CPU we incur a penalty of 40ms since *each* of the four GPUs needs to send the data to the CPU, yielding a total of 80ms. Lastly assume that we are able to split the gradients into 4 parts of 40MB each. Now we can aggregate each of the parts on a different GPU *simultaneously* since the PCIe switch offers a full-bandwidth operation between all links. Instead of 30ms this takes 7.5ms, yielding a total of 15ms for a synchronization operation. In short, depending on how we synchronize parameters the same operation can take anywhere from 15ms to 80ms. Fig. 12.7.3 depicts the different strategies for exchanging parameters.

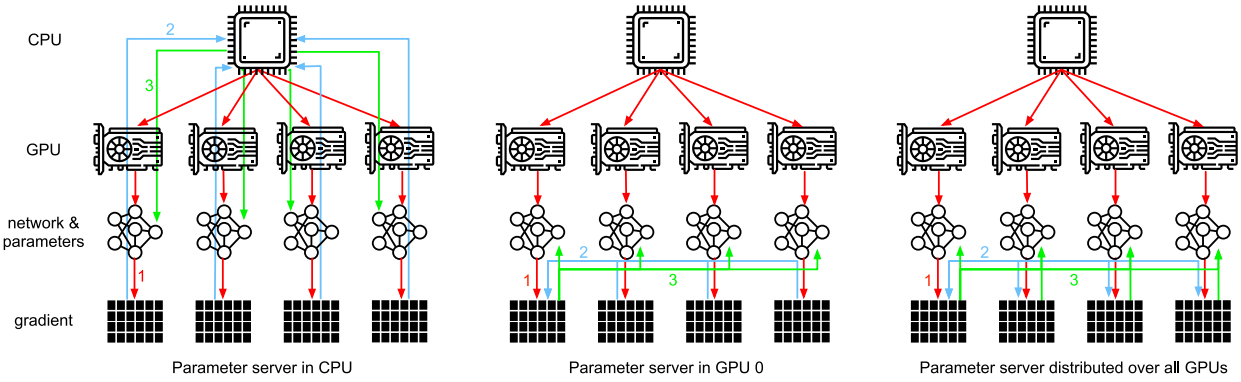


Fig. 12.7.3: Synchronization strategies.

Note that we have yet another tool at our disposal when it comes to improving performance: in a deep network it takes some time to compute all gradients from the top to the bottom. We can begin synchronizing gradients for some parameter groups even while we're still busy computing them for others (the technical details for that are somewhat involved). See e.g. (Sergeev & DelBalso, 2018) for details on how to do this in Horovod[195].

[195] https://github.com/horovod/horovod

12.7.2 Ring Synchronization

When it comes to synchronization on modern deep learning hardware we often encounter significantly bespoke network connectivity. For instance, the AWS P3.16xlarge and NVIDIA DGX-2 instances share the connectivity structure of Fig. 12.7.4. Each GPU connects to a host CPU via a PCIe link which operates at best at 16 GB/s. Additionally each GPU also has 6 NVLink connections, each of which is capable of transferring 300 Gbit/s bidirectionally. This amounts to around 18 GB/s per link per direction. In short, the aggregate NVLink bandwidth is significantly higher than the PCIe bandwidth. The question is how to use it most efficiently.

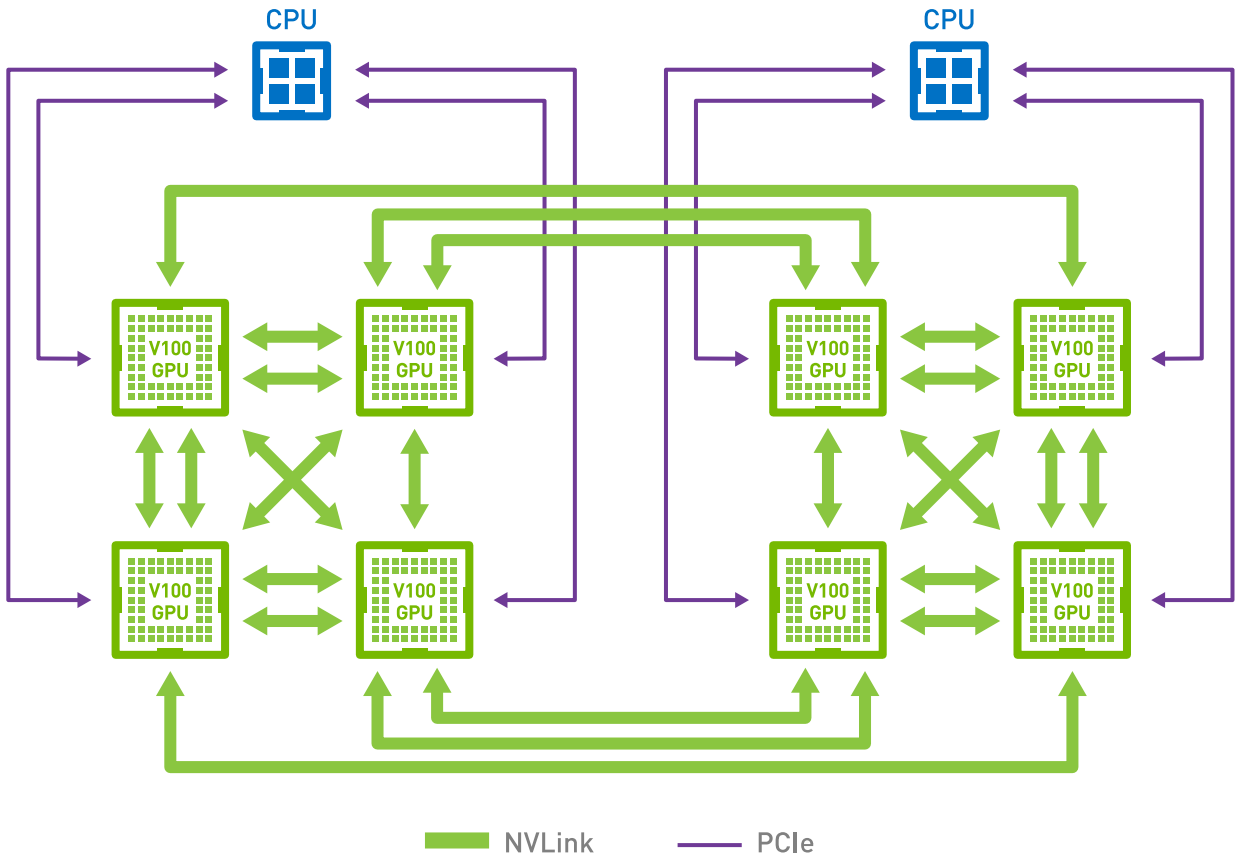


Fig. 12.7.4: NVLink connectivity on 8GPU V100 servers (image courtesy of NVIDIA).

It turns out (Wang et al., 2018) that the optimal synchronization strategy is to decompose the network into two rings and to use them to synchronize data directly. Fig. 12.7.5 illustrates that the network can be decomposed into one ring (1-2-3-4-5-6-7-8-1) with double NVLink bandwidth and into one (1-4-6-3-5-8-2-7-1) with regular bandwidth. Designing an efficient synchronization protocol in this case is nontrivial.

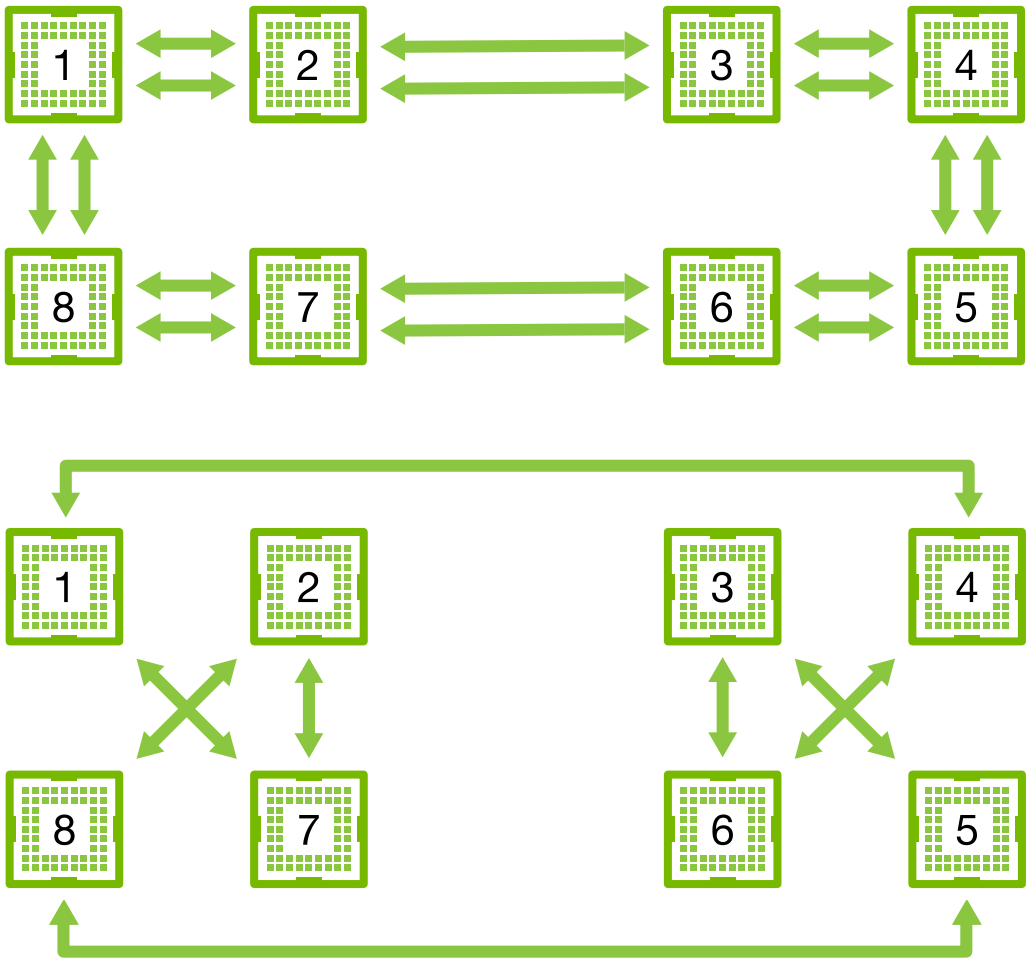


Fig. 12.7.5: Decomposition of the NVLink network into two rings.

Consider the following thought experiment: given a ring of n compute nodes (or GPUs) we can send gradients from the first to the second node. There it is added to the local gradient and sent on to the third node, and so on. After $n-1$ steps the aggregate gradient can be found in the last-visited node. That is, the time to aggregate gradients grows linearly with the number of nodes. But if we do this the algorithm is quite inefficient. After all, at any time there's only one of the nodes communicating. What if we broke the gradients into n chunks and started synchronizing chunk i starting at node i. Since each chunk is of site $1/n$ the total time is now $(n-1)/n \approx 1$. In other words, the time spent to aggregate gradients *does not grow* as we increase the size of the ring. This is quite an astonishing result. Fig. 12.7.6 illustrates the sequence of steps on $n = 4$ nodes.

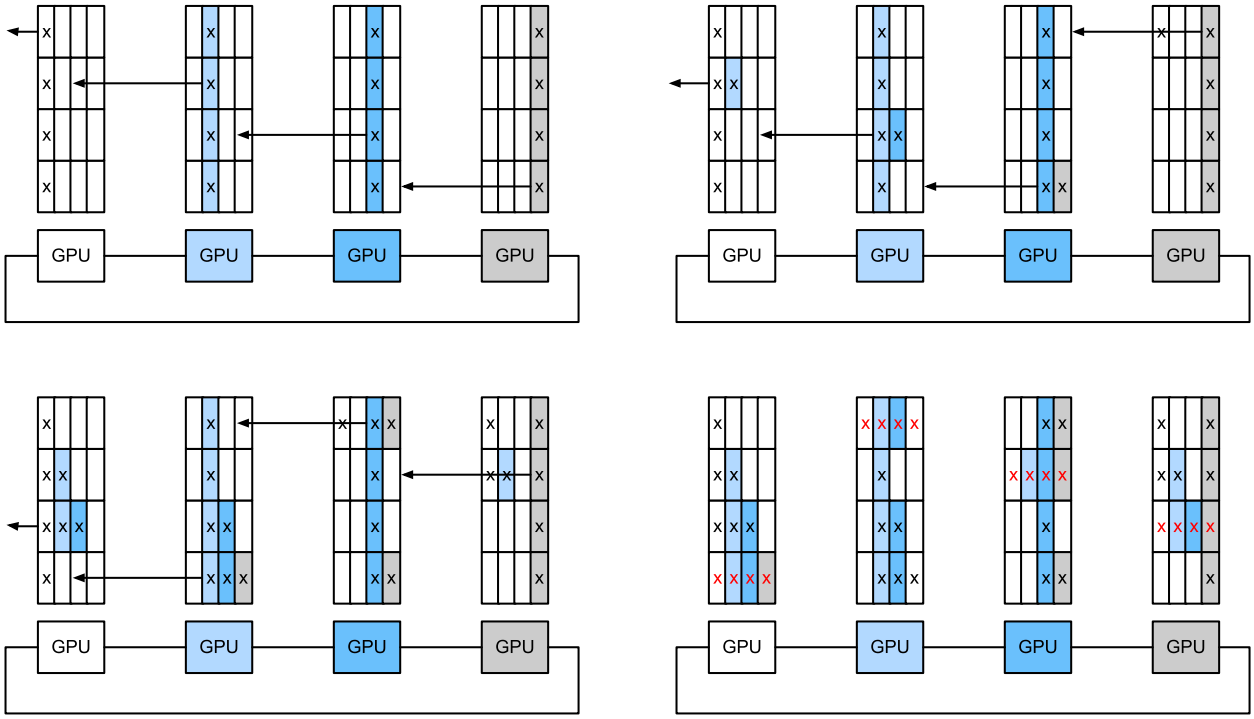


Fig. 12.7.6: Ring synchronization across 4 nodes. Each node starts transmitting parts of gradients to its left neighbor until the assembled gradient can be found in its right neighbor.

If we use the same example of synchronizing 160MB across 8 V100 GPUs we arrive at approximately $2 \cdot 160\text{MB}/(3 \cdot 18\text{GB/s}) \approx 6\text{ms}$ This is quite a bit better than using the PCIe bus, even though we are now using 8 GPUs. Note that in practice these numbers are quite a bit worse, since deep learning frameworks often fail to assemble communication into large burst transfers. Moreover, timing is critical. Note that there is a common misconception that ring synchronization is fundamentally different from other synchronization algorithms. The only difference is that the synchronization path is somewhat more elaborate when compared to a simple tree.

12.7.3 Multi-Machine Training

Distributed training on multiple machines adds a further challenge: we need to communicate with servers that are only connected across a comparatively lower bandwidth fabric which can be over an order of magnitude slower in some cases. Synchronization across devices is tricky. After all, different machines running training code will have subtly different speed. Hence we need to *synchronize* them if we want to use synchronous distributed optimization. Fig. 12.7.7 illustrates how distributed parallel training occurs.

1. A (different) batch of data is read on each machine, split across multiple GPUs and transferred to GPU memory. There predictions and gradients are computed on each GPU batch separately.
2. The gradients from all local GPUs are aggregated on one GPU (or alternatively parts of it are aggregated over different GPUs.
3. The gradients are sent to the CPU.
4. The CPU sends the gradients to a central parameter server which aggregates all the gradients.

5. The aggregate gradients are then used to update the weight vectors and the updated weight vectors are broadcast back to the individual CPUs.
6. The information is sent to one (or multiple) GPUs.
7. The updated weight vectors are spread across all GPUs.

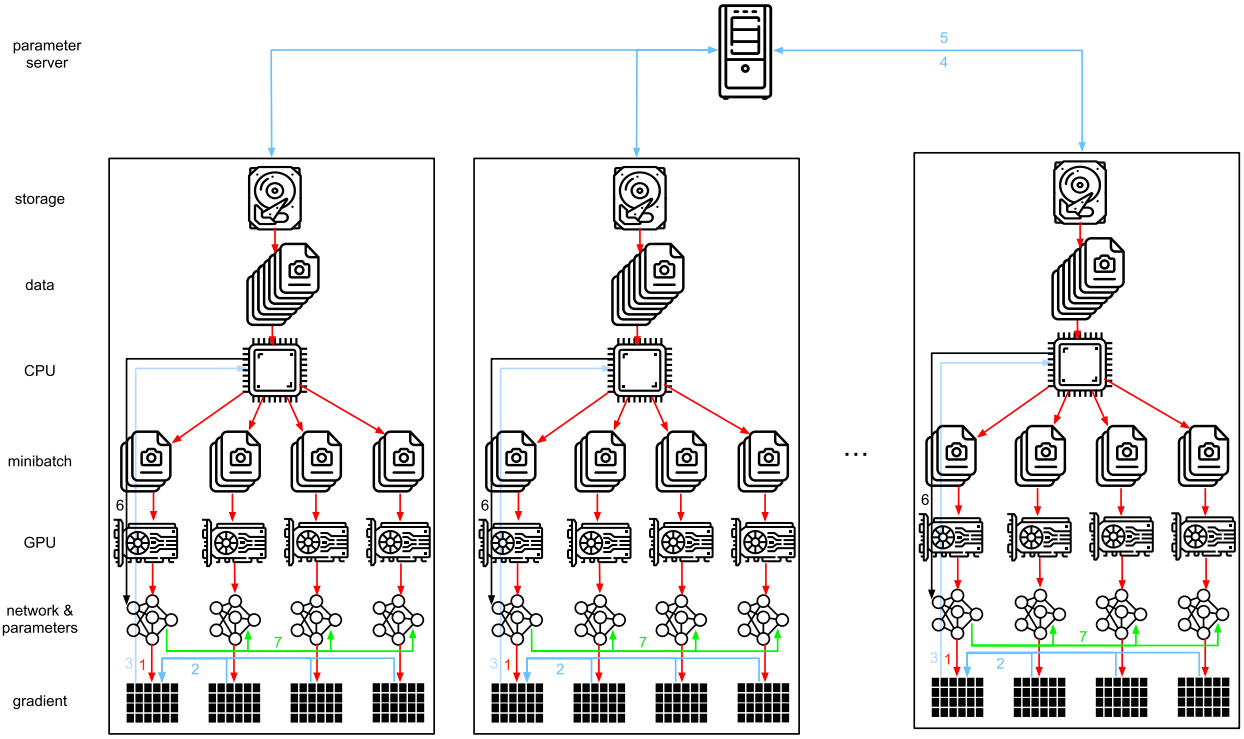


Fig. 12.7.7: Multi-machine multi-GPU distributed parallel training.

Each of these operations seems rather straightforward. And, indeed, they can be carried out efficiently *within* a single machine. Once we look at multiple machines, though, we can see that the central parameter server becomes the bottleneck. After all, the bandwidth per server is limited, hence for m workers the time it takes to send all gradients to the server is $O(m)$. We can break through this barrier by increasing the number of servers to n. At this point each server only needs to store $O(1/n)$ of the parameters, hence the total time for updates and optimization becomes $O(m/n)$. Matching both numbers yields constant scaling regardless of how many workers we are dealing with. In practice we use the *same* machines both as workers and as servers. Fig. 12.7.8 illustrates the design. See also (Li et al., 2014) for details. In particular, ensuring that multiple machines work without unreasonable delays is nontrivial. We omit details on barriers and will only briefly touch on synchronous and asynchronous updates below.

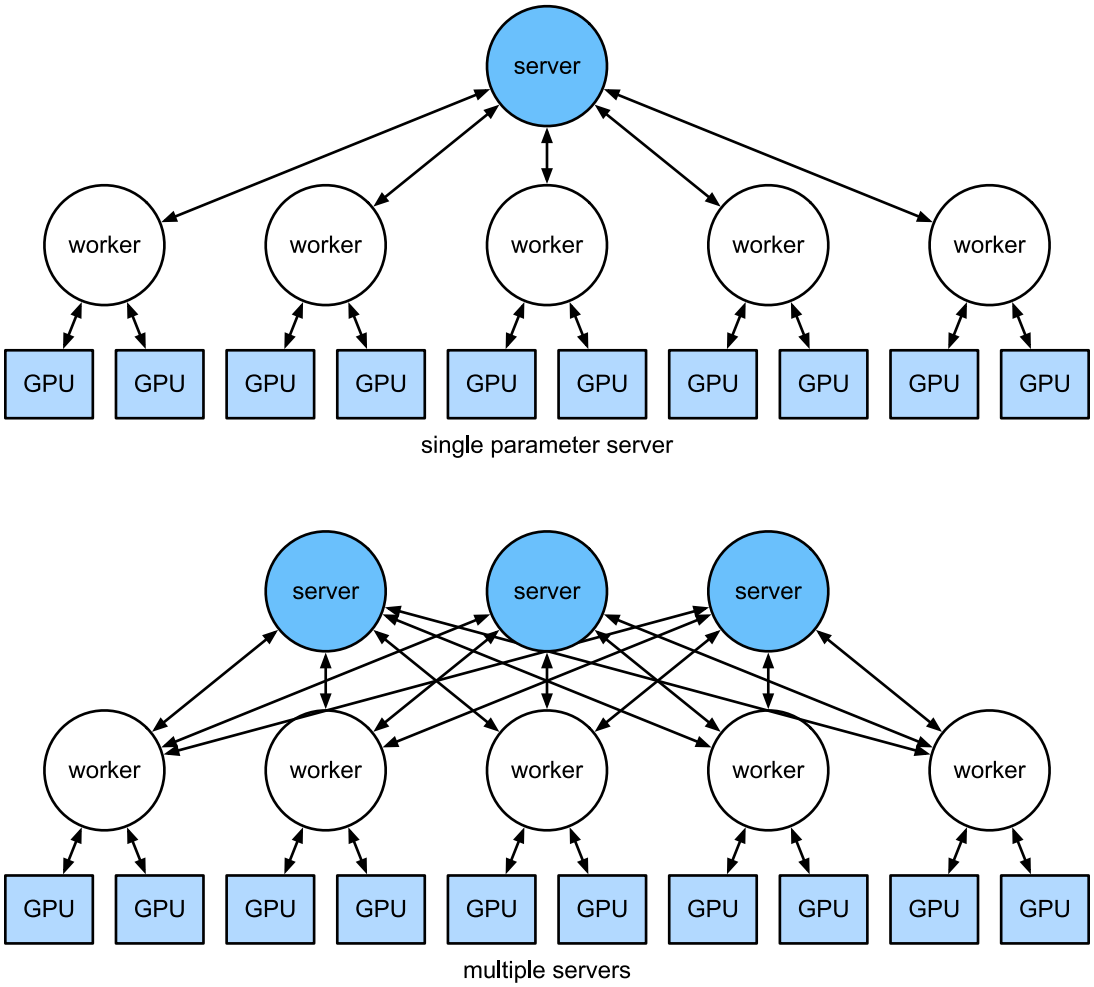


Fig. 12.7.8: Top - a single parameter server is a bottleneck since its bandwidth is finite. Bottom - multiple parameter servers store parts of the parameters with aggregate bandwidth.

12.7.4 (key,value) Stores

Implementing the steps required for distributed multi-GPU training in practice is nontrivial. In particular, given the many different choices that we might encounter. This is why it pays to use a common abstraction, namely that of a (key,value) store with redefined update semantics. Across many servers and many GPUs the gradient computation can be defined as

$$\mathbf{g}_i = \sum_{k \in \text{workers}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}. \tag{12.7.2}$$

The key aspect in this operation is that it is a *commutative reduction*, that is, it turns many vectors into one and the order in which the operation is applied doesn't matter. This is great for our purposes since we don't (need to) have fine grained control over when which gradient is received. Note that it's possible for us to perform the reduction stagewise. Furthermore, note that this operation is independent between blocks i pertaining to different parameters (and gradients).

This allows us to define the following two operations: push, which accumulates gradients, and pull, which retrieves aggregate gradients. Since we have many different sets of gradients (after all, we have many layers), we need to index the gradients with a key i. This similarity to (key,value) stores, such as the one introduced in Dynamo (DeCandia et al., 2007) is not by coincidence. They,

too, satisfy many similar characteristics, in particular when it comes to distributing the parameters across multiple servers.

- **push(key, value)** sends a particular gradient (the value) from a worker to a common storage. There the parameter is aggregated, e.g. by summing it up.
- **pull(key, value)** retrieves an aggregate parameter from common storage, e.g. after combining the gradients from all workers.

By hiding all the complexity about synchronization behind a simple push and pull operation we can decouple the concerns of the statistical modeler who wants to be able to express optimization in simple terms and the systems engineer who needs to deal with the complexity inherent in distributed synchronization. In the next section we will experiment with such a (key,value) store in practice.

Summary

- Synchronization needs to be highly adaptive to specific network infrastructure and connectivity within a server. This can make a significant difference to the time it takes to synchronize.
- Ring-synchronization can be optimal for P3 and DGX-2 servers. For others possibly not so much.
- A hierarchical synchronization strategy works well when adding multiple parameter servers for increased bandwidth.
- Asynchronous communication (while computation is still ongoing) can improve performance.

Exercises

1. Can you increase the ring synchronization even further? Hint - you can send messages in both directions.
2. Fully asynchronous. Some delays permitted?
3. Fault tolerance. How? What if we lose a server? Is this a problem?
4. Checkpointing
5. Tree aggregation. Can you do it faster?
6. Other reductions (commutative semiring).