第十五章　并行计算

15.1　简介

15.1 简 介

- 并行计算是一种通过同时处理多个计算任务来提高计算效率的方式, 通过同时处理多个任务来提高整体计算速度, 减少计算时间, 同时处理的多个计算任务, 可以在不同处理单元上并行执行. 并行计算广泛应用于大规模数据处理、复杂模型训练、科学计算等领域. 在线学习注重模型持续适应新数据的能力, 而并行计算注重通过同时处理多个任务来提高计算效率.
- 随着数据的增加, 机器学习中的计算瓶颈越发凸显, 并行计算就成为解决这个瓶颈的关键技术. 根据计算设备的不同, 一般分为基于 CPU 的并行计算和基于 GPU 的并行计算, 本章基于 Python 和 R 编程环境, 给出了基于CPU 和基于 GPU 的并行计算的相关概念、原理和计算流程.

15.2　并行计算相关概念

15.2.1 进程

- 进程的概念是 60 年代初首先由麻省理工学院的 MULTICS 系统和 IBM公司的 CTSS/360 系统引入的. **进程** (process) 是具有一定独立功能的程序在某个数据集合上的一次运行活动, 是系统进行资源分配和调度的一个独立单位. 程序只是一组指令的有序集合, 它本身没有任何运行的含义, 只是一个静态实体. 而进程则不同, 它是程序在某个数据集上的执行, 是一个动态实体. 它因创建而产生, 因调度而运行, 因等待资源或事件而被处于等待状态,因完成任务而被撤销, 反映了一个程序在一定的数据集上运行的全部动态过程. 图 15.1 为 Windows 任务管理器中进程一览表, 每个进程表现为一个独立的执行程序.

15.2.1 进程

1. 进程的特征

- 根据进程的特点, 它具有以下特征：
  - (1) 动态性：进程的实质是程序在多道程序系统中的一次执行过程, 进程是动态产生, 动态消亡的.
  - (2) 并发性：任何进程都可以同其他进程一起并发执行.
  - (3) 独立性：进程是一个能独立运行的基本单位, 同时也是系统分配资源和调度的独立单位.
  - (4) 异步性：由于进程间的相互制约, 使进程具有执行的间断性, 即进程按各自独立的、不可预知的速度向前推进.
  - (5) 结构特征：进程由程序、数据和进程控制块三部分组成.
- 多个不同的进程可以包含相同的程序：一个程序在不同的数据集里就构成不同的进程, 能得到不同的结果; 但是执行过程中, 程序不能发生改变.

2. 进程的状态

- 进程执行时的间断性, 决定了进程可能具有多种状态. 事实上, 运行中的进程可能具有如图 15.2 所示的三种基本状态：运行态、就绪态和阻塞态.

15.2.1 进程

- (1) 就绪态 (ready)

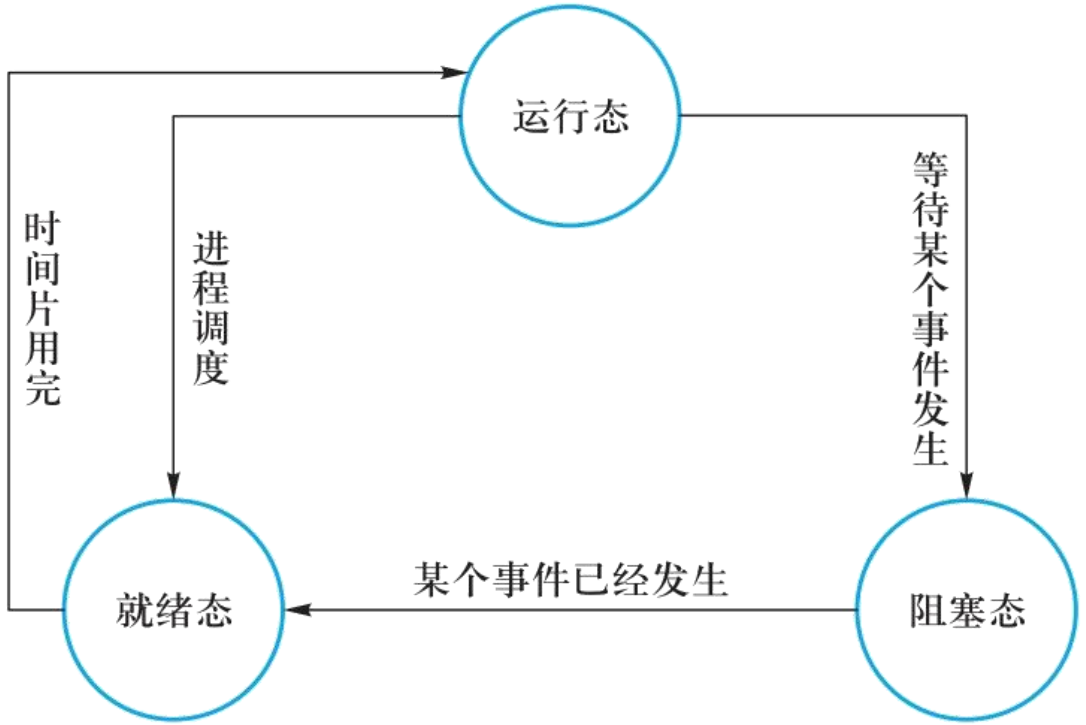
  进程已获得除处理器外的所需资源, 等待分配处理器资源; 只要分配了处理器进程就可执行. 就绪进程可以按多个优先级来划分队列. 例如, 当一个进程由于时间片用完而进入就绪状态时, 排入低优先级队列; 当进程由 I/O操作完成而进入就绪状态时, 排入高优先级队列.

- (2) 运行态 (running)

  进程占用处理器资源, 处于此状态的进程的数目小于等于处理器的数目.在没有其他进程可以执行时 (如所有进程都在阻塞状态), 通常会自动执行系统的空闲进程.

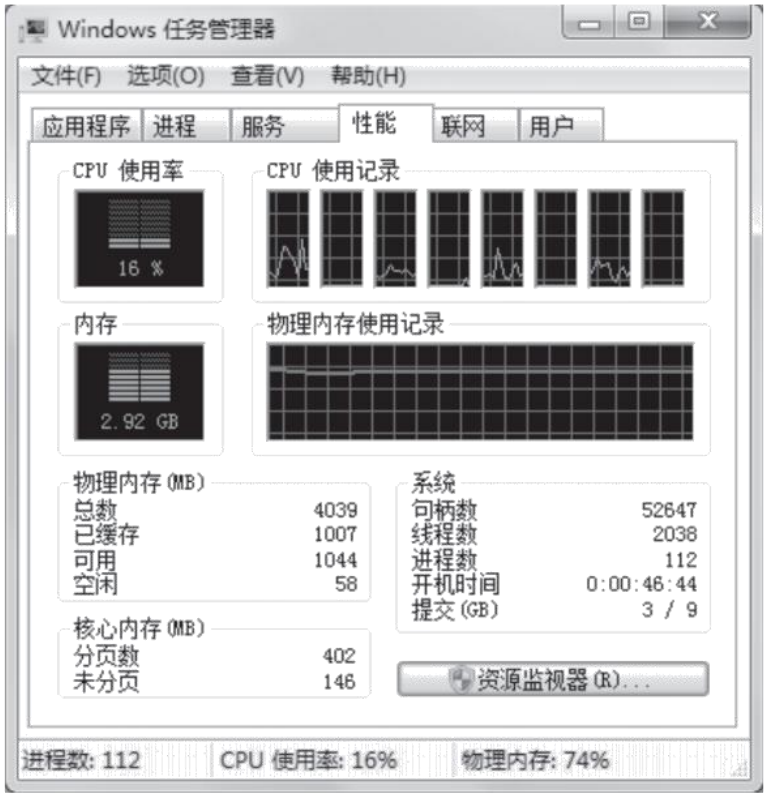
- (3) 阻塞态 (blocked)

  由于进程等待某种条件 (如 I/O 操作或进程同步), 在条件满足之前无法继续执行. 该事件发生前即使把处理器分配给该进程, 也无法运行.



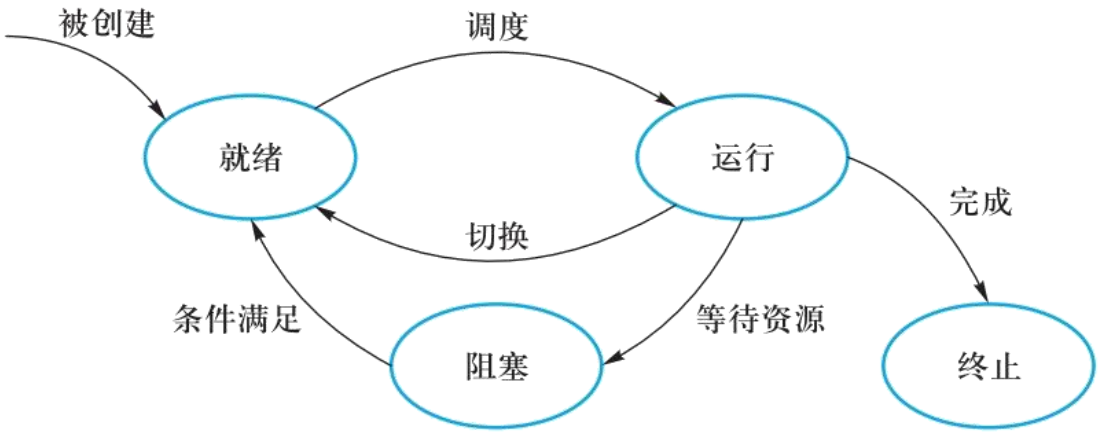
15.2.2 线程

- **线程** (thread) 是进程的一个实体, 是 CPU 调度和分派的基本单位. 线程不能够独立执行, 必须依存在进程中, 由进程提供多个线程执行控制. 从内核角度讲线程是活动体对象, 而进程只是一组静态的对象集, 进程必须至少拥有一个活动线程才能维持运转. 图 15.3 表明一个进程最多可以调用 8 个线程.
- 线程和进程的关系是: 线程是属于进程的, 线程运行在进程空间内, 同一进程所产生的线程共享同一内存空间, 当进程退出时该进程所产生的线程都会被强制退出并清除. 线程可与属于同一进程的其他线程共享进程所拥有的全部资源, 但是其本身基本上不拥有系统资源, 只拥有一点在运行中必不可少的信息 (如程序计数器、一组寄存器和栈).
- 在操作系统中引入线程带来的主要好处是:
  - (1) 在进程内创建、终止线程比创建、终止进程要快.
  - (2) 同一进程内的线程间切换比进程间的切换要快, 尤其是用户级线程间的切换.



15.2.2 线程

- 另外, 线程的出现还因为以下几个原因:
  - (1) 并发程序的并发执行, 在多处理环境下更为有效. 一个并发程序可以建立一个进程, 而这个并发程序中的若干并发程序段就可以分别建立若干线程, 使这些线程在不同的处理器上执行.
  - (2) 每个进程具有独立的地址空间, 而该进程内的所有线程共享该地址空间. 这样可以解决父子进程模型中, 子进程必须复制父进程地址空间的问题.
  - (3) 线程对解决客户/服务器模型非常有效.
- 不同的平台对线程的状态定义不同, 大致可以定义为运行、挂起、睡眠、阻塞、就绪、终止这六种, 如图 15.4 所示.



15.2.2 线程

- 运行：就是线程获得了 CPU 的控制权, 正在执行计算.
- 挂起：一般是指被挂起, 因为同一时刻, 需要“同步”运行的线程不止它一个, 所以基于时间片轮转的原则, 它在独占了一段时间的 CPU 后, 被挂起,线程环境被压栈.
- 睡眠：一般是指主动挂起, 这种情况在 Windows 平台不存在.
- 阻塞：与挂起和睡眠类似, 都是失去 CPU 的控制权. 与挂起更相像, 也是被挂起的. 不同之处在于, 被挂起的线程没有额外的表示, 而被阻塞的线程会被记录下来, 当等待的因素就绪后, 线程会转为就绪状态. 例如你在线程中调用一些系统服务函数, 会引起线程控制权的一次裁决, 从而挂起本线程,造成本线程的阻塞. 挂起、睡眠、阻塞看起来差不多, 但本质上还是有所区别的.
- 就绪：顾名思义, 就是指它准备好了, 一旦轮到它, 它就可以转为运行状态.
- 终止：线程结束.

15.2.3 并行计算与分布式计算

- **并行计算** (parallel computing) 是指同时使用多种计算资源解决计算问题的过程, 是提高计算机系统计算速度和处理能力的一种有效手段. 它的基本思想是用多个处理器 (多个核) 来协同求解同一问题, 即将被求解的问题分解成若干个部分, 各部分均由一个独立的处理器来并行计算. 并行计算系统既可以是专门设计的、含有多个处理器的超级计算机, 也可以是以某种方式互连的若干台的独立计算机构成的集群. 通过并行计算集群完成数据的处理,再将处理的结果返回给用户.
- 并行计算可分为时间上的并行和空间上的并行. 时间上的并行是指流水线技术. 而空间上的并行是指多个处理器并发的执行计算, 即通过网络将两个以上的处理器连接起来, 达到同时计算同一个任务的不同部分, 或者单个处理器无法解决的大型问题. 但这个空间一般是指集中放在一起的集群计算机. 并行计算中主要研究的是空间上的并行问题. 从程序和算法设计人员的角度来看, 并行计算又可分为数据并行和任务并行. 空间上的并行导致了两类并行机的产生, 按照弗林 (Flynn) 分类法为：单指令流多数据流 (SIMD)和多指令流多数据流 (MIMD). 我们常用的串行机也叫做单指令流单数据流(SISD).

15.2.3 并行计算与分布式计算

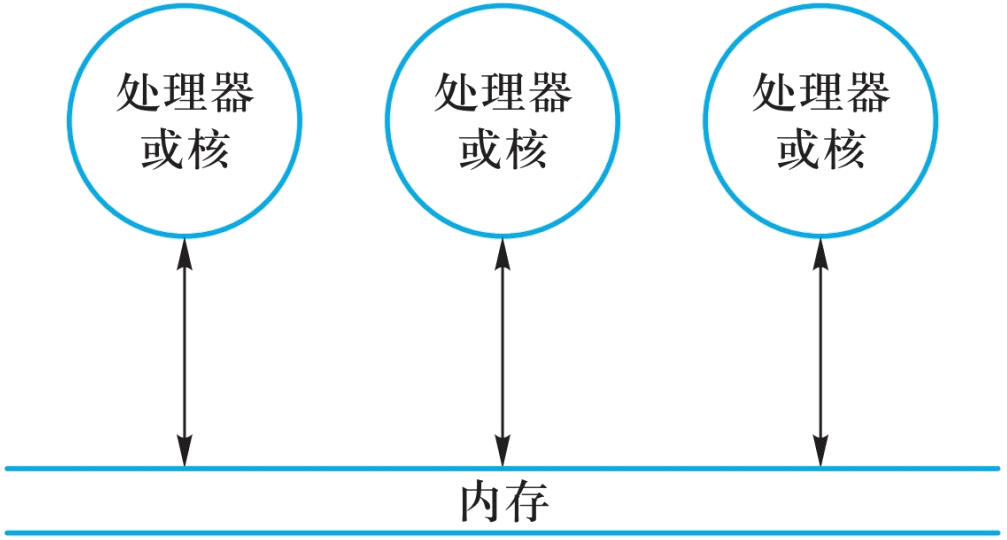
- 而**分布式计算** (distributed computing) 也是一种并行计算, 主要是指通过网络相互连接的两个以上的处理器相互协调, 各自执行相互依赖的不同应用, 从而达到协调资源访问, 提高资源使用效率的目的. 但是, 它无法达到并行计算所倡导的提高求解同一个应用的速度, 或者提高求解同一个应用的问题规模的目的. 对于一些复杂应用系统, 分布式计算和并行计算通常相互配合, 既要通过分布式计算协调不同应用之间的关系, 又要通过并行计算提高求解单个应用的能力.
- 因此, 并行计算一般在企业内部进行, 而分布式计算可能会跨越局域网,或者直接部署在互联网上, 节点之间几乎不互相通信. 很多公益性的项目, 就是使用分布式计算的方式在互联网上实现, 比如以寻找外星人为目的的 SETI项目.

15.2.4 同步与异步

- **同步** (synchronization)：进程之间的关系不是相互排斥临界资源的关系,而是相互依赖的关系. 进一步的说明：就是前一个进程的输出作为后一个进程的输入, 当第一个进程没有输出时第二个进程必须等待. 具有同步关系的一组并发进程相互发送的信息称为消息或事件.
- **异步** (asynchronization)：异步和同步是相对的一个概念, 同步就是顺序执行, 执行完一个再执行下一个, 需要等待、协调运行. 异步就是彼此独立,在等待某事件的过程中继续做自己的事, 不需要等待这一事件完成后再工作.线程就是实现异步的一个方式. 异步让调用方法的主线程不需要同步等待另一线程的完成, 从而可以让主线程干其他的事情.

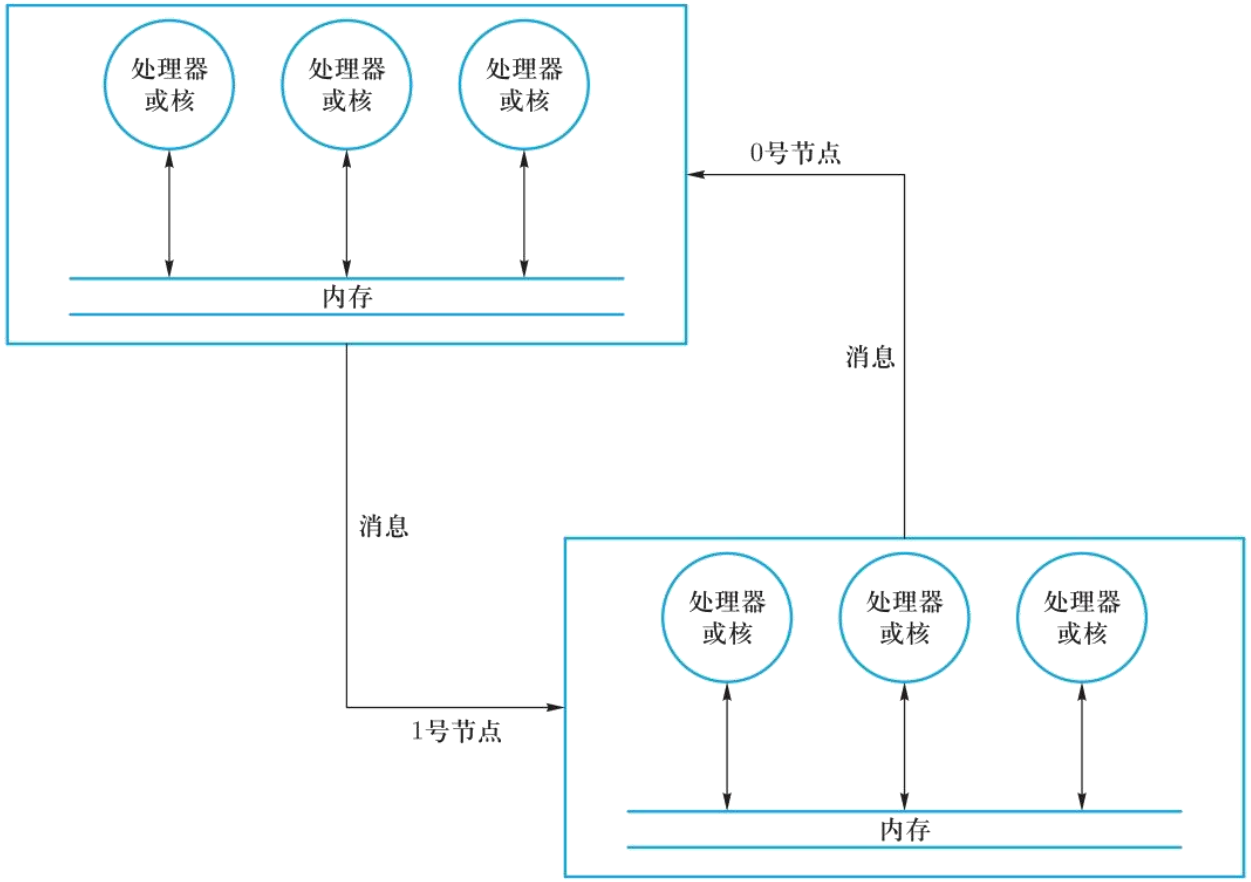
15.2.5 通信

- 在并行计算中, 通信是指进程或线程之间的数据传输或内存访问. 一般分为两种：共享内存和消息传递.
- 如图 15.5, 共享内存这种方式在多核并行计算中比较常见, 通常会设置一个共享变量, 然后多个线程去操作同一个共享变量, 从而达到线程通信的目的. 这种通信模式中, 不同的线程之间是没有直接联系的. 都是通过共享变量这个“中间人”来进行交互. 而这个“中间人”必要情况下还需被保护在临界区内 (加锁或同步). 由此可见, 一旦共享变量变得多起来, 并且涉及多种不同线程对象的交互, 这种管理会变得非常复杂, 极容易出现数据竞争、死锁等问题.



15.2.5 通信

- 而消息传递方式主要针对多处理器或多节点并行计算, 采取的是进程或线程之间的直接通信, 不同的进程或线程之间通过显式的发送消息来达到交互目的. 如图 15.6, 消息传递模型有以下特征:
  - (1) 计算时任务集可以用它们自己的内存. 多任务可以在相同的物理处理器上, 同时可以访问任意数量的处理器.
  - (2) 任务之间通过接收和发送消息来进行数据通信.
  - (3) 数据传输通常需要每个处理器协调操作来完成. 例如, 发送操作有一个接受操作来配合.



15.2.5 通信

- 表 15.1 总结了这两种通信模式的异同情况.

并发模型	通信机制	同步机制
共享内存	线程之间共享程序的公共状态, 线程之间通过写–读内存中的公共状态隐式进行通信.	同步是显式进行的. 程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行.
消息传递	线程之间没有公共状态, 线程之间必须通过明确的发送消息显式进行通信.	由于消息的发送必须在消息的接收之前, 因此同步是隐式进行的.

15.2.6 加速比

- 为了更方便地描述并行计算的性能, 一般采用加速比指标来进行度量.**加速比**定义为

$$S = \frac{T_s}{T_p}, \tag{15.2.1}$$

  - 其中, T_s 表示单处理器上最优串行化算法计算的时间, T_p 表示使用 p 个 CPU处理器并行计算的时间.

- 加速比 $S<1$ 意味着并行计算的时间比串行计算的时间还长, 并行计算效率反而降低; $S<p$, 表示次线性加速; $S \approx p$, 表示线性加速; $S>p$, 表示超线性加速. 一般来说, 加速比通常都小于 CPU 核数, 只有极少数并行算法可以获得超线性加速比, 例如并行搜索工作量少于串行搜索工作量等算法, 另外, 也有可能是由于高速缓存产生的额外加速效果所导致. 因此, 在并行算法设计中, 其加速比一般要求向 CPU 核数靠近, 加速比越接近线性加速, 程序性能就越好.

- 但是, 对于某些串行程序, 并不是所有部分都可以用并行程序替代, 有一部分必须要串行执行. 令 W_s 为程序中的串行部分, W_p 为程序中的并行部分,则 $W = W_s + W_p$.

15.2.6 加速比

- 根据阿姆达尔 (Amdahl) 定律, 在计算规模一定的情况下,加速比定义为

$$S = \frac{W}{W_s + W_p / p}, \tag{15.2.2}$$

  - 串行部分比例为 $f = \frac{W_s}{W}$, 则式 (15.2.2) 为

$$S = \frac{1}{f + (1 - f) / p}. \tag{15.2.3}$$

- 随着 CPU 核数 p 的增加, 这是一个递增的函数, 但这个函数有上限, 当$p \to \infty$ 时, 有

$$S = \frac{1}{f}.$$

- 图 15.7 为串行程序比例分别是 0.0, 0.1, …, 0.2 时的加速比随 CPU 核数p 变化 (这里核数取 2、4、8、16、24、48) 的情形.

15.2.6 加速比

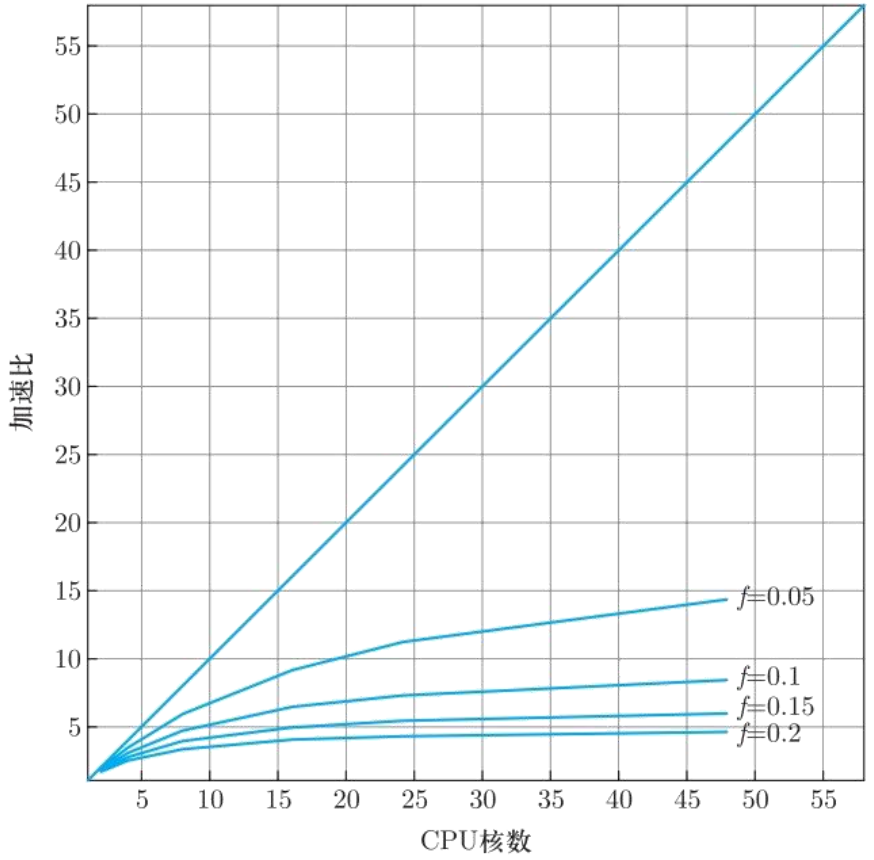
- 从图中可以看出, 当一定的情况下, 加速比随核数的增加而增加, 但是在 16 核之前增速较快, 之后增速较为缓慢; 当 CPU 核数一定时, 加速比随着串行程序占比的减少而增加. 因此, 要想提高并行计算效率, 需要从 CPU 核数和串行程序占比两个因素综合考虑.

- 若考虑并行计算时的通信、同步和归约等操作所花费的额外开销 W_0, 则

$$S = \frac{W}{W_s + W_p / p + W_0}, \tag{15.2.4}$$

$$S = \frac{1}{f + (1-f)/p + W_0 / W}. \tag{15.2.5}$$

- 当并行计算时的通信、同步和归约等操作所花费的额外开销比重较大时, 并行效率降低, 严重时加速比小于 1. 在后面例子将遇到这种情况.



15.2.6 加速比

- 对于很多大型计算, 精度要求高, 而计算时间要求不能增加. 在这种计算规模增加的情况下, 要想保持原有计算时间, 必须增加处理器才能完成计算任务.Gustafson[157] 提出了变问题规模的加速比模型如下:

$$S = \frac{f + (1-f)p}{1 + W_0 / W}. \tag{15.2.6}$$

- 从式 (15.2.6) 可以看出, 当处理器个数增加时, 必须控制额外开销的增加, 才能达到线性加速. 因此, 在并行计算中, 如何优化程序, 是一个值得思考的问题. 考虑到加速比计算的简便性, 在本书中采用加速比公式 (15.2.1).

15.3　基于 CPU 线程的并行计算

15.3 基于 CPU 线程的并行计算

- 在 Python 中, 提供了基于线程的并行模块 threading, threading 模块除了 Thread 类之外, 还包括其他很多的同步机制, 这些同步机制可以避免数据竞争问题的发生.

15.3.1 创建线程

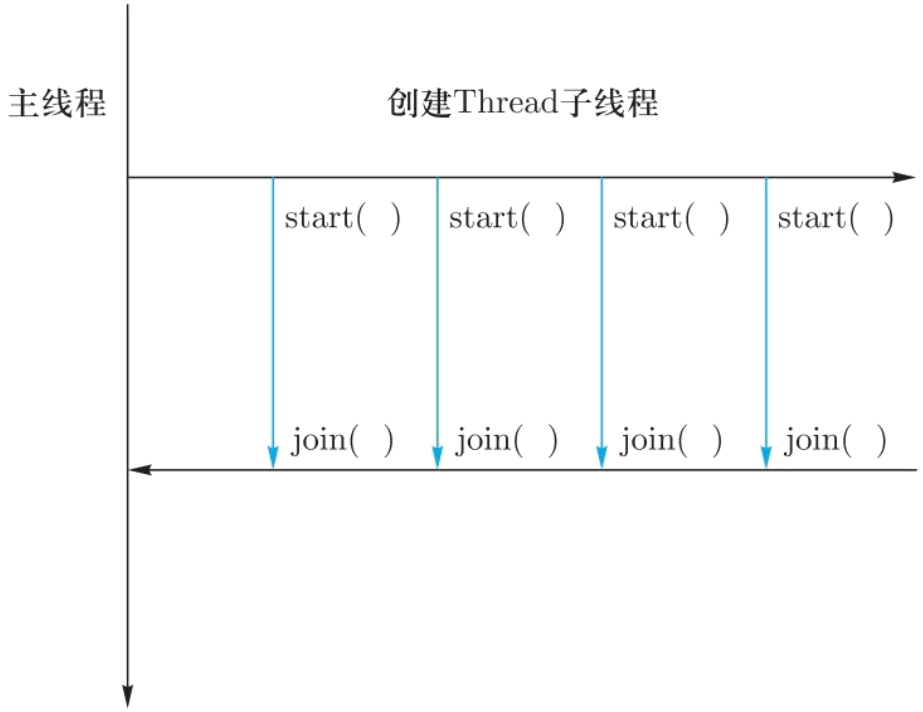
- 使用 Thread 类主要有两种方法创建线程：
  - (1) 创建 Thread 类的实例, 传递一个函数;
  - (2) 派生 Thread 类的子类, 并创建子类的实例.
- 本教材主要采用第一种方式创建线程. 对于 Thread 类, 其主要属性和方法见表 15.2.

属性/方法	描述
Thread 类属性	
name	线程名
ident	线程的标识符
daemon	布尔值, 表示这个线程是不是守护线程
Thread 类方法	
init(group,...)	实例化一个线程对象, 需要一个可调用的 target 对象, 以及参数 args 或者 kwargs. 还可以传递 name 和 group 参数. daemon 的值将会设定 thread.daemon 的属性.
start()	线程启动函数. 开始执行该线程
run()	定义线程的方法. (通常在子类中重写)
join(timeout=None)	等待函数. 直至启动的线程终止之前一直挂起, 除非给出了 timeout(单位秒), 否则一直被阻塞.

15.3.1 创建线程

- 注意：守护线程一般是一个等待客户端请求的服务器. 如果没有客户端请求, 守护线程就是空闲的. 如果把一个线程设置为守护线程, 就表示这个线程是不重要的, 进程退出时不需要等待这个线程执行完成. 使用下面的语句：thread.daemon=True 可以将一个线程设置为守护线程, 同样的也可以通过这个值来查看线程的守护状态. 对于主线程, 将在所有的非守护线程退出之后才退出.

- 如图 15.8, 当线程对象被创建, 其活动可通过调用线程的 start() 方法开始. 一旦线程活动开始, 该线程会被启动. 主线程可以调用一个线程的 join()方法, 这会阻塞调用该方法的线程, 直到被调用 join() 方法的线程终结.



15.3.1 创建线程

- 下面给出具体的 Thread 类构造函数：class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
- 调用这个构造函数时, 必须带有关键字参数. 参数如下：
  - (1) group 应该为 None, 为了日后扩展 ThreadGroup 类实现而保留.
  - (2) target 是用于 run() 方法调用的可调用对象, 通常是启动一个线程活动时要执行的目标函数. 默认是 None, 表示不需要调用任何方法.
  - (3) name 是线程名称. 默认情况下, 由“Thread-N”格式构成一个唯一的名称, 其中 N 是小的十进制数.
  - (4) args 是用于调用目标函数的参数元组. 默认是 ().
  - (5) kwargs 是用于调用目标函数的关键字参数字典, 默认是 {}.
  - (6) daemon 如果不是 None, 将显式地设置该线程是否为守护模式. 如果是 None (默认值), 线程将继承当前线程的守护模式属性.

15.3.1 创建线程

例 15.1 定义线程函数 HelloWorld, 其参数为自定义的线程号 (从 0 开始), 其内容为延迟 1 秒后, 输出形如 Hello world！thread id:0 这样的字符串.使用 Thread 创建当前计算机最大 CPU 或核数的线程数, 并用 start 函数启动该线程, 同时使用计时函数来了解使用 join 函数的作用.

```
import threading
from time import sleep
from time import perf_counter
import multiprocessing
N_core = multiprocessing.cpu_count()#获取当前计算机最大CPU或核数
def HelloWorld(thread_id):#定义线程函数
    sleep(1)#延迟1秒
    print('Hello world! thread id:{} \n'.format(thread_id))
    return
if __name__ == "__main__":
    start = perf_counter()#计时开始
    for i in range(N_core):
        t = threading.Thread(target=HelloWorld,args=(i,))#创建子线程
        t.start()#启动线程
        t.join()#阻塞
    end = perf_counter()#计时结束
    print('运行时间为: %s Seconds'%(end-start))
```

15.3.1 创建线程

- 输出结果为：

```
 Hello world! thread id:2
 Hello world! thread id:1
 Hello world! thread id:0
 Hello world! thread id:5
 Hello world! thread id:6
 Hello world! thread id:4
 Hello world! thread id:3
 Hello world! thread id:7
运行时间为：0.002644299998792121 Seconds
```

- 在主线程中创建了 8 个子线程, 每个子线程都调用了线程函数 HelloWorld,计时函数对主线程创建和启动子线程进行计时, 由于不需要等待子线程结束,所以先输出了主线程运行时间, 然后乱序输出各子线程的输出内容. 如果在程序中加入 t.join() 等待函数, 输出结果如下：

```
Hello world! thread id:0
Hello world! thread id:1
Hello world! thread id:2
Hello world! thread id:3
Hello world! thread id:4
Hello world! thread id:5
Hello world! thread id:6
Hello world! thread id:7
运行时间为：8.120153000000073 Seconds
```

15.3.1 创建线程

- 很明显, 主线程等子线程一个个执行完, 再输出运行时间, 该时间包含了等待时间. 从时间上看与串行计算无异, 也就是说并没有真正按照图 15.8 运行. 因此, 对于 join 函数, 多数情况下根本不需要调用它. 一旦线程启动, 就会一直执行, 直到给定的函数完成后退出. 如果主线程还有其他事情要做 (并不需要等待这些线程完成), 可以不调用 join 函数. join 函数只有在你需要等待线程完成时候才是有用的.

15.3.2 同步

- 在多线程程序中, 一般有一些特定的函数或代码块不希望被多个线程同时执行, 这就需要使用同步了. threading 模块的同步机制, 如表 15.3 所示, 主要有锁机制、事件、信号量、栅栏.

对象	描述
Lock	锁对象
RLock	递归锁, 是一个线程可以再次拥有已持有的锁对象
Condition	条件变量对象, 使一个线程等待另一个线程满足特定的条件触发
Event	事件对象, 普通版的 Condition
Semaphore	信号量, 为线程间共享的资源提供一个“计数器”, 计数开始值为设置的值, 默认为 1
BoundedSemaphore	与 Semaphore 相同, 有边界, 不能超过设置的值
Timer	定时运行的线程对象, 定时器
Barrier	栅栏, 当达到某一栅栏后才可以继续执行

15.3.2 同步

1. 锁

- 锁 (lock) 仅有锁定和非锁定两种状态. 它被创建时为非锁定状态. 它有两个基本方法：acquire 和 release. 当状态为非锁定时, acquire 将状态改为锁定并立即返回. 当状态是锁定时, acquire 将阻塞至其他线程, 调用 release 将其改为非锁定状态, 然后 acquire 调用重置其为锁定状态并返回. release 只在锁定状态下调用, 它将状态改为非锁定并立即返回. 如果尝试释放一个非锁定的锁, 则会引发 RuntimeError 异常.
- 当多个线程在 acquire 等待状态转变为未锁定被阻塞, 然后 release 重置状态为未锁定时, 只有一个线程能继续执行; 至于哪个等待线程继续执行没有定义, 并且会根据实现而不同.
- 具体方法如下：
  - acquire(blocking=True, timeout=-1)

    可以阻塞或非阻塞地获得锁. 当调用时参数 blocking 设置为 True(缺省值), 阻塞直到锁被释放, 然后将锁锁定并返回 True . 在参数 blocking 被设置为 False 的情况下调用, 将不会发生阻塞. 如果调用时 blocking 设为 True 会阻塞, 并立即返回 False, 否则, 将锁锁定并返回 True.

15.3.2 同步

- release()

  释放一个锁. 这个方法可以在任何线程中调用, 不单指获得锁的线程. 当锁被锁定, 将它重置为未锁定, 并返回. 如果其他线程正在等待这个锁解锁而被阻塞, 则只允许其中一个. 在未锁定的锁调用时, 会引发 RuntimeError 异常. 没有返回值.

- locked()

  若获得了锁则返回真值.

15.3.2 同步

例 15.2 定义计数函数 CountNum(thread id), 使用两个线程调用该函数, 其中第一个线程延迟 1 秒, 而第二个线程延迟 2 秒, 在主线程中输出总计数. 定义全局变量 counts, 程序设计如下:

```
import threading
from time import sleep
counts=0
N_threads=2
def CountNum(thread_id):#定义线程函数
    global counts
    sleep(thread_id)
    for i in range(1,101):
        counts=counts+1
    print('thread id:{},its counts is {}\n'.format(thread_id,counts))
    return
if __name__ == "__main__":
    for i in range(N_threads):
        t = threading.Thread(target=CountNum,args=(i+1,))#创建子线程
        t.start()#启动线程
    print('counts=%d Seconds'%counts)
```

15.3.2 同步

- 运行结果如下：

```
counts=0 Seconds
thread id:1,its counts is 100
thread id:2,its counts is 200
```

15.3.2 同步

- 很明显, 主线程并没有得到子线程的计数结果. 如果采用锁, 则当子线程各自计数时, 主线程等待其完成后再进行输出, 其结果就是总计数了. 程序修改为:

```
import threading
from time import sleep
counts=0
N_threads=2
lock = threading.Lock()#创建锁
def CountNum(thread_id):#定义线程函数
    lock.acquire()#请求锁
    global counts
    sleep(thread_id)
    for i in range(1,101):
        counts=counts+1
    print('thread id:{},its counts is {}\n'.format(thread_id,counts))
    lock.release()#释放锁
    return
if __name__ == "__main__":
    for i in range(N_threads):
        t = threading.Thread(target=CountNum,args=(i+1,))#创建子线程
        t.start()#启动线程
    lock.acquire()#请求锁
    print('counts=%d Seconds'%counts)
    lock.release()#释放锁
```

15.3.2 同步

- 运行结果为：

```
thread id:1,its counts is 100
thread id:2,its counts is 200
counts=200 Seconds
```

2. 事件

- 事件 (event) 用于不同进程间的相互通信. 事件对象管理一个内部标识,调用 set 方法可将其设置为 true. 调用 clear 方法可将其设置为 false. 调用wait 方法将进入阻塞直到标识为 true, 这个标识初始时为 false.
- 具体方法如下：
  - is set()

    当且仅当内部标识为 true 时返回 True.
  - set()

    将内部标识设置为true, 所有正在等待这个事件的线程将被唤醒. 当标识为true时, 调用 wait() 方法的线程不会被阻塞.

15.3.2 同步

- clear()

  将内部标识设置为 false, 之后调用 wait() 方法的线程将会被阻塞, 直到调用 set() 方法将内部标识再次设置为 true.

- wait(timeout=None)

  阻塞线程直到内部变量为 true. 如果调用时内部标识为 true, 将立即返回. 否则将阻塞线程, 直到调用 set() 方法将标识设置为 true 或者发生可选的超时. 当提供了 timeout 参数且不是 None 时, 它应该是一个浮点数, 代表操作的超时时间, 以秒为单位 (可以为小数).

15.3.2 同步

例 15.3 针对例 15.2, 使用事件输出线程函数各自的计数.

```
import threading
from time import sleep
counts = 0
N_threads = 2
evGetData = threading.Event()#创建事件对象，内部标志默认为False
evOutput = threading.Event()
def CountNum(thread_id):#定义线程函数
    evGetData.wait()#等待主线程激活
    global counts
    counts = 0
    sleep(thread_id)
    for i in range(1, 100*thread_id+1):
        counts = counts + 1
    print('thread id:{}, its counts is {}'.format(thread_id, counts))
    evOutput.set()#激活主线程
    return
if __name__ == "__main__":
    for i in range(N_threads):
        t = threading.Thread(target=CountNum, args=(i+1,))#创建子线程
        t.start()#启动线程
        evGetData.set()#激活从线程
        evOutput.wait()#等待从线程激活
        print('counts=%d Seconds'%counts)
        evOutput.clear()
```

15.3.2 同步

- 输出结果为：

```
thread id:1,its counts is 100
counts=100 Seconds
thread id:2,its counts is 200
counts=200 Seconds
```

15.4　基于 CPU 进程的并行计算

15.4 基于 CPU 进程的并行计算

- 在 Python 程序中, 代码执行由 Python 虚拟机 (解释器主循环) 来控制.对 Python 虚拟机的访问由全局解释器锁 (global interpreter lock, GIL) 控制,GIL 保证同一时刻只有一个线程在执行. 由于 GIL 的限制, Python 多线程实际只能运行在单核 CPU, 所以 15.3 节中的 threading 模块并不能真正实现多核 CPU 并行计算. 如要实现多核 CPU 并行, 只能通过多进程的方式实现.而 multiprocessing 模块是最常用的多进程模块, 它同时提供了本地和远程并发操作, 通过使用子进程而非线程有效地绕过了 GIL. 因此, multiprocessing模块允许程序员充分利用给定机器上的多个处理器或核进行并行计算.

15.4.1 创建进程

- 使用 multiprocessing 模块创建进程主要有两种方式：一是使用 Process对象, 二是采用 Pool 进程池.

1. Process 对象

- 在 multiprocessing 中, 通过创建一个 Process 对象然后调用它的 start 方法来生成进程, 创建过程同线程创建过程.
- Process 对象表示在单独进程中运行的活动, 具体的 Process 构造函数形式为：
  - class multiprocessing.Process(group=None, target=None, name=None,args=(), kwargs=, *, daemon=None)
- 参数如下：
  - (1) group 应该是 None, 它仅用于兼容 threading.Thread.
  - (2) target 是由 run() 方法调用的可调用对象, 它默认为 None, 意味着什么都没有被调用.
  - (3) name 是进程名称.
  - (4) args 是目标调用的参数元组.
  - (5) kwargs 是目标调用的关键字参数字典.

15.4.1 创建进程

- (6) daemon 可以设置为 True 或 False. 如果是 None (默认值), 则该标志将从创建的进程继承.

Process 对象拥有和 threading.Thread 等价的大部分方法. 具体说明如下:

- run() 表示进程活动的方法, 可以在子类中重载此方法. 标准 run() 方法调用传递给对象构造函数的可调用对象作为目标参数 (如果有), 分别从 args和 kwargs 参数中获取顺序和关键字参数.
- start() 为启动进程活动, 这个方法每个进程对象最多只能调用一次, 它会将对象的 run() 方法安排在一个单独的进程中调用.
- join([timeout]) 表示如果可选参数 timeout 是 None (默认值), 则该方法将阻塞, 直到调用 join() 方法的进程终止. 如果 timeout 是一个正数, 它最多会阻塞 timeout 秒. 请注意, 如果进程终止或方法超时, 那么该方法返回None. 检查进程的 exitcode 以确定它是否终止. 一个进程可以被 join 多次.进程无法 join 自身, 因为这会导致死锁. 尝试在启动进程之前 join 进程是错误的.
- name 为进程的名称. 该名称是一个字符串, 仅用于识别目的, 它没有语义, 可以为多个进程指定相同的名称. 初始名称由构造器设定, 如果没有为构造器提供显式名称, 那么会构造一个形式为 “Process-N1:N2:...:Nk” 的名称.
- daemon为进程的守护标志, 一个布尔值. 这必须在start()被调用之前设置. 注意：在Windows上要想使用进程模块, 就必须把有关进程的代码写在当前.py文件的if name ==‘main’:语句的下面, 才能正常使用Windows下的进程模块. Unix/Linux 下则不需要.

15.4.1 创建进程

- **例 15.4** 定义函数 HelloWorld, 其内容为延迟 1 秒后, 输出形如 Helloworld! current process name=Process-1 这样的字符串. 在主进程中创建 4 个进程, 同时在主进程中输出形如 Hello world! current process name=MainProcess 这样的字符串.

- 在 py 文件中输入程序:

```
from time import sleep
import multiprocessing as mp
def HelloWorld():
    name=mp.current_process().name
    sleep(1)#延迟1秒
    print('Hello world! current_process name=%s \n'%name)
    return
if __name__ == "__main__":
    name=mp.current_process().name
    print('Hello world! current_process name=%s \n'%name)
    for i in range(4):
        p = mp.Process(target=HelloWorld)
        p.start()
        p.join()
        name=mp.current_process().name
    print('Hello world! current_process name=%s \n'%name)
打开Anaconda Prompt, 键入以下命令(文件位置要换成自己的位置):
python E:\MyPython\并行计算\eg4_1.py
```

15.4.1 创建进程

- 输出结果如下：

```
Hello world! current_process name=MainProcess
Hello world! current_process name=Process-1
Hello world! current_process name=Process-2
Hello world! current_process name=Process-3
Hello world! current_process name=Process-4
Hello world! current_process name=MainProcess
```

15.4.1 创建进程

- 一般来讲, 在主程序中创建并启动子进程, 然后主进程有处理数据、与子进程通信等工作, 并等待子进程完成任务, 基于这种框架, 上述程序修改为:

```
from time import sleep
import multiprocessing as mp
N_core=4#进程数
def HelloWorld(ID):
    name=mp.current_process().name
    sleep(1)#延迟1秒
    print('Hello world! current_process name=%s,ID=%d \n'%(name,ID))
    return
if __name__ == "__main__":
    name=mp.current_process().name#输出主进程名
    print('Hello world! current_process name=%s \n'%name)
    processes = []#进程列表
    for id in range(N_core):#创建子进程
        p = mp.Process(target=HelloWorld,args=(id+1, ))
        p.start()
        processes.append(p)
    name=mp.current_process().name#输出主进程名
    print('Hello world! current_process name=%s \n'%name)
    #主进程处理
    #......
    #主进程处理结束
    #等待各子进程处理结果
    for p in processes:
        p.join()
```

15.4.1 创建进程

2. Pool 进程池

- Pool 进程池可以提供指定数量的进程供用户调用, 当有新的请求提交到Pool 中时, 如果池还没有满, 就会创建一个新的进程来执行请求. 如果池满,请求就会告知先等待, 直到池中有进程结束, 才会创建新的进程来执行这些请求.
- Pool 进程池常见的方法有：
  - apply 函数, 原型：apply(func[, args=()[, kwds=]])
    该函数用于传递不定参数, 使用阻塞方式调用 func, 执行完一个进程后再去执行其他进程.
  - apply async 函数, 原型：apply async(func[, args=()[, kwds=[, callback=None]]])
    该函数与 apply 用法一致, 但它是非阻塞的且支持结果返回后进行回调.能够多个线程同时异步执行.
  - map() 函数, 原型：map(func, iterable[, chunksize=None])
    该函数与内置的 map 函数用法行为基本一致, 它会使进程阻塞直到结果返回. 注意：虽然第二个参数是一个迭代器, 但在实际使用中, 必须在整个队列都就绪后, 程序才会运行子进程.

15.4.1 创建进程

- map async() 函数, 原型：map async(func, iterable[, chunksize[, callback]])

  该函数与 map 用法一致, 但是它是非阻塞的. 其有关事项见 apply async.
- close 函数：关闭进程池, 使其不再接受新的任务.
- terminal：结束工作进程, 不再处理未处理的任务.
- join 函数：表示主进程阻塞等待子进程的退出, join 要在 close 或 termi_x0002_nate 之后使用.

15.4.1 创建进程

例 15.5 用 Pool 进程池完成例 15.4.

```
from time import sleep
import multiprocessing as mp
N_core=4#进程数
def HelloWorld(ID):
    name = mp.current_process().name
    sleep(1)#延迟1秒
    print('Hello world! current_process name=%s,ID=%d \n'%(name,ID))
    return
if __name__ == "__main__":
    name = mp.current_process().name#输出主进程名
    print('Hello world! current_process name=%s \n'%name)
    pools = []#进程池列表
    pool = mp.Pool(processes=N_core)#创建进程池
    for id in range(N_core):
        pools.append(pool.apply_async(HelloWorld, args = (id+1,)))
        pool.close()
    name=mp.current_process().name#输出主进程名
    print('Hello world! current_process name=%s \n'%name)
    #主进程处理
    #......
    #主进程处理结束
    #等待各子进程处理结果
    pool.join()
打开Anaconda Prompt, 键入以下命令(文件位置要换成自己的位置):
python E:\MyPython\并行计算\eg5.py
```

15.4.1 创建进程

- 输出结果如下：

```
Hello world! current_process name=MainProcess
Hello world! current_process name=MainProcess
Hello world! current_process name=SpawnPoolWorker-1,ID=1
Hello world! current_process name=SpawnPoolWorker-2,ID=2
Hello world! current_process name=SpawnPoolWorker-3,ID=3
Hello world! current_process name=SpawnPoolWorker-4,ID=4
```

15.4.2 进程间通信

- 在 multiprocessing 中, 进程间通信主要有数据共享和数据传递, 本书只介绍数据传递, 而数据传递有队列 (queue) 和管道 (pipe) 两种, 主要区别为:
  - (1) 队列使用 put 和 get 维护队列, 管道使用 send 和 recv 维护队列.
  - (2) 管道只提供两个端点, 而队列没有限制.
  - (3) 队列的封装比较好, 队列只提供一个结果, 可以被多个进程同时调用;而管道返回两个结果, 分别由两个进程调用.
  - (4) 队列的实现基于管道, 所以管道的运行速度比队列快很多.
  - (5) 当只需要两个进程时, 管道更快, 当需要多个进程同时操作队列时,使用队列.

1. 队列

- 队列是一个先进先出 (FIFO) 的数据结构, 很多场景需要按先来后到的顺序进行处理. 其形式为:

class multiprocessing.Queue([maxsize])

- Queue 返回一个使用一个管道和少量锁和信号量实现的共享队列实例.当一个进程将一个对象放进队列中时, 一个写入线程会启动并将对象从缓冲区写入管道中.

15.4.2 进程间通信

- 除了 task done() 和 join() 之外, Queue 实现了标准库类 queue.Queue 中所有的方法. 常见方法有:
  - (1) put(obj[, block[, timeout]])

    将 obj 放入队列. 如果可选参数 block 是 True (默认值) , 而且 timeout 是None (默认值), 将会阻塞当前进程, 直到有空的缓冲槽. 如果 timeout 是正数,将会在阻塞了最多 timeout 秒之后还是没有可用的缓冲槽时抛出 queue.Full异常. 反之 (block 是 False 时), 仅当有可用缓冲槽时才放入对象, 否则抛出queue.Full 异常 (在这种情形下 timeout 参数会被忽略).
  - (2) get([block[, timeout]])

    从队列中取出并返回对象. 如果可选参数 block 是 True (默认值) 而且timeout 是 None (默认值), 将会阻塞当前进程, 直到队列中出现可用的对象.如果 timeout 是正数, 将会在阻塞了最多 timeout 秒之后还是没有可用的对象时抛出 queue.Empty 异常. 反之 (block 是 False 时), 仅当有可用对象能够取出时返回, 否则抛出 queue.Empty 异常 (在这种情形下 timeout 参数会被忽略).
  - (3) close()

    指示当前进程将不会再往队列中放入对象. 一旦所有缓冲区中的数据被写入管道之后, 后台的线程会退出.

15.4.2 进程间通信

- (4) join thread()

  等待后台线程. 这个方法仅在调用了 close() 之后可用. 这会阻塞当前进程, 直到后台线程退出, 确保所有缓冲区中的数据都被写入管道中. 默认情况下, 如果一个不是队列创建者的进程试图退出, 它会尝试等待这个队列的后台线程. 这个进程可以使用 cancel join thread() 让 join thread() 什么都不做直接跳过.

- (5) cancel join thread()

  防止 join thread() 方法阻塞当前进程. 具体而言, 这防止进程退出时自动等待后台线程退出.

15.4.2 进程间通信

例 15.6 定义函数 HelloWorld, 形参为队列和 ID 号, 获得当前进程的name, 把形如 [‘Hello world!’, ID,name] 这样的列表加入队列中. 在主进程中创建 4 个子进程, 同时在主进程中得到并输出子进程的列表.

```
import multiprocessing as mp
N_core=4#进程数
def HelloWorld(q,ID):
    name=mp.current_process().name
    q.put([ 'Hello world!', ID,name])
    return
if __name__ == '__main__':
    q = mp.Queue()
    processes = []#进程列表
    for id in range(N_core):
        p = mp.Process(target=HelloWorld,args=(q,id+1))
        p.start()
        processes.append(p)
    print(q.get())
    for p in processes:#等待各子进程处理结果
        p.join()
打开Anaconda Prompt, 键入以下命令(文件位置要换成自己的位置):
python E:\MyPython\并行计算\eg6.py
```

15.4.2 进程间通信

▶ 输出结果如下：

```
['Hello world!', 1, 'Process-1']
['Hello world!', 2, 'Process-2']
['Hello world!', 3, 'Process-3']
['Hello world!', 4, 'Process-4']
```

2. 管道

- 管道返回一个由管道连接的对象, 默认情况下是双向. 每个连接对象都有 send() 和 recv() 方法 (相互之间的). 请注意, 如果两个进程 (或线程) 同时尝试读取或写入管道的同一端, 那么管道中的数据可能会损坏. 当然, 在不同进程中同时使用管道的不同端的情况下不存在损坏的风险. 其形式为

multiprocessing.Pipe([duplex])

- 返回一对 Connection 对象 (conn1, conn2) , 分别表示管道的两端. 如果 duplex 被置为 True (默认值), 那么该管道是双向的. 如果 duplex 被置为False , 那么该管道是单向的, 即 conn1 只能用于接收消息, 而 conn2 仅能用于发送消息.

15.4.2 进程间通信

例 15.7 定义函数 HelloWorld, 形参为队列和 ID 号, 获得当前进程的name, 把形如 [‘Hello world!’, ID,name] 这样的列表发送到主进程中. 在主进程中创建 4 个子进程, 同时在主进程中接收子进程传输的数据.

```
import  multiprocessing as mp
N_core=4#进程数
def HelloWorld(conn,ID):
    name=mp.current_process().name
    conn.send(['Hello world!', ID,name])
    conn.close()
    return
if __name__ == '__main__':
    parent_conn, child_conn = mp.Pipe()
    processes = []#进程列表
    for id in range(N_core):
        p = mp.Process(target=HelloWorld,args=(child_conn,id+1))
        p.start()
        processes.append(p)
    print(parent_conn.recv())
    for p in processes:#等待各子进程处理结果
        p.join()
```

- 运行情况同例 15.6 .

15.4.3 同步

- threading 模块中的同步机制, 在多进程 multiprocessing 模块中也有, 比如锁机制、事件、信号量、栅栏.

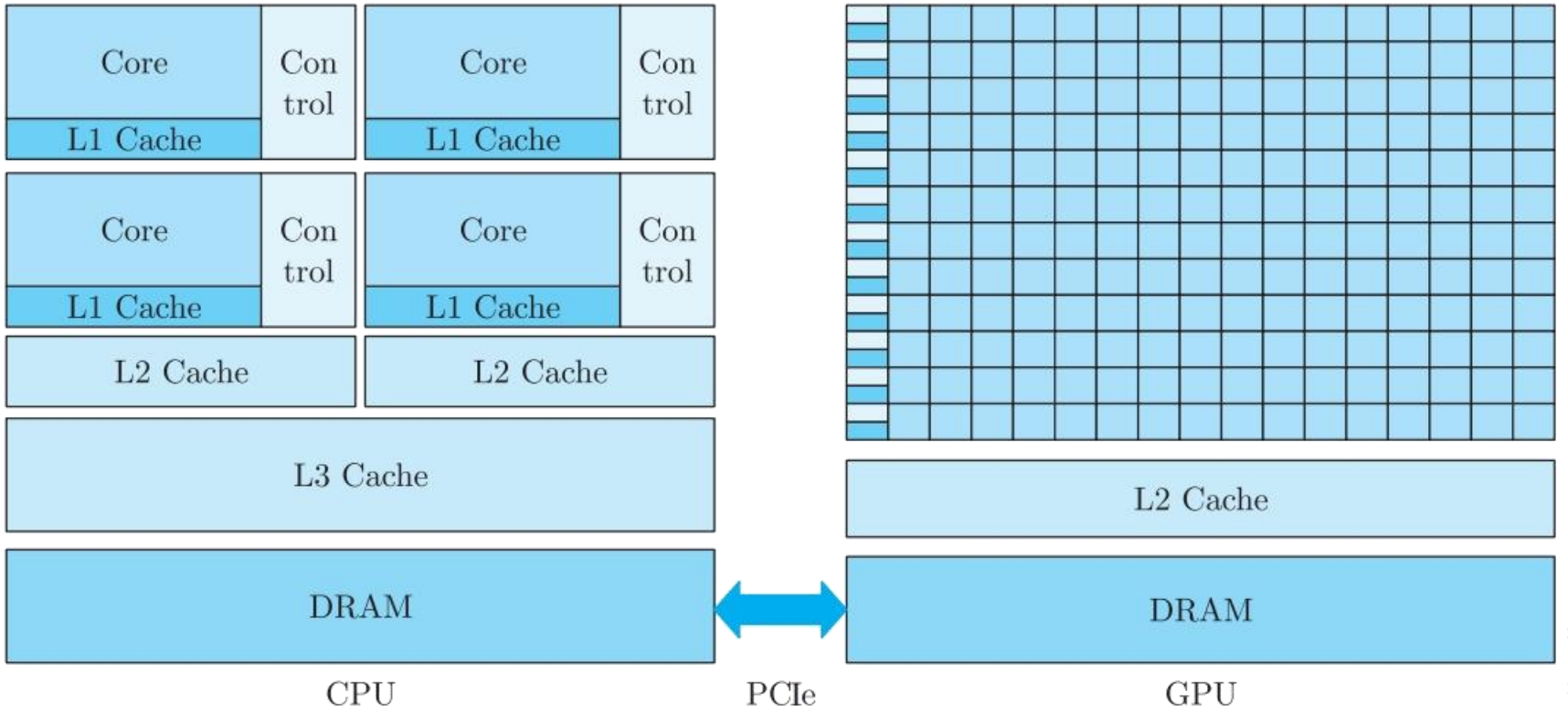
15.4　基于 GPU 线程的并行计算

15.5 基于 GPU 线程的并行计算

- 前面两节是基于 CPU 的并行计算, 实际上, 在机器学习和深度学习中,使用更多的是基于 GPU 线程的并行计算, 其优势在于核数众多, 计算速度更快, 特别对于矩阵乘法和卷积具有极大的计算优势. 另外, GPU 还拥有大量且快速的寄存器以及 L1 缓存的易于编程性, 使得 GPU 非常适合用于深度学习. 目前, 常用的 GPU 计算范式是来自英伟达的 CUDA 并行计算框架, 这种架构可以通过 GPU 加速使得机器学习并行化.

15.5.1 CUDA 基本概念

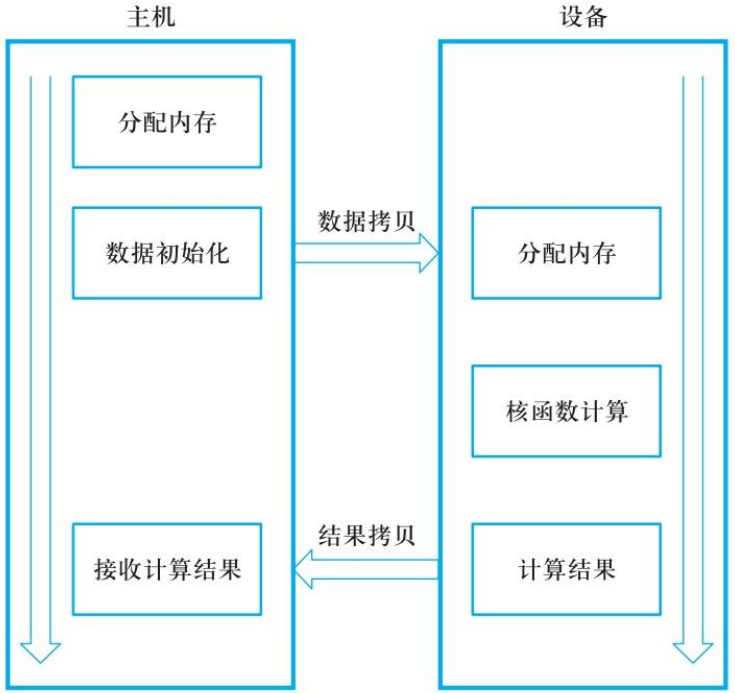
- CUDA 并不是一个独立运行的计算平台, 而需要与 CPU 协同工作, 可以看成是 CPU 的协处理器, 因此当我们在说 CUDA 并行计算时, 其实指的是基于 CPU+GPU 的异构计算架构. 在异构计算架构中, GPU 与 CPU 通过PCIe 总线连接在一起来协同工作, 如图 15.9 所示.



15.5.1 CUDA 基本概念

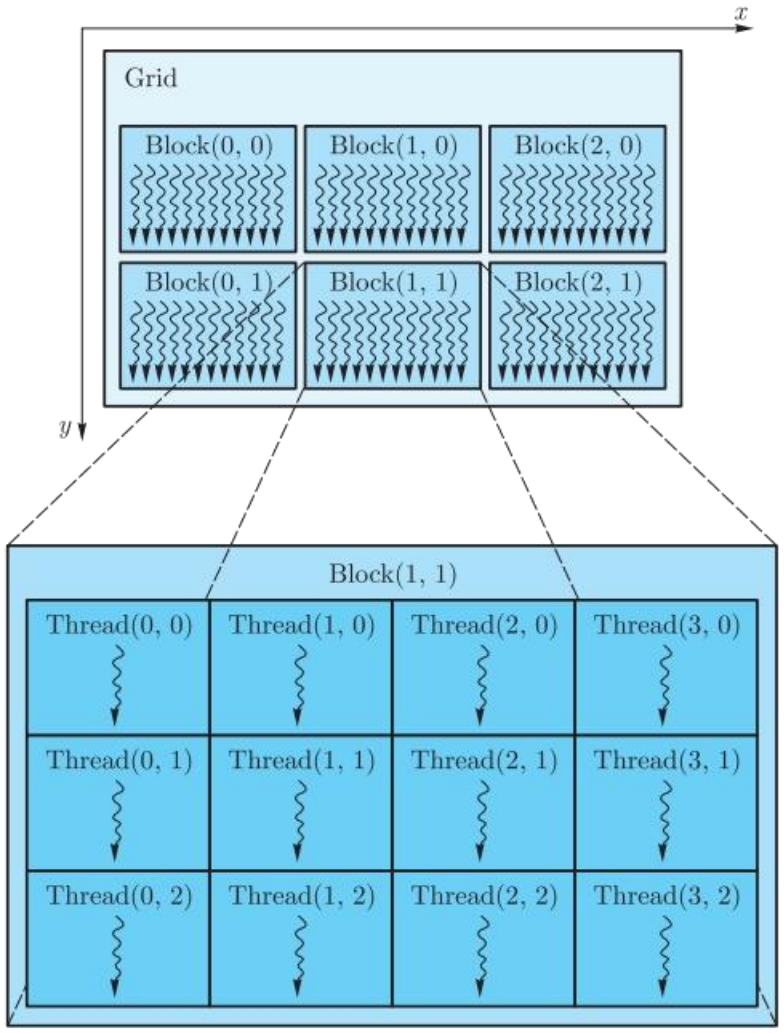
- 其中, CPU 所在位置称为主机 (host), 又称为主处理器, 在并行计算中,主要涉及 CPU 及其内存, 而 GPU 所在位置称为设备 (device), 又称为协处理器, 在并行计算中, 主要涉及 GPU 及其内存. 在 CUDA 程序中既包含主机程序, 又包含设备程序, 它们分别在 CPU 和 GPU 上运行. 同时, 主机与设备之间可以进行通信, 即可以相互进行数据拷贝. 在设备上的计算通过核函数(kernel) 形式执行, 该函数以 global 为前缀作为标记. 如图 15.10 所示, 典型的 CUDA 程序的执行流程如下:

  - (1) 分配主机内存, 并进行数据初始化;
  - (2) 配设备内存, 并从主机将数据拷贝到设备上;
  - (3) 调用 CUDA 的核函数在设备上完成指定的运算;
  - (4) 将设备上的运算结果拷贝到主机上;
  - (5) 释放设备和主机上分配的内存.



15.5.2 CUDA 线程组织

- 如图 15.11 所示, CUDA 二维线程组织从逻辑层面来讲, 主要分两层, 外层称为网格 (grid), 同一个网格上的线程共享相同的全局内存空间, 里层称为线程块 (block). 网格是由若干线程块组成的, 而每个线程块里面包含很多线程. 线程坐标的原点在左上角, 向右为 x 方向, 向左为 y 方向.
- 网格和线程块都是定义为 dim3 类型的变量, dim3 可以看成是包含三个无符号整数 (x, y, z) 成员的结构体变量, 在定义时, 缺省值初始化为 1.



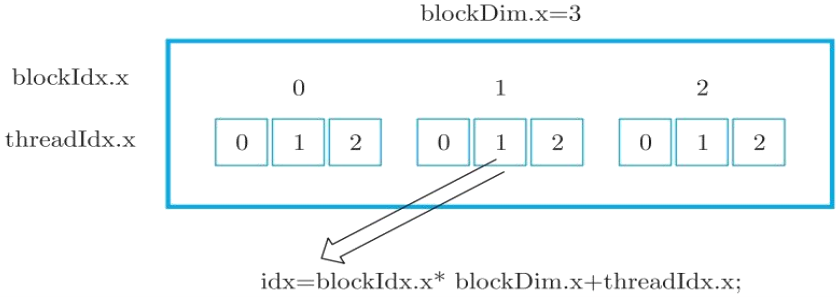
15.5.2 CUDA 线程组织

- 因此,网格和线程块可以灵活地定义为 1-dim, 2-dim 以及 3-dim 结构. 核函数在调用时也必须通过执行配置 <<<grid, block>>> 来指定核函数所使用的线程数及结构. 图 15.11 中的网格和线程块可以这样定义:

```
dim3 grid(3, 2,1);
dim3 block(4, 3,1);
kernel_fun<<< grid, block >>>(prams...);
```

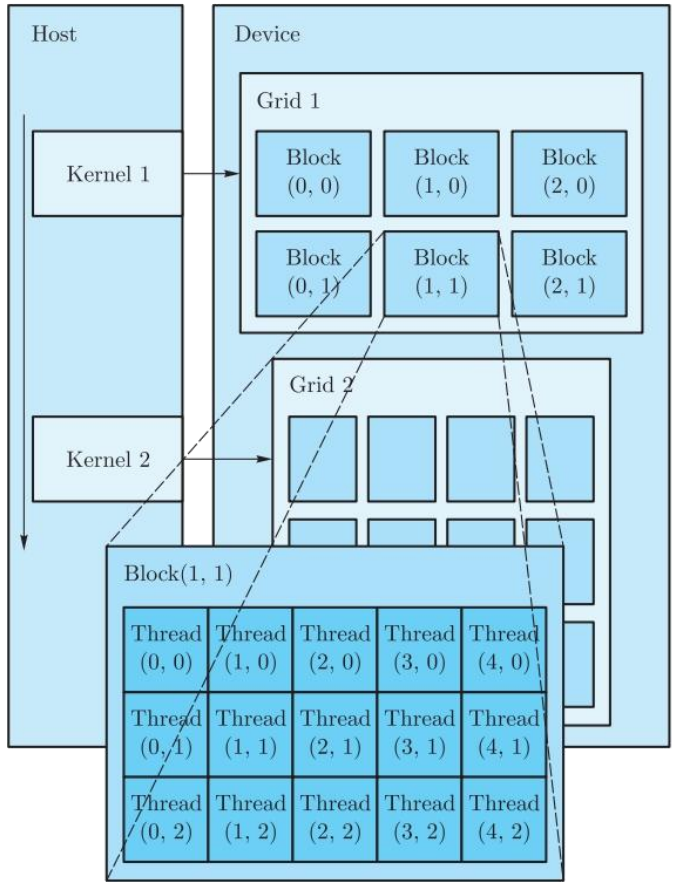
- 所以, 一个线程需要两个内置的坐标变量 (blockIdx,threadIdx) 来唯一标识, 它们都是 dim3 类型变量, 其中 blockIdx 指明线程所在网格中的位置, 而threaIdx 指明线程所在线程块中的位置, 如图 15.11 中的 Thread(1,1) 满足: threadIdx.x = 1, threadIdx.y = 1, blockIdx.x = 1, blockIdx.y = 1.

- 对于每个线程, 以一维为例, 在实际计算中, 坐标 x 的计算方式如图 15.12,箭头所在位置的线程的坐标 x = 5.



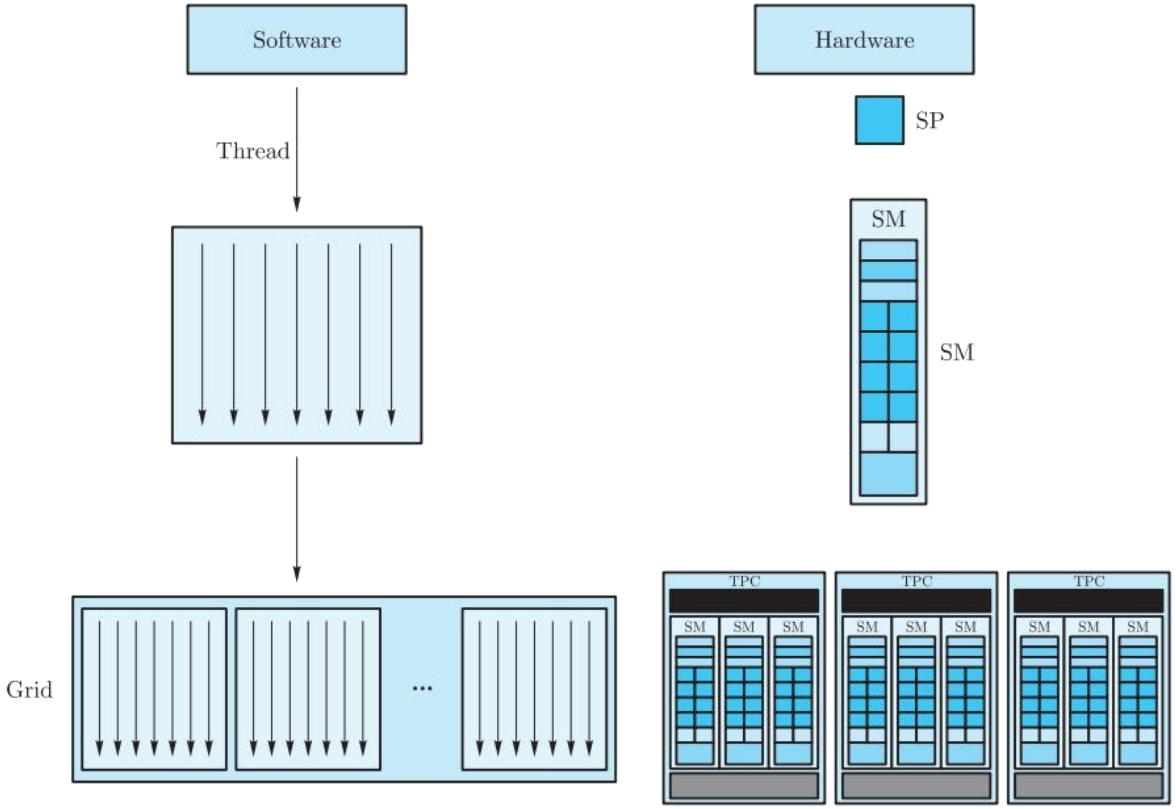
15.5.2 CUDA 线程组织

- 同理, 可以得到二维的索引坐标：idx = blockIdx.x * blockDim.x + threadIdx.x; idy = blockIdx.y * blockDim.y + threadIdx.y.
- 对于核函数来讲, 一般情况下, 一个核函数运行在一个网格上, 如图15.13所示.



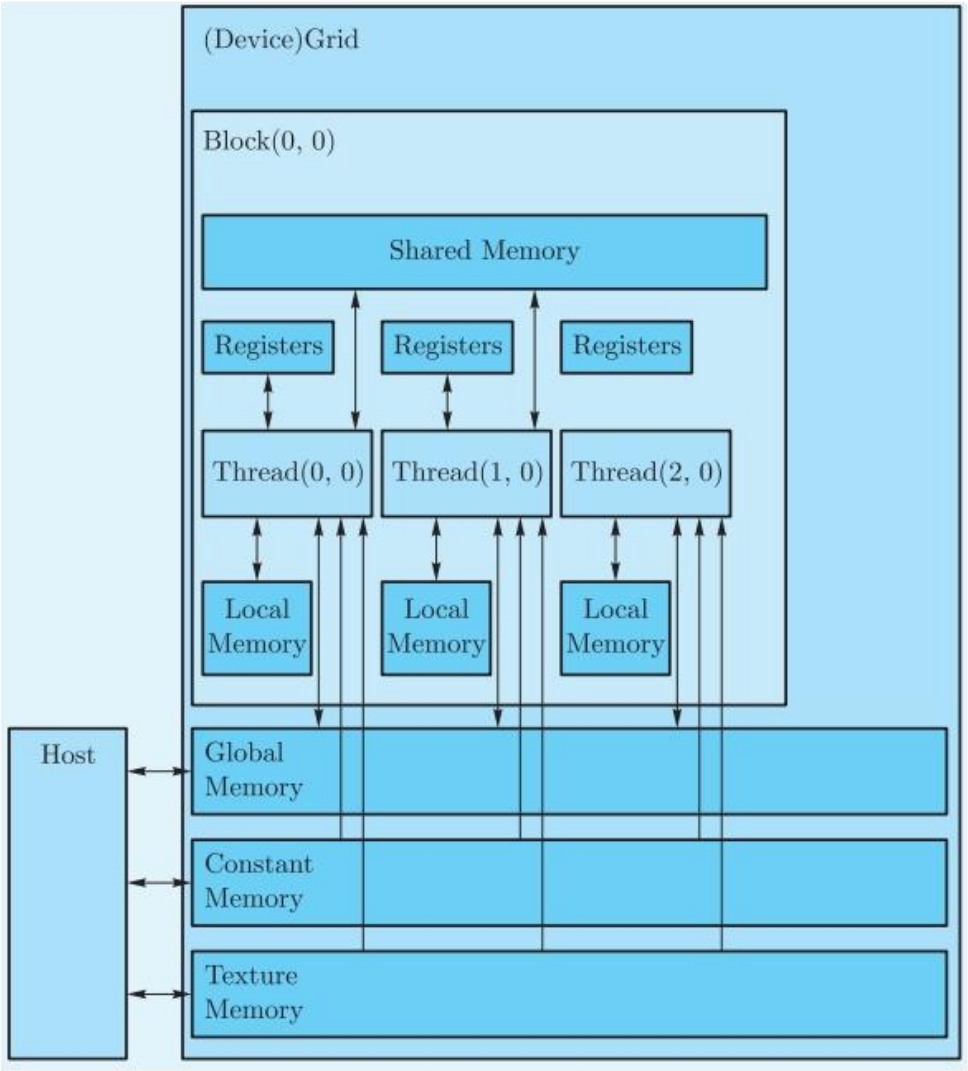
15.5.2 CUDA 线程组织

- 上面讲的是软件逻辑层方面的线程组织方式. 如图 15.14 所示, 从显卡的物理层面来看, 核心组件是流式多处理器 (streaming multiprocessor, SM), 包括 CUDA 核心 (也称为 streaming processor, SP)、共享内存、寄存器等, SM可以并发地执行数百个线程, 并发能力就取决于 SM 所拥有的资源数. 当一个核函数被执行时, 它的网格中的线程块被分配到 SM 上, 一个线程块只能在一个 SM 上被调度, 而 SM 一般可以调度多个线程块, 基本的执行单元是线程束 (warps), 线程束包含 32 个线程, 这些线程同时执行相同的指令, 但是每个线程都包含自己的指令地址计数器和寄存器状态, 也有自己独立的执行路径.



15.5.3 CUDA 内存组织

- CUDA 内存组织如图 15.15 所示. 可以看到, 每个线程有自己的私有本地内存 (local memory), 而每个线程块又包含共享内存 (shared memory), 可以被线程块中所有线程共享, 其生命周期与线程块一致. 此外, 所有的线程都可以访问全局内存 (global memory). 还可以访问一些只读内存块：常量内存(constant memory) 和纹理内存 (texture memory).



15.5.4 PyCUDA

- PyCUDA 作为 Python 的第三方库, 可以访问 NVDIA 的 CUDA 并行计算 API. 它的异构编程模式和前面 CUDA 一致, 它自己具有的特点是:
  - (1) 对象的自动清理. 它使编写正确、无泄漏和无崩溃的代码变得更加容易. PyCUDA 在其分配的所有内存被释放之前, 不会与上下文分离.
  - (2) 编程方便. 提供了 pycuda.compiler.SourceModule 和 pycuda.gpuarray.GPUArray 等接口, 让 CUDA 编程甚至比 NVDIA 的基于 C 语言的运行时更方便.
  - (3) 完整支持 CUDA. PyCUDA 能够完全支持 CUDA 的 API. 并能够自动进行错误检查, 把所有 CUDA 错误自动转换为 Python 异常.
  - (4) 加速效果好. PyCUDA 的基础层是用 C++ 编写的, 因此计算速度比较快.

1. pycuda 安装

- 到官网下载 CUDA 包并安装, 到微软官网下载 Visual Studio 2015 以上的 C++ 开发工具并安装. 到 Python 环境使用 pip install pycuda 安装pycuda.

15.5.4 PyCUDA

2. 实现过程

- 下面以随机生成的两个向量的点积计算为例, 讲解实现过程. 其中, driver是导入 CUDA 的 API, autoinit 用来启动和自动初始化 GPU 系统, Source_x0002_Module 是 NVDIA 编译器 (nvcc) 的指令, 指示要编译并上传到设备的对象.很明显, 以下核函数是 C 语言形式.

```
#导入相关库
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
from pycuda.compiler import SourceModule
#在CPU生成随机向量
a = numpy.random.randn(5).astype(numpy.float32)
b = numpy.random.randn(5).astype(numpy.float32)
#定义核函数
mod = SourceModule("""
__global__ void vector_dot(float *dest, float *a, float *b)
{const int i = threadIdx.x;dest[i] = a[i] * b[i];}
""")
func = mod.get_function("vector_dot")
#线程设置和执行核函数
dest = numpy.zeros_like(a)
func(drv.Out(dest), drv.In(a), drv.In(b), block=(400,1,1), grid=(1,1))
print(dest)#输出结果
```

15.5.5 TensorFlow

- TensorFlow 是一个用于设计和部署数值计算的软件库, 主要专注于机器学习中的应用程序. 借助这个库, 可以将算法描述为相关运算的图形, 这些运算可以在各种支持 GPU 的平台上执行, 包括便携式设备、台式机和高端服务器.
- 安装好 GPU 版的 TensorFlow 后, 使用下面程序来测试 Tensorflow 是否支持 GPU:

```
import tensorflow as tf
tf.config.list_physical_devices()
```

- 如下输出结果表示支持 GPU:

```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'),
PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

15.5.5 TensorFlow

- 可以用以下程序测试 CPU 和 GPU 下运行时间：

```
import tensorflow as tf
import timeit
def cpu_run():
    with tf.device('/cpu:0'):
    c = tf.matmul(cpu_a, cpu_b)
    return c
def gpu_run():
    with tf.device('/gpu:0'):
    c = tf.matmul(gpu_a, gpu_b)
    return c
cpu_time = timeit.timeit(cpu_run, number=10)
gpu_time = timeit.timeit(gpu_run, number=10)
print('run time:', cpu_time, gpu_time)
```

15.6　并行计算实践

实践代码