局部性和数据复用

翻自：《Introduction to High Performance Scientific Computing》-Victor Eijkhout

李岳昆(realyurk@gmail.com)、蒋志政(gezelligheid@aliyun.com)译

知乎专栏：高性能计算翻译计划

Github专栏：https://github.com/realYurkOfGitHub/translation-Introduction-to-HPC

局部性和数据复用

算法的执行不仅包含计算操作，也包含数据传输部分，事实上，数据传输可能是影响算法效率的主要因素。由于缓存和寄存器的存在，数据传输量可以通过编程的方式最小化，使数据尽可能地留在处理器附近。这部分是一个巧妙编程的问题，但我们也可以看看理论上的问题：算法是否一开始就允许这样做。

事实证明，在科学计算中，数据往往主要与在某种意义上靠近的数据互动，这将导致数据的局部性；1.6.2节。通常这种局部性来自于应用的性质，就像第四章看到的PDEs的情况。在其他情况下，如分子动力学（第7章），没有这种内在的局部性，因为所有的粒子都与其他粒子相互作用，为了获得高性能，需要相当的编程技巧。

数据复用和计算密度

在前面的章节中，我们了解到处理器的设计有些不平衡：加载数据比执行实际操作要慢。这种不平衡对于主存储器来说是很大的，而对于各种高速缓存级别来说则较小。因此，我们有动力将数据保存在高速缓存中，并尽可能地保持数据的复用量。

当然，我们首先需要确定计算是否允许数据被重复使用。为此，我们定义了一个算法的计算密度如下。

- 如果 n 是一个算法所操作的数据项的数量，而 $f(n)$ 是它所需要的操作的数量，那么算术强度就是 $f(n)/n$ 。

(我们可以用浮点数或字节来衡量数据项。后者使我们更容易强度与处理器的硬件规格相关联)

计算密度也与延迟隐藏有关：即你可以减轻计算活动背后的数据加载对性能的负面影响的概念。要做到这一点，你需要比数据加载更多的计算来使这种隐藏有效。而这正是计算强度的定义：每一个字节/字/数字加载的高比率操作。

示例：向量操作

考虑到向量加法

$$\forall_i : x_i \leftarrow x_i + y_i \tag{1}$$

这涉及到三次内存访问（两次加载和一次存储）和每次迭代的一次操作，给出的算术强度为1/3。axpy（表示 "a乘以 x 加 y "）操作

$$\forall_i : x_i \leftarrow ax_i + y_i \tag{2}$$

有两个操作，但内存访问的数量相同，因为 a 的一次性负载被摊销了。因此，它比简单的加法更有效率，重用率为2/3。因此，它比简单的加法更有效，重用率为2/3。

内积计算

$$\forall_i : s \leftarrow s + x_i \cdot y_i \tag{3}$$

在结构上类似于axpy操作，每次迭代涉及一个乘法和加法，涉及两个向量和一个标量。然而，现在只有两个加载操作，因为 s 可以保存在寄存器中，只在循环结束时写回内存。这里的重用是1。

示例：矩阵操作

考虑矩阵乘法

$$\forall_{i,j}: c_{ij} = \sum_k a_{ik} b_{kj} \tag{4}$$

这涉及 $3n^2$ 个数据项和 $2n^3$ 个运算，属于高阶运算。算术强度为 $O(n)$，每个数据项将被使用$O(n)$次。这意味着，通过适当的编程，这种操作有可能通过将数据保存在快速缓存中来克服带宽/时钟速度的差距。

练习 1.14 根据上述定义，矩阵-矩阵乘积作为一种操作，显然具有数据重用性。矩阵-矩阵乘积显然具有数据复用。请你论证一下，这种复用并不是自然形成的，是什么决定了初始算法中国呢缓存是否对数据进行复用?

[在这次讨论中，我们只关心某个特定实现的操作数，而不是数学操作。例如，有一些方法可以在少于$O(n^3)$的操作中执行矩阵-矩阵乘法和高斯消除算法[189, 167]。然而，这需要不同的实现方式，在内存访问和重用方面有自己的分析]。

矩阵与矩阵乘积是LINPACK基准[51]的核心；见2.11.4节。如果将其作为评价计算机从能的标准则结果可能较为乐观：矩阵与矩阵的乘积是一个具有大量数据复用的操作，因此这对内存带宽并不敏感，对于并行计算及而言，这对网络通信也并不敏感。通常情况下，计算机在Linpack基准测试中会达到其峰值性能的60-90%，而其他测试标准得到的数值则可能较低。

Roofline模型

有一种平价计算及性能的理想模型，就是所谓的「屋脊线」（roofline model）模型[202]，该模型指出：性能受两个因素的制约，如图1.16的第一个图所示。

1. 图中顶部的横线所表示的峰值性能是对性能的绝对约束3，只有在CPU的各个方面（流水线、多个浮点单元）都完美使用的情况下才能达到。这个数字的计算纯粹是基于CPU的特性和时钟周期；假定内存带宽不是一个限制因素。
2. 每秒的操作数受限于带宽、绝对数和计算密度的乘积。

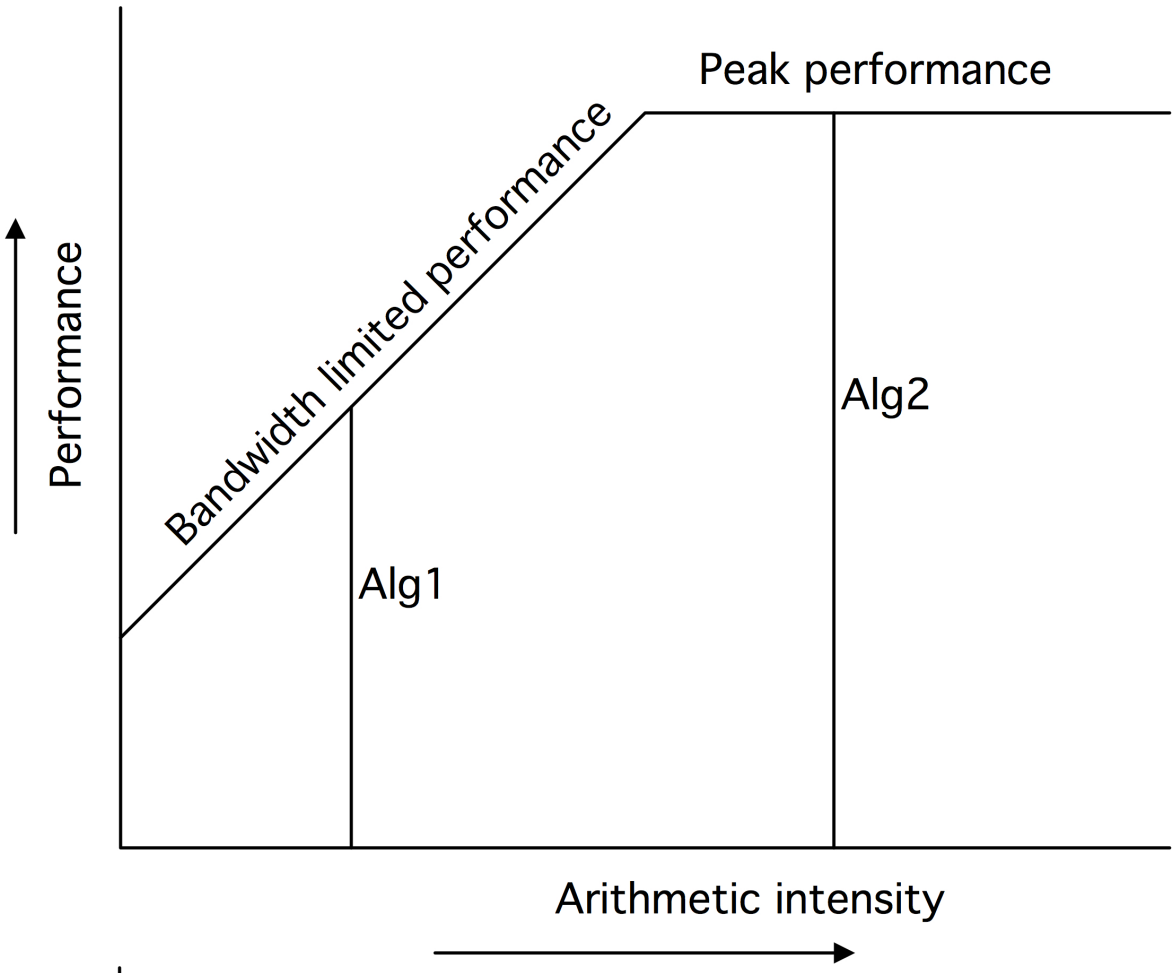
$$\frac{operations}{second} = \frac{operations}{\text{data item}} \cdot \frac{\text{data items}}{second} \tag{5}$$

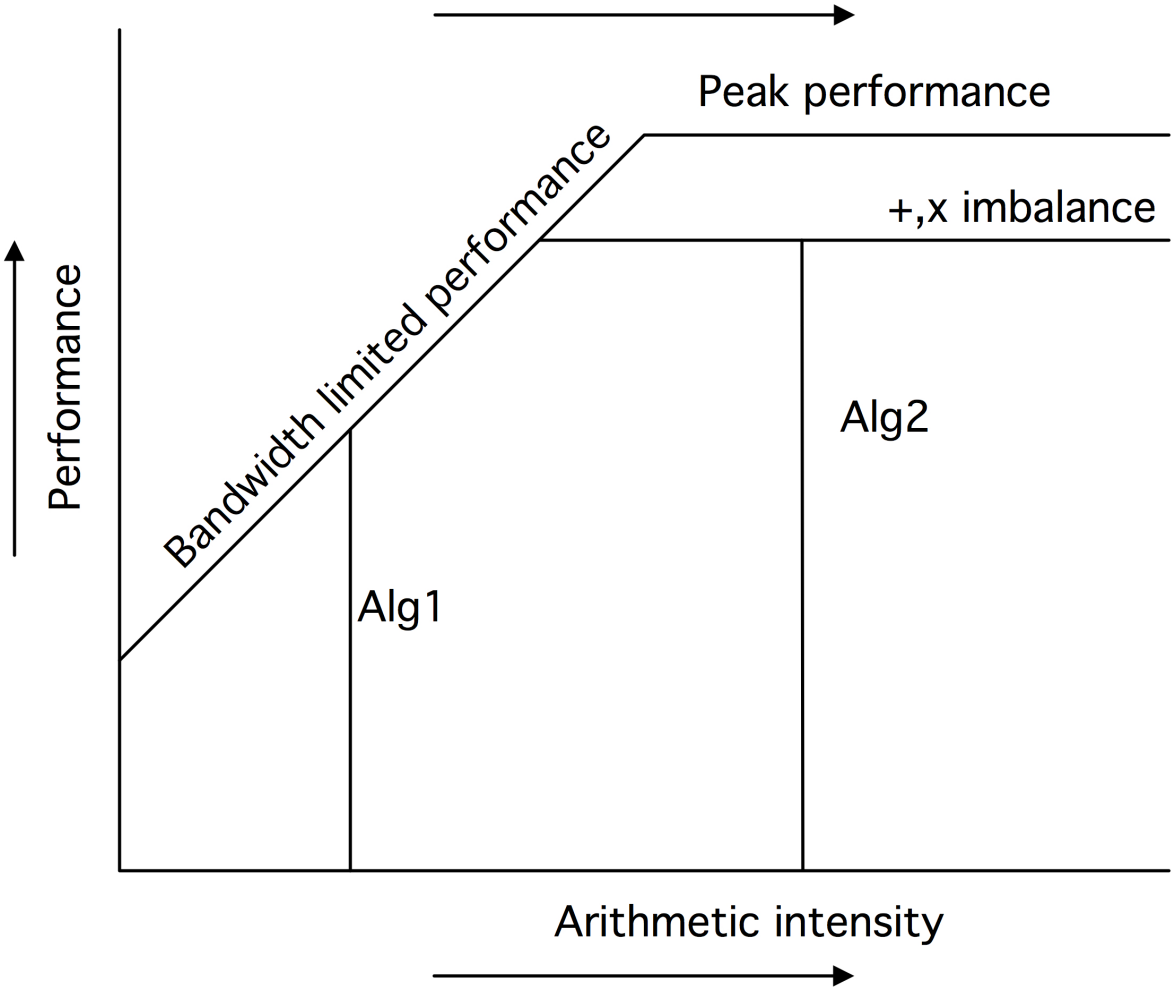
这是由图中的线性增长线所描述的。

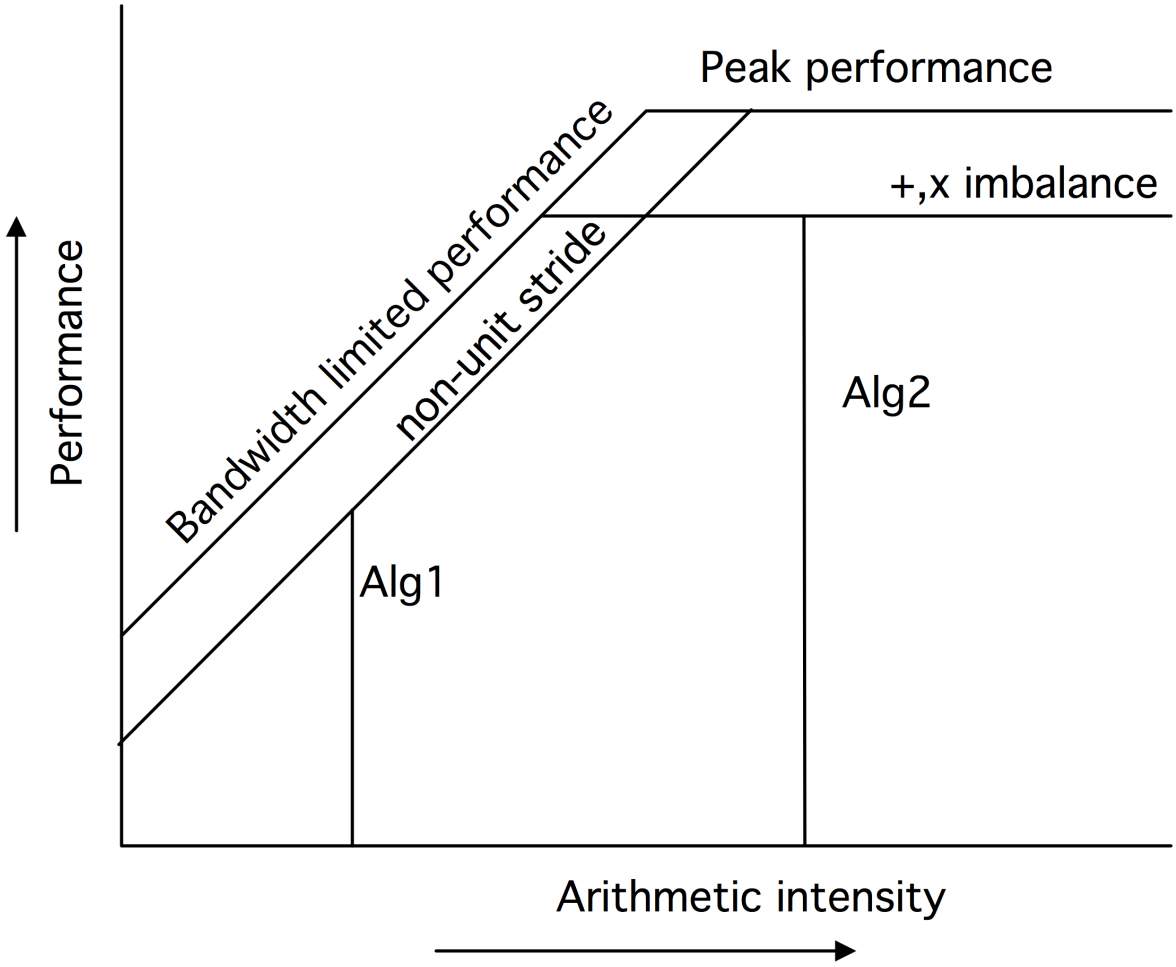
Roofline模型优雅地指出了影响性能的因素。例如，如果一个算法没有使用全部的SIMD宽度，这种不平衡会降低可达到的峰值。图1.16中的第二张图显示了降低上限的各种因素。还有各种降低可用带宽的因素，比如不完善的数据隐藏。这在第三张图中由倾斜的屋顶线的降低表示。

对于一个给定的计算密度，其性能是由其垂直线与Roofline相交的位置决定的。如果这是在水平部分，那么该计算被称为受「计算约束」（compute-bound）：性能由处理器的特性决定，而带宽不是问题。另一方面，如果这条垂直线与屋顶的倾斜部分相交，那么计算被称为受「带宽约束」（bandwidth-bound）：性能由内存子系统决定，处理器的全部能力没有被使用。

练习 1.15 如何确定一个给定的程序内核是受到带宽约束还是计算约束的？


Peak performance
Bandwidth limited performance
Performance
Alg2
Alg1
Arithmetic intensity


Peak performance
+,x imbalance
Bandwidth limited performance
Performance
Alg2
Alg1
Arithmetic intensity



局部性

由于从缓存中读取数据的时间开销要小于从内存中读取，我们当然希望以这种方式进行编码，进而使缓存中的数据最大程度上得到复用。虽然缓存中的数据不受程序员的控制，甚至编写汇编语言也无法控制（在Cell处理器和一些GPU中，低级别的内存访问可以由程序员控制），但在大多数CPU中，知道缓存的行为，明确什么数据在缓存中，并在一定程度上控制它，还是有可能的。

这里的两个关键概念是「时间局部性」（temporal locality）和「空间局部性」（spatial locality）。时间局部性是最容易解释的：即数据使用一次后短时间内再次被使用。由于大多数缓存使用LRU替换策略，如果在两次引用之间被引用的数据少于缓存的大小，那么该元素仍然会存在缓存之中，进而实现快速访问。而对于其他的替换策略，例如随机替换，则不能保证同样结果。

时间局部性

下面为时间局部性的例子，考虑重复使用一个长向量：

```
for (loop=0; loop<10; loop++) {
     for (i=0; i<N; i++) {
       ... = ... x[i] ...
     }
}
```

x 的每个元素将被使用10次，但是如果向量（加上其他被访问的数据）超过了缓存的大小，每个元素将在下一次使用前被刷新。因此，$x[i]$ 的使用并没有表现出时间局部性：再次使用时的时间间隔太远，使得数据无法停留在缓存中。

如果计算的结构允许我们交换循环。

```
for (i=0; i<N; i++) {
     for (loop=0; loop<10; loop++) {
       ... = ... x[i] ...
     }
}
```

x 的元素现在被反复使用，因此更有可能留在缓存中。这个重新排列的代码在使用 $x[i]$ 时显示了更好的时间局部性。

空间局部性

空间局部性的概念要稍微复杂一些。如果一个程序引用的内存与它已经引用过的内存 "接近"，那么这个程序就被认为具有空间局部性。经典的冯·诺依曼架构中只有一个处理器和内存，此时空间局部性并不突出，因为内存中的一个地址可以像其他地址一样被快速检索。然而，在一个有缓存的现代CPU中，情况就不同了。上面我们已经看到了两个空间局部性的例子。

- 由于数据是以缓存线而不是单独的字或字节为单位移动的，因此以这样的方式进行编码，因此使缓存线所有的元素都得到应用是有所裨益的，在下列循环中

```
for (i=0; i<N*s; i+=s){
  ... x[i] ...
}
```

空间局部性体现为函数所进行的跨步递减s。

设 S 为缓存线的大小，那么当 s 的范围从$1....S$，每个缓存线使用的元素数就会从S下降到1。相对来说，这增加了循环中的内存流量花销：如果$s=1$，我们为每个元素加载$1/S$的缓存线；如果$s=S$，我们为每个元素加载一个缓存线。这个效果在1.7.4节中得到了证明。

- 第二个值得注意的空间局部性的例子是TLB（1.3.8.2节）。如果一个程序引用的元素距离很近，它们很可能在同一个内存页上，通过TLB的地址转换会很迅速。另一方面，如果一个程序引用了许多不同的元素，它也将引用许多不同的页。由此产生的TLB缺失是时间花销十分庞大；另见1.7.5节。

练习 1.16 请考虑以下对 n 数字 $x[i]$ 进行求和的算法的伪码，其中 n 是2的倍数。

```
for s=2,4,8,...,n/2,n:
    for i=0 to n-1 with steps s:
        x[i] = x[i] + x[i+s/2]
sum = x[0]
```

分析该算法的空间和时间局部性，并将其与标准算法进行对比

```
sum = 0
for i=0,1,2, ... ,n-1
  sum = sum+ x[i]
```

练习 1.17 考虑以下代码，并假设nvectors相比于缓存很小，而长度很大。

```
for (k=0; k<nvectors; k++)
     for (i=0; i<length; i++)
       a[k,i] = b[i] * c[k]
```

以下概念与该代码的性能有什么关系。

- 复用（Reuse）
- 缓存尺寸（Cache size）
- 关联性（Associativity）

下面这段交换了循环的代码的性能是更好还是更差，为什么?

```
for (i=0; i<length; i++)
  for (k=0; k<nvectors; k++)
    a[k,i] = b[i] * c[k]
```

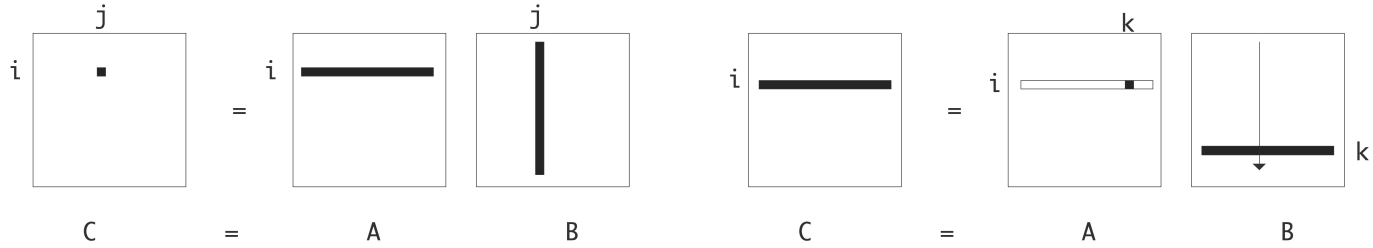
局部性示例

让我们看一个实际的例子。矩阵与矩阵的乘法 $C \leftarrow A \cdot B$可以用几种方法计算。我们比较两种实现方式，假设所有的矩阵都是按行存储的，且缓存大小不足以存储整个行或列。

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] += a[i,k]*b[k,j]
```

```
for i=1..n
  for k=1..n
    for j=1..n
      c[i,j] += a[i,k]*b[k,j]
```

这些实现如图1所示。 第一个实现构建了 (i, j) 元素



A 的一行与 B 的一列的内积来更新 C，在第二行中，B 的一行是通过对 A 的元素进行缩放来更新。A 的元素来更新 B 的行数。

我们的第一个观察结果是，这两种实现都确实计算了 $C \leftarrow C + A \cdot B$，并且它们都花费了大约 $2n^3$ 的操作。然而，它们的内存行为，包括空间和时间的局部性是非常不同的。

- $c[i,j]$ ：在第一个实现中，$c[i,j]$在内部迭代中是不变的，这构成了时间局部性，所以它可以被保存在寄存器中。因此，C 的每个元素将只被加载和存储一次。

  在第二个实现中，$c[i,j]$ 将在每个内部迭代中被加载和存储。特别是，这意味着现在有 n^3 次存储操作，比第一次实现多了n。

- $a[i,k]$：在这两种实现中，$a[i,k]$元素都是按行访问的，所以有很好的空间局部性，因为每个加载的缓存线都会被完全使用。在第二个实现中，$a[i,k]$在内循环中是不变的，这构成了时间局部性；它可以被保存在寄存器中。因此，在第二种情况下，A只被加载一次，而在第一种情况下则是n次。

- $b[k,j]$：这两种实现方式在访问矩阵 B 的方式上有很大不同。首先，$b[k,j]$ 从来都是不变的，所以它不会被保存在寄存器中，而且 B 在两种情况下都会产生 n^3 的内存负载。但是，访问模式不同。

  在第二种情况下，$b[k,j]$ 是按行访问的，所以有很好的空间局部性性：缓存线在被加载后将被完全利用。

  在第一种实现中，$b[k,j]$ 是通过列访问的。由于矩阵的行存储，一个缓存线包含了一个行的一部分，所以每加载一个缓存线，只有一个元素被用于列的遍历。这意味着第一个实现对 B 的加载量要比缓存线长度的系数大。也有可能是TLB的影响。

请注意，我们并没有对这些实现的代码性能做任何绝对的预测，甚至也没有对它们的运行时间做相对比较。这种预测是很难做到的。然而，上面的讨论指出了与广泛的经典CPU相关的问题。

练习 1.18 乘积 $C \leftarrow A \cdot B$ 的实现算法较多。请考虑以下情况。

```
for k=1..n:
    for i=1..n:
        for j=1..n:
            c[i,j] += a[i,k]*b[k,j]
```

分析矩阵 C 的内存流量，并表明它比上面给出的两种算法更糟糕。

核心局部性

上述空间和时间局部性的概念主要是程序的属性，尽管诸如高速缓存线长度和高速缓存大小这样的硬件属性在分析局部性的数量方面发挥了作用。还有第三种类型的局部性与硬件有更密切的联系：「核心局部性」（core locality）。

如果空间上或时间上接近的写访问是在同一个核心或处理单元上进行的，那么代码的执行就会表现出核心局部性。这里的问题是缓存一致性的问题，两个核心在他们的本地存储中都有某个缓存线的副本。如果它们都从该缓存中读取，那就没有问题了。但是，如果他们中的一个对它进行了写操作，一致性协议就会把这个缓存线复制到另一个核心的本地存储中。这需要占用宝贵的内存带宽，所以要避免这种情况。

核心局部性不仅仅是一个程序的属性，而且在很大程度上也是程序的并行执行方式。