计算性能

编译器和解释器

命令式编程:

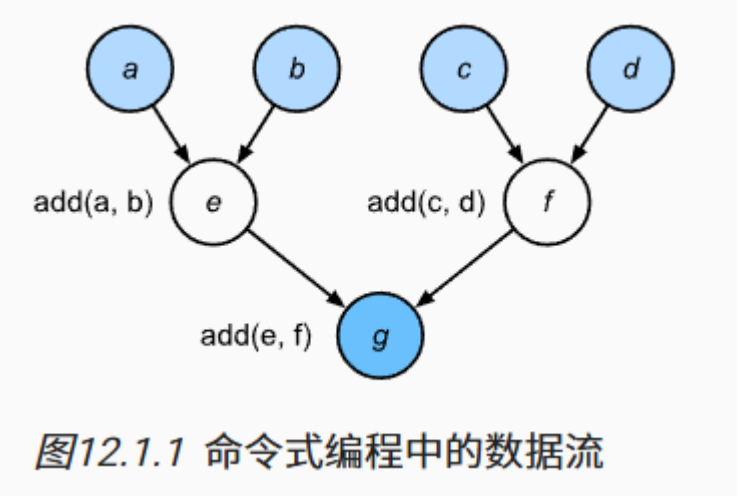


图12.1.1 命令式编程中的数据流

效率不高：分别重复调用函数，开销很大，中间变量储存

符号式编程：

考虑另一种选择*符号式编程*（symbolic programming），即代码通常只在完全定义了过程之后才执行计算。这个策略被多个深度学习框架使用，包括Theano和TensorFlow（后者已经获得了命令式编程的扩展）。一般包括以下步骤：

1. 定义计算流程；
2. 将流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

这将允许进行大量的优化。首先，在大多数情况下，我们可以跳过Python解释器。从而消除因为多个更快的GPU与单个CPU上的单个Python线程搭配使用时产生的性能瓶颈。其次，编译器可以将上述代码优化和重写为`print((1 + 2) + (3 + 4))`甚至`print(10)`。因为编译器在将其转换为机器指令之前可以看到完整的代码，所以这种优化是可以实现的。例如，只要某个变量不再需要，编译器就可以释放内存（或者从不分配内存），或者将代码转换为一个完全等价的片段。下面，我们将通过模拟命令式编程来进一步了解符号式编程的概念。

混合式编程

Pytorch有torchscript:命令式编程进行开发和调试，但是也能够将大多数程序变为符号式编程的程序
python解释器是执行所有层的代码生成指令来给cpu或gpu,瓶颈：单线程解释器，很难让多个cpu均保持忙碌

```
import torch
from torch import nn
from d2l import torch as d2l


# 生产网络的工厂模式
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
                        nn.ReLU(),
```

```
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)

net = torch.jit.script(net)#转换模型
net(x)
```

编译模型的好处：可以将模型和参数序列化后保存下来，允许部署到其他设备

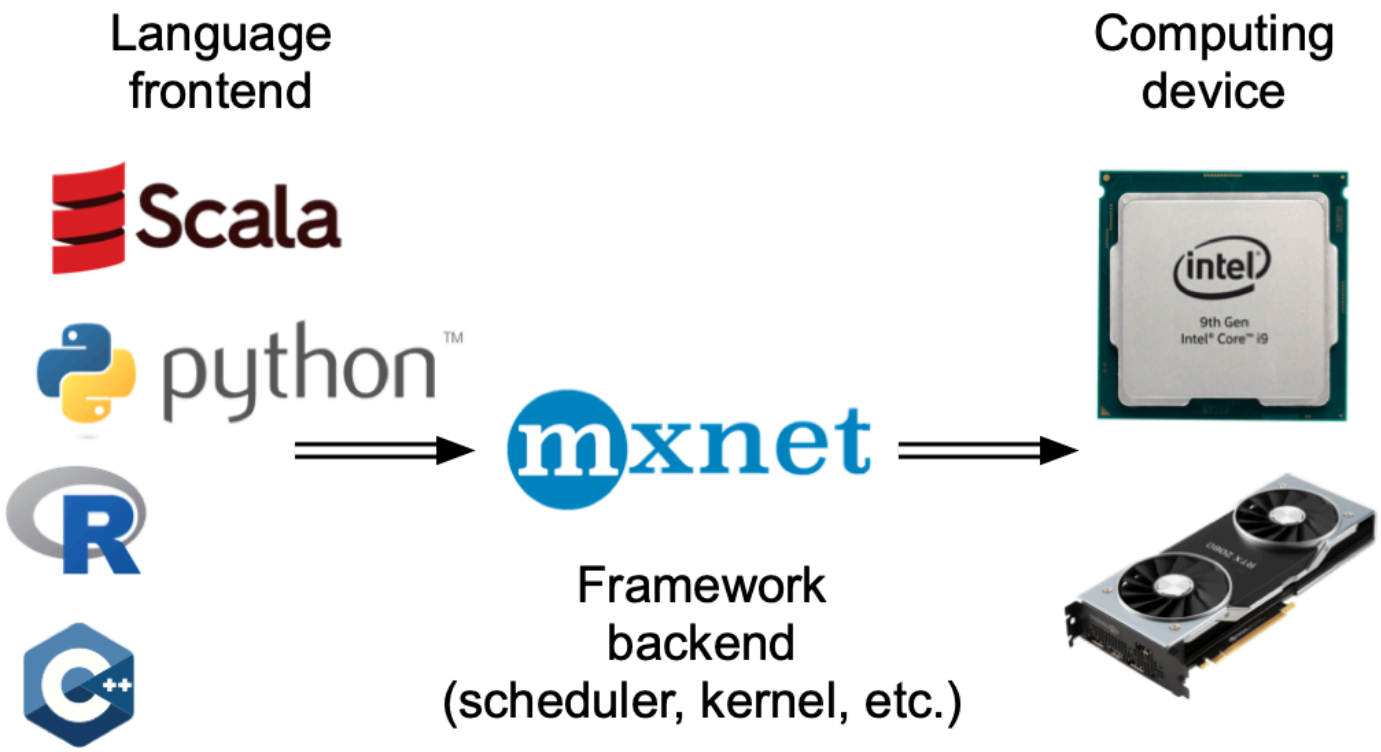
异步计算

PyTorch则使用了Python自己的调度器来实现不同的性能权衡。对PyTorch来说GPU操作在默认情况下是异步的。当调用一个使用GPU的函数时，操作会排队到特定的设备上，但不一定要等到以后才执行。这允许我们并行执行更多的计算，包括在CPU或其他GPU上的操作。

- 导入必要的包

```
import os
import subprocess
import numpy
import torch
from torch import nn
from d2l import torch as d2l
```

广义上说，PyTorch有一个用于与用户直接交互的前端（例如通过Python），还有一个由系统用来执行计算的后端。如 图12.2.1所示，用户可以用各种前端语言编写PyTorch程序，如Python和C++。不管使用的前端编程语言是什么，PyTorch程序的执行主要发生在C++实现的后端。由前端语言发出的操作被传递到后端执行。后端管理自己的线程，这些线程不断收集和执行排队的任务。请注意，要使其工作，后端必须能够跟踪计算图中各个步骤之间的依赖关系。因此，不可能并行化相互依赖的操作。



自动并行

深度学习框架（例如，MxNet、飞桨和PyTorch）会在后端自动构建计算图。利用计算图，系统可以了解所有依赖关系，并且可以选择性地并行执行多个不相互依赖的任务以提高速度。例如，12.2节中的图12.2.2独立初始化两个变量。因此，系统可以选择并行执行它们。

通常情况下单个操作符将使用所有CPU或单个GPU上的所有计算资源。例如，即使在一台机器上有多个CPU处理器，dot操作符也将使用所有CPU上的所有核心（和线程）。这样的行为同样适用于单个GPU。因此，并行化对单设备计算机来说并不是很有用，而并行化对于多个设备就很重要了。虽然并行化通常应用在多个GPU之间，但增加本地CPU以后还将提高少许性能。例如，(Hadjis et al., 2016)则把结合GPU和CPU的训练应用到计算机视觉模型中。借助自动并行化框架的便利性，我们可以依靠几行Python代码实现相同的目标。对自动并行计算的讨论主要集中在使用CPU和GPU的并行计算上，以及计算和通信的并行化内容。

基于gpu的并行计算

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])

run(x_gpu1)
run(x_gpu2)  # 预热设备(对设备进行一次传递，防止缓存影响结果)
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
```

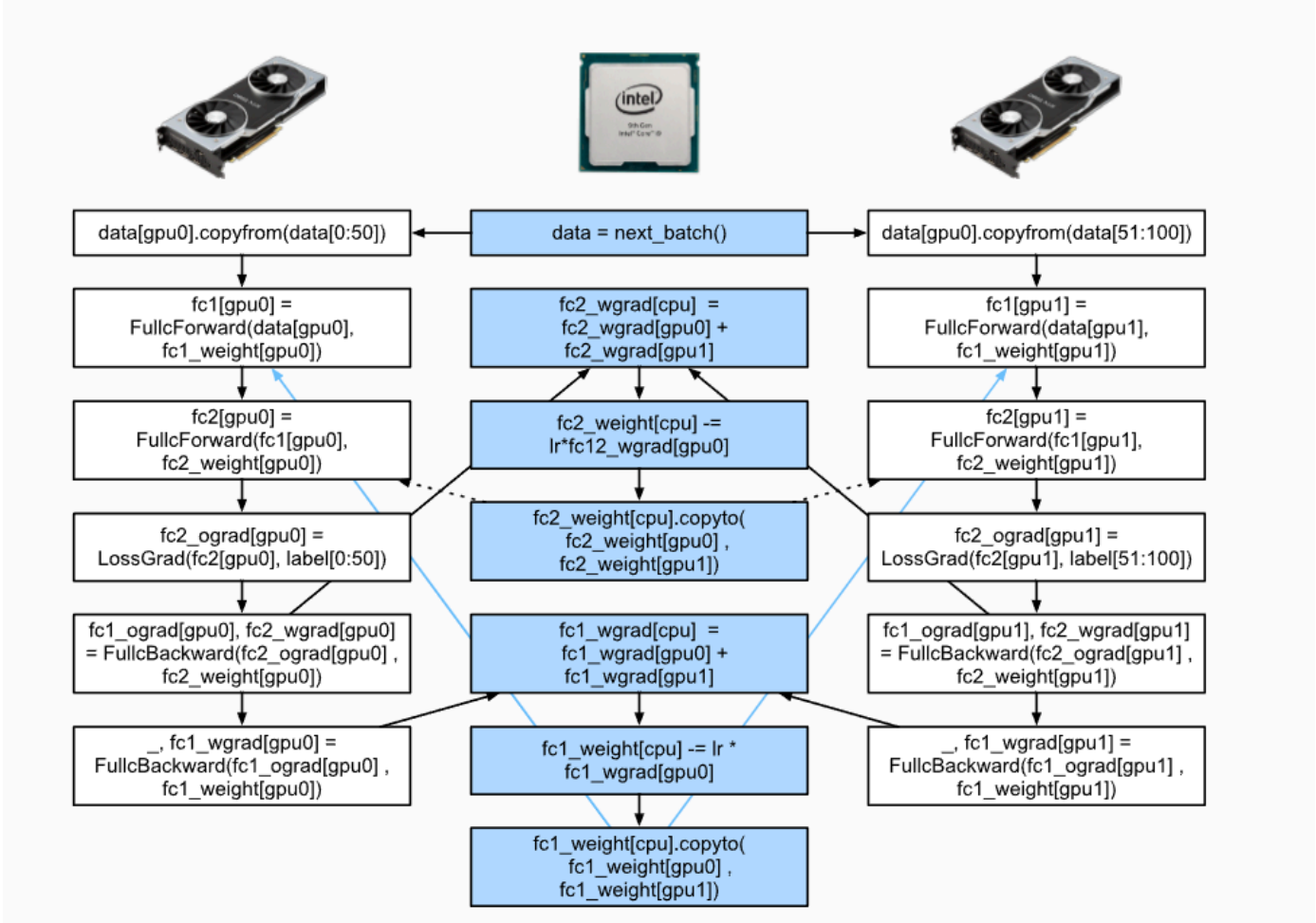
```
    torch.cuda.synchronize(devices[0])#等待一个cuda设备上的所有计算完成

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])

#如果删除两个sunchronize语句就可以实现自动并行计算
```

并行计算与通信

在许多情况下，我们需要在不同的设备之间移动数据，比如在CPU和GPU之间，或者在不同的GPU之间。例如，当执行分布式优化时，就需要移动数据来聚合多个加速卡上的梯度。
请注意，与并行计算的区别是通信操作使用的资源：CPU和GPU之间的总线。事实上，我们可以在两个设备上同时进行计算和通信。如上所述，计算和通信之间存在的依赖关系是必须先计算y[i]，然后才能将其复制到CPU。幸运的是，系统可以在计算y[i]的同时复制y[i-1]，以减少总的运行时间。



硬件

计算机

大多数深度学习研究者和实践者都可以使用一台具有相当数量的内存、计算资源、某种形式的加速器（如一个或者多个GPU）的计算机。计算机由以下关键部件组成：

- 一个处理器（也被称为CPU），它除了能够运行操作系统和许多其他功能之外，还能够执行给定的程序。它通常由8个或更多个核心组成；
- 内存（随机访问存储，RAM）用于存储和检索计算结果，如权重向量和激活参数，以及训练数据；
- 一个或多个以太网连接，速度从1GB/s到100GB/s不等。在高端服务器上可能用到更高级的互连；
- 高速扩展总线（PCIe）用于系统连接一个或多个GPU。服务器最多有8个加速卡，通常以更高级的拓扑方式连接，而桌面系统则有1个或2个加速卡，具体取决于用户的预算和电源负载的大小；
- 持久性存储设备，如磁盘驱动器、固态驱动器，在许多情况下使用高速扩展总线连接。它为系统需要的训练数据和中间检查点需要的存储提供了足够的传输速度。

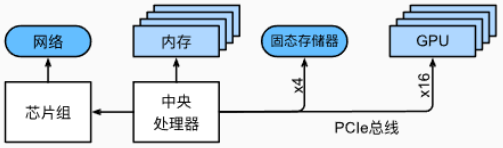


图12.4.2 计算机组件的连接

如 图12.4.2所示，高速扩展总线由直接连接到CPU的多个通道组成，将CPU与大多数组件（网络、GPU和存储）连接在一起。例如，AMD的Threadripper3有64个PCIe4.0通道，每个通道都能够双向传输16Gbit/s的数据。内存直接连接到CPU，总带宽高达100GB/s。

当我们在计算机上运行代码时，需要将数据转移到处理器上（CPU或GPU）执行计算，然后将结果从处理器移回到随机访问存储和持久存储器中。因此，为了获得良好的性能，需要确保每一步工作都能无缝链接，而不希望系统中的任何一部分成为主要的瓶颈。例如，如果不能快速加载图像，那么处理器就无事可做。同样地，如果不能快速移动矩阵到CPU（或GPU）上，那么CPU（或GPU）就会无法全速运行。最后，如果希望在网络上同步多台计算机，那么网络就不应该拖累计算速度。一种选择是通信和计算交错进行。接下来将详细地介绍各个组件。

内存

主要用于存储需要随时访问的数据。
流程：将地址发送到RAM->选择（读取一条64位记录还是一长串记录（突发读取））
!第一次读取的成本是后续读取的500倍，应尽可能使用突发模式写入和读取
GPU要求的带宽更高：处理单元远多于cpu

other

硬盘驱动器（HDD）：便宜，读取延迟高，读写慢
固态驱动器（SSD）：贵，闪存，更快

cpu

中央处理器（central processing unit，CPU）是任何计算机的核心。它们由许多关键组件组成：*处理器核心*（processor cores）用于执行机器代码的；*总线*（bus）用于连接不同组件（注意，总线会因为处理器型号、各代产品和供应商之间的特定拓扑结构有明显不同）；*缓存*（cach）相比主内存实现更高的读取带宽和更低的延迟内存访问。最后，因为高性能线性代数和卷积运算常见于媒体处理和机器学习中，所以几乎所有的现代CPU都包含*向量处理单元*（vector processing unit）为这些计算提供辅助。

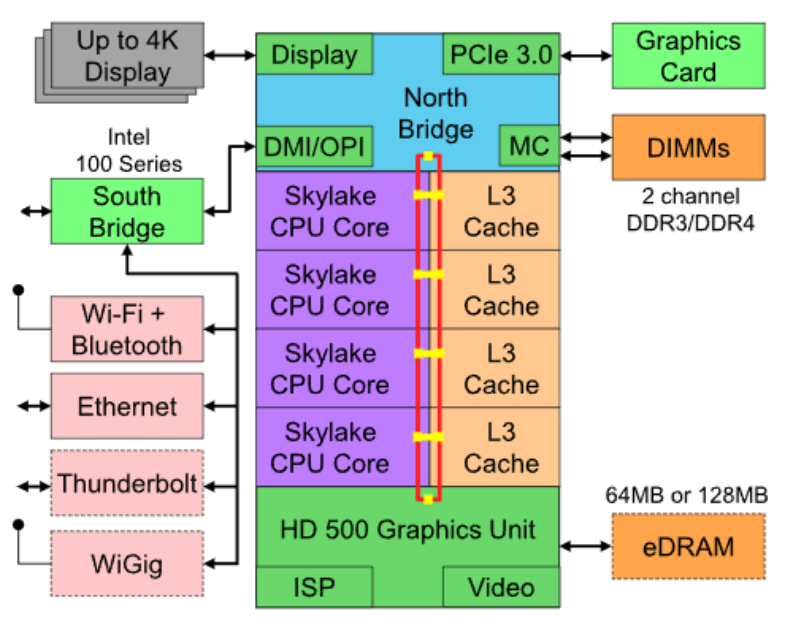


图12.4.3 Intel Skylake消费级四核CPU

微体系结构

前端：加载指令并尝试预测将采用哪条路径
指令：汇编代码->微指令->解码成为更低级的操作再实际处理

缓存

- **寄存器**，严格来说不是缓存的一部分，用于帮助组织指令。也就是说，寄存器是CPU可以以时钟速度访问而没有延迟的存储位置。CPU有几十个寄存器，因此有效地使用寄存器取决于编译器（或程序员）。例如，C语言有一个`register`关键字。
- **一级缓存**是应对高内存带宽要求的第一道防线。一级缓存很小（常见的大小可能是32-64KB），内容通常分为数据和指令。当数据在一级缓存中被找到时，其访问速度非常快，如果没有在那里找到，搜索将沿着缓存层次结构向下寻找。
- **二级缓存**是下一站。根据架构设计和处理器大小的不同，它们可能是独占的也可能是共享的。即它们可能只能由给定的核心访问，或者在多个核心之间共享。二级缓存比一级缓存大（通常每个核心256-512KB），而速度也更慢。此外，我们首先需要检查以确定数据不在一级缓存中，才会访问二级缓存中的内容，这会增加少量的额外延迟。
- **三级缓存**在多个核之间共享，并且可以非常大。AMD的EPYC 3服务器的CPU在多个芯片上拥有高达256MB的高速缓存。更常见的数字在4-8MB范围内。

预测下一步需要哪个存储设备是优化芯片设计的关键参数之一。例如，建议以*向前*的方向遍历内存，因为大多数缓存算法将试图*向前读取*（read forward）而不是向后读取。同样，将内存访问模式保持在本地也是提高性能的一个好方法。

虽然保证处理器core不缺乏数据，但是增加了芯片的尺寸，并且缓存未命中的代价很昂贵（core1需要core2中的数据时，2要暂停，把信息写回内存,1再读取）

GPU和其他加速卡

注意判断：加速卡是为了训练还是推断而优化的

回想一下如 图12.4.5所示的矢量化。处理器核心中添加向量处理单元可以显著提高吞吐量。例如，在 图12.4.5的例子中，我们能够同时执行16个操作。首先，如果我们添加的运算不仅优化了向量运算，而且优化了矩阵运算，会有什么好处？稍后我们将讨论基于这个策略引入的张量核（tensor cores）。第二，如果我们增加更多的核心呢？简而言之，以上就是GPU设计决策中的两种策略。 图12.4.7给出了基本处理块的概述。它包含16个整数单位和16个浮点单位。除此之外，两个张量核加速了与深度学习相关的附加操作的狭窄的子集。每个流式多处理器都由这样的四个块组成。

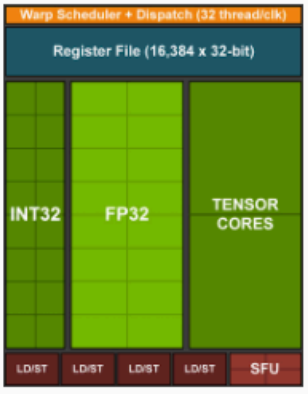


图12.4.7 NVIDIA Turing处理块（图片由英伟达提供）