27-SIMD：如何加速矩阵乘法？

上一讲里呢，我进一步为你讲解了CPU里的“黑科技”，分别是超标量（Superscalar）技术和超长指令字（VLIW）技术。

超标量（Superscalar）技术能够让取指令以及指令译码也并行进行；在编译的过程，超长指令字（VLIW）技术可以搞定指令先后的依赖关系，使得一次可以取一个指令包。

不过，CPU里的各种神奇的优化我们还远远没有说完。这一讲里，我就带你一起来看看，专栏里最后两个提升CPU性能的架构设计。它们分别是，你应该常常听说过的**超线程**（Hyper-Threading）技术，以及可能没有那么熟悉的**单指令多数据流**（SIMD）技术。

超线程：Intel多卖给你的那一倍CPU

不知道你是不是还记得，在第21讲，我给你介绍了Intel是怎么在Pentium 4处理器上遭遇重大失败的。如果不太记得的话，你可以回过头去回顾一下。

那时我和你说过，Pentium 4失败的一个重要原因，就是它的CPU的流水线级数太深了。早期的Pentium 4的流水线深度高达20级，而后期的代号为Prescott的Pentium 4的流水线级数，更是到了31级。超长的流水线，使得之前我们讲的很多解决“冒险”、提升并发的方案都用不上。

因为这些解决“冒险”、提升并发的方案，本质上都是一种**指令级并行**（Instruction-level parallelism，简称IPL）的技术方案。换句话说就是，CPU想要在同一个时间，去并行地执行两条指令。而这两条指令呢，原本在我们的代码里，是有先后顺序的。无论是我们在流水线里面讲到的流水线架构、分支预测以及乱序执行，还是我们在上一讲说的超标量和超长指令字，都是想要通过同一时间执行两条指令，来提升CPU的吞吐率。

然而在Pentium 4这个CPU上，这些方法都可能因为流水线太深，而起不到效果。我之前讲过，更深的流水线意味着同时在流水线里面的指令就多，相互的依赖关系就多。于是，很多时候我们不得不把流水线停顿下来，插入很多NOP操作，来解决这些依赖带来的“冒险”问题。

不知道是不是因为当时面临的竞争太激烈了，为了让Pentium 4的CPU在性能上更有竞争力一点，2002年底，Intel在的3.06GHz主频的Pentium 4 CPU上，第一次引入了**超线程**（Hyper-Threading）技术。

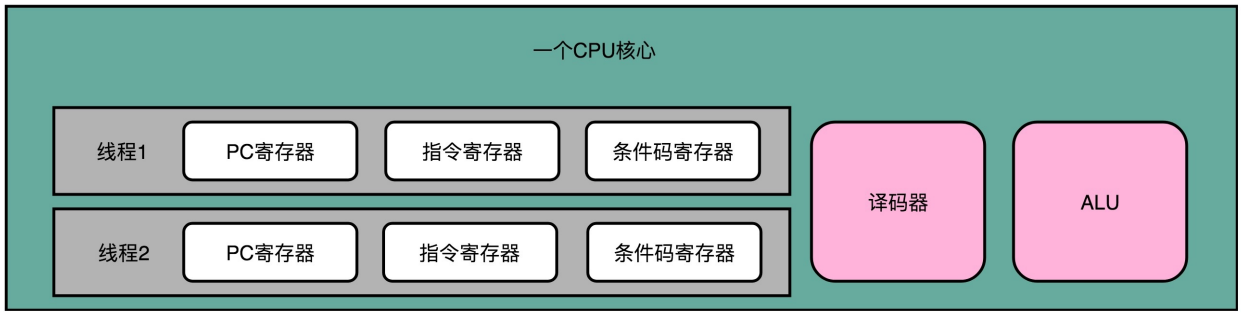
什么是超线程技术呢？Intel想，既然CPU同时运行那些在代码层面有前后依赖关系的指令，会遇到各种冒险问题，我们不如去找一些和这些指令完全独立，没有依赖关系的指令来运行好了。那么，这样的指令哪里来呢？自然同时运行在另外一个程序里了。

你所用的计算机，其实同一个时间可以运行很多个程序。比如，我现在一边在浏览器里写这篇文章，后台同样运行着一个Python脚本程序。而这两个程序，是完全相互独立的。它们两个的指令完全并行运行，而不会产生依赖问题带来的“冒险”。

然而这个时候，你可能就会觉得奇怪了，这么做似乎不需要什么新技术呀。现在我们用的CPU都是多核的，本来就可以用多个不同的CPU核心，去运行不同的任务。即使当时的Pentium 4是单核的，我们的计算机本来也能同时运行多个进程，或者多个线程。这个超线程技术有什么特别的用处呢？

无论是上面说的多个CPU核心运行不同的程序，还是在单个CPU核心里面切换运行不同线程的任务，在同一

时间点上，一个物理的CPU核心只会运行一个线程的指令，所以其实我们并没有真正地做到指令的并行运行。



超线程可不是这样。超线程的CPU，其实是把一个物理层面CPU核心，“伪装”成两个逻辑层面的CPU核心。这个CPU，会在硬件层面增加很多电路，使得我们可以在一个CPU核心内部，维护两个不同线程的指令的状态信息。

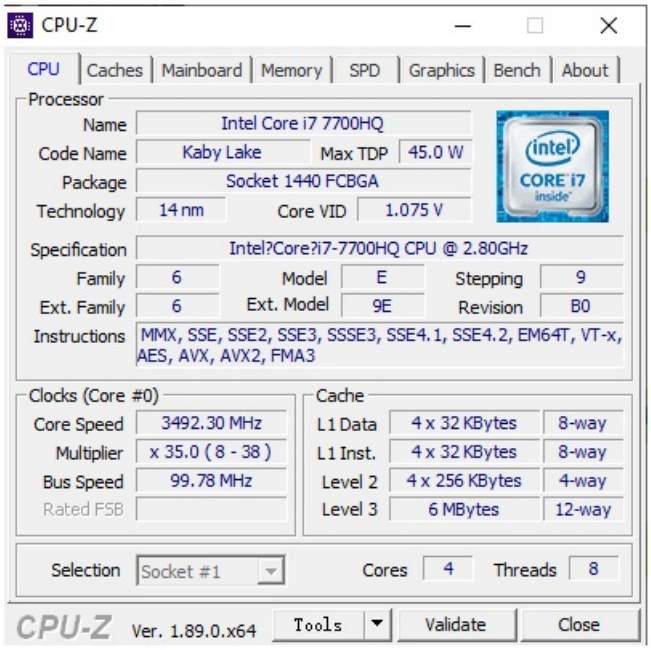
比如，在一个物理CPU核心内部，会有双份的PC寄存器、指令寄存器乃至条件码寄存器。这样，这个CPU核心就可以维护两条并行的指令的状态。在外面看起来，似乎有两个逻辑层面的CPU在同时运行。所以，超线程技术一般也被叫作**同时多线程**（Simultaneous Multi-Threading，简称SMT）技术**。**

不过，在CPU的其他功能组件上，Intel可不会提供双份。无论是指令译码器还是ALU，一个CPU核心仍然只有一份。因为超线程并不是真的去同时运行两个指令，那就真的变成物理多核了。超线程的目的，是在一个线程A的指令，在流水线里停顿的时候，让另外一个线程去执行指令。因为这个时候，CPU的译码器和ALU就空出来了，那么另外一个线程B，就可以拿来干自己需要的事情。这个线程B可没有对于线程A里面指令的关联和依赖。

这样，CPU通过很小的代价，就能实现“同时”运行多个线程的效果。通常我们只要在CPU核心的添加10%左右的逻辑功能，增加可以忽略不计的晶体管数量，就能做到这一点。

不过，你也看到了，我们并没有增加真的功能单元。所以超线程只在特定的应用场景下效果比较好。一般是在那些各个线程“等待”时间比较长的应用场景下。比如，我们需要应对很多请求的数据库应用，就很适合使用超线程。各个指令都要等待访问内存数据，但是并不需要做太多计算。

于是，我们就可以利用好超线程。我们的CPU计算并没有跑满，但是往往当前的指令要停顿在流水线上，等待内存里面的数据返回。这个时候，让CPU里的各个功能单元，去处理另外一个数据库连接的查询请求就是一个很好的应用案例。



我的移动工作站的CPU信息

我这里放了一张我的电脑里运行CPU-Z的截图。你可以看到，在右下角里，我的CPU的Cores，被标明了是4，而Threads，则是8。这说明我手头的这个CPU，只有4个物理的CPU核心，也就是所谓的4核CPU。但是在逻辑层面，它“装作”有8个CPU核心，可以利用超线程技术，来同时运行8条指令。如果你用的是Windows，可以去下载安装一个CPU-Z来看看你手头的CPU里面对应的参数。

SIMD：如何加速矩阵乘法？

在上面的CPU信息的图里面，你会看到，中间有一组信息叫作Instructions，里面写了有MMX、SSE等等。这些信息就是这个CPU所支持的指令集。这里的MMX和SSE的指令集，也就引出了我要给你讲的最后一个提升CPU性能的技术方案，**SIMD**，中文叫作**单指令多数据流**（Single Instruction Multiple Data）。

我们先来体会一下SIMD的性能到底怎么样。下面是两段示例程序，一段呢，是通过循环的方式，给一个list里面的每一个数加1。另一段呢，是实现相同的功能，但是直接调用NumPy这个库的add方法。在统计两段程序的性能的时候，我直接调用了Python里面的timeit的库。

```
$ python
>>> import numpy as np
>>> import timeit
>>> a = list(range(1000))
>>> b = np.array(range(1000))
>>> timeit.timeit("[i + 1 for i in a]", setup="from __main__ import a", number=1000000)
32.82800309999993
>>> timeit.timeit("np.add(1, b)", setup="from __main__ import np, b", number=1000000)
0.9787889999997788
>>>
```

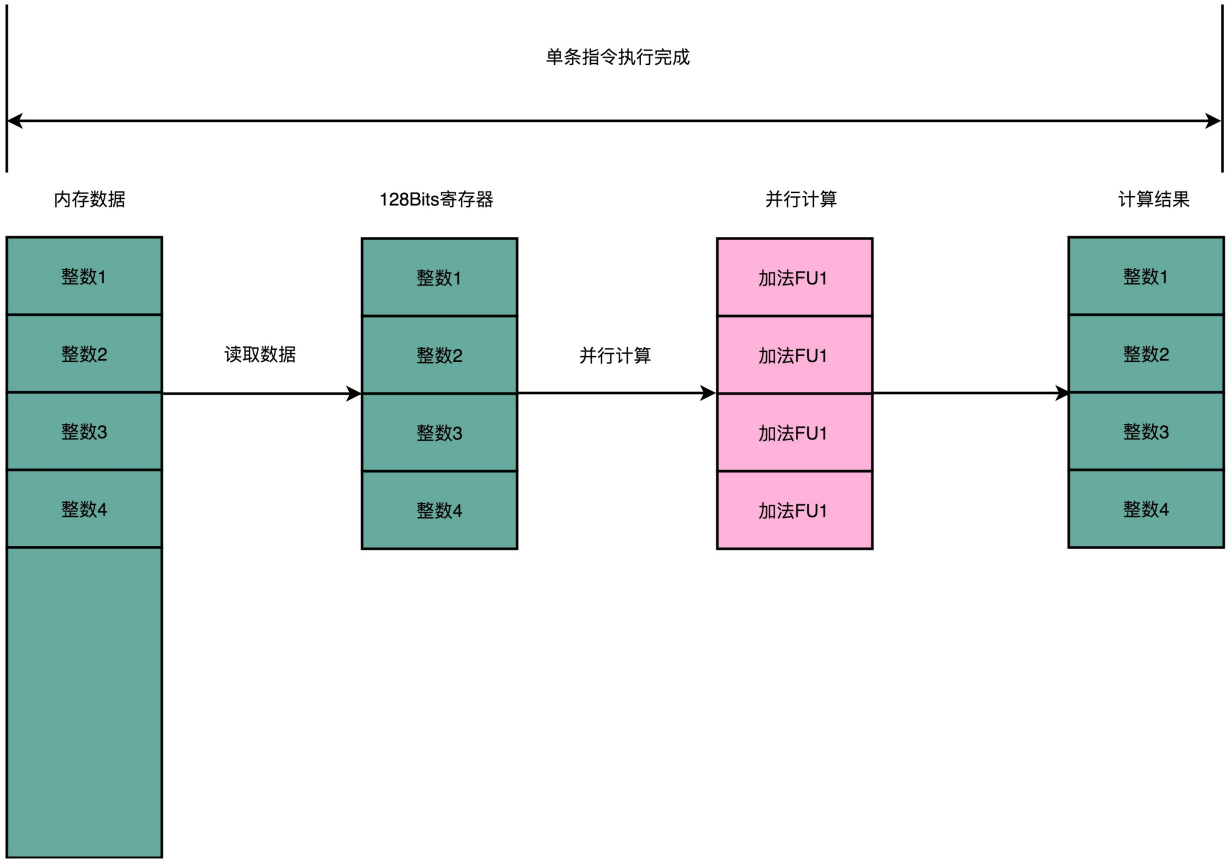
从两段程序的输出结果来看，你会发现，两个功能相同的代码性能有着巨大的差异，足足差出了30多倍。也难怪所有用Python讲解数据科学的教程里，往往在一开始就告诉你不要使用循环，而要把所有的计算都向量化（Vectorize）。

有些同学可能会猜测，是不是因为Python是一门解释性的语言，所以这个性能差异会那么大。第一段程序的循环的每一次操作都需要Python解释器来执行，而第二段的函数调用是一次调用编译好的原生代码，所以才会那么快。如果你这么想，不妨试试直接用C语言实现一下1000个元素的数组里面的每个数加1。你会发现，即使是C语言编译出来的代码，还是远远低于NumPy。原因就是，NumPy直接用到了SIMD指令，能够并行进行向量的操作。

而前面使用循环来一步一步计算的算法呢，一般被称为**SISD**，也就是**单指令单数据**（Single Instruction Single Data）的处理方式。如果你手头的是一个多核CPU呢，那么它同时处理多个指令的方式可以叫作**MIMD**，也就是**多指令多数据**（Multiple Instruction Multiple Dataa）。

为什么SIMD指令能快那么多呢？这是因为，SIMD在获取数据和执行指令的时候，都做到了并行。一方面，在从内存里面读取数据的时候，SIMD是一次性读取多个数据。

就以我们上面的程序为例，数组里面的每一项都是一个integer，也就是需要 4 Bytes的内存空间。Intel在引入SSE指令集的时候，在CPU里面添上了8个 128 Bits的寄存器。128 Bits也就是 16 Bytes ，也就是说，一个寄存器一次性可以加载 4 个整数。比起循环分别读取4次对应的数据，时间就省下来了。



在数据读取到了之后，在指令的执行层面，SIMD也是可以并行进行的。4个整数各自加1，互相之前完全没有依赖，也就没有冒险问题需要处理。只要CPU里有足够多的功能单元，能够同时进行这些计算，这个加法就是4路同时并行的，自然也省下了时间。

所以，对于那些在计算层面存在大量“数据并行”（Data Parallelism）的计算中，使用SIMD是一个很划算的办法。在这个大量的“数据并行”，其实通常就是实践当中的向量运算或者矩阵运算。在实际的程序开发过程中，过去通常是在进行图片、视频、音频的处理。最近几年则通常是在进行各种机器学习算法的计算。

而基于SIMD的向量计算指令，也正是在Intel发布Pentium处理器的时候，被引入的指令集。当时的指令集

叫作**MMX**，也就是Matrix Math eXtensions的缩写，中文名字就是**矩阵数学扩展**。而Pentium处理器，也是CPU第一次有能力进行多媒体处理。这也正是拜SIMD和MMX所赐。

从Pentium时代开始，我们能在电脑上听MP3、看VCD了，而不用专门去买一块“声霸卡”或者“显霸卡”了。没错，在那之前，在电脑上看VCD，是需要专门买能够解码VCD的硬件插到电脑上去的。而到了今天，通过GPU快速发展起来的深度学习技术，也一样受益于SIMD这样的指令级并行方案，在后面讲解GPU的时候，我们还会遇到它。

总结延伸

这一讲，我们讲完了超线程和SIMD这两个CPU的“并行计算”方案。超线程，其实是一个“线程级并行”的解决方案。它通过让一个物理CPU核心，“装作”两个逻辑层面的CPU核心，使得CPU可以同时运行两个不同线程的指令。虽然，这样的运行仍然有着种种的限制，很多场景下超线程并不一定能带来CPU的性能提升。但是Intel通过超线程，让使用者有了“占到便宜”的感觉。同样的4核心的CPU，在有些情况下能够发挥出8核心CPU的作用。而超线程在今天，也已经成为Intel CPU的标配了。

而SIMD技术，则是一种“指令级并行”的加速方案，或者我们可以说，它是一种“数据并行”的加速方案。在处理向量计算的情况下，同一个向量的不同维度之间的计算是相互独立的。而我们的CPU里的寄存器，又能放得下多条数据。于是，我们可以一次性取出多条数据，交给CPU并行计算。

正是SIMD技术的出现，使得我们在Pentium时代的个人PC，开始有了多媒体运算的能力。可以说，Intel的MMX、SSE指令集，和微软的Windows 95这样的图形界面操作系统，推动了PC快速进入家庭的历史进程。

推荐阅读

如果你想看一看Intel CPU里面的SIMD指令具体长什么样，可以去读一读《计算机组成与设计：硬件/软件接口》的3.7章节。

课后思考

最后，给你留一道思考题。超线程这样的技术，在什么样的应用场景下最高效？你在自己开发系统的过程中，是否遇到超线程技术为程序带来性能提升的情况呢？

欢迎留言和我分享你的疑惑和见解。你也可以把今天的内容，分享给你的朋友，和他一起学习和进步。

精选留言：

- 易儿易 2019-06-27 20:20:02
  终于知道为什么挖矿烧显卡啦 ~

- 陆离 2019-06-26 15:19:45
  老师这个从超线程技术是不是可以和各种语言中的多线程概念联系起来？
  看起来像是多个线程在运行，其实这是当流水线停顿的时候执行另一个线程的指令，这个是经常说的时间片是什么关系？
  那线程的阻塞，唤醒操作又是如何实现的呢？

- magicnum 2019-06-26 12:03:45
  I/O密集型单不是CPU密集型的场景下超线程效率高。数据库连接池、定制线程池处理I/O读写

- Destroy、 2019-06-26 11:36:38
  老师超线程，是不是有点像python的协程？

- pebble 2019-06-26 09:13:43
  MMX指令是多媒体扩展指令吧，最早是为多媒体引入的

- Linuxer 2019-06-26 08:33:58
  这里有个问题请教，之前做性能监控由于超线程的存在，一般看负载和cpu利用率会按照 CPU数*核数*线程数，通过今天的课程来看，好像不能这么看了?

- null 2019-06-26 08:30:04
  simd只是用来加速向量么？有没有其他方面可以优化代码的呀？感觉学了很有帮助。。