对于改善该计算机的存储带宽有很大益处。

图 6-21 将图 6-20 的天花板整合成一个图。核心程序的算术强度决定了优化区域，优化区域反过来又给出了哪些优化手段可以尝试。需要注意的是，对大多数算术强度，计算优化和存储带宽优化都是重叠的。图 6-21 中有三处不同的阴影标记，表示不同的优化策略。例如，核心程序 2 位于右侧的梯形中，表明只工作在计算优化上。核心程序 1 位于中间的平行四边形中，表示两种优化均可尝试。此外，它建议从优化 2 和 4 开始。注意到核心程序 1 的垂直线低于浮点失衡优化，因此优化 1 是没有必要的。如果核心程序落在左下角的三角形区域，则表示只需进行存储优化即可。

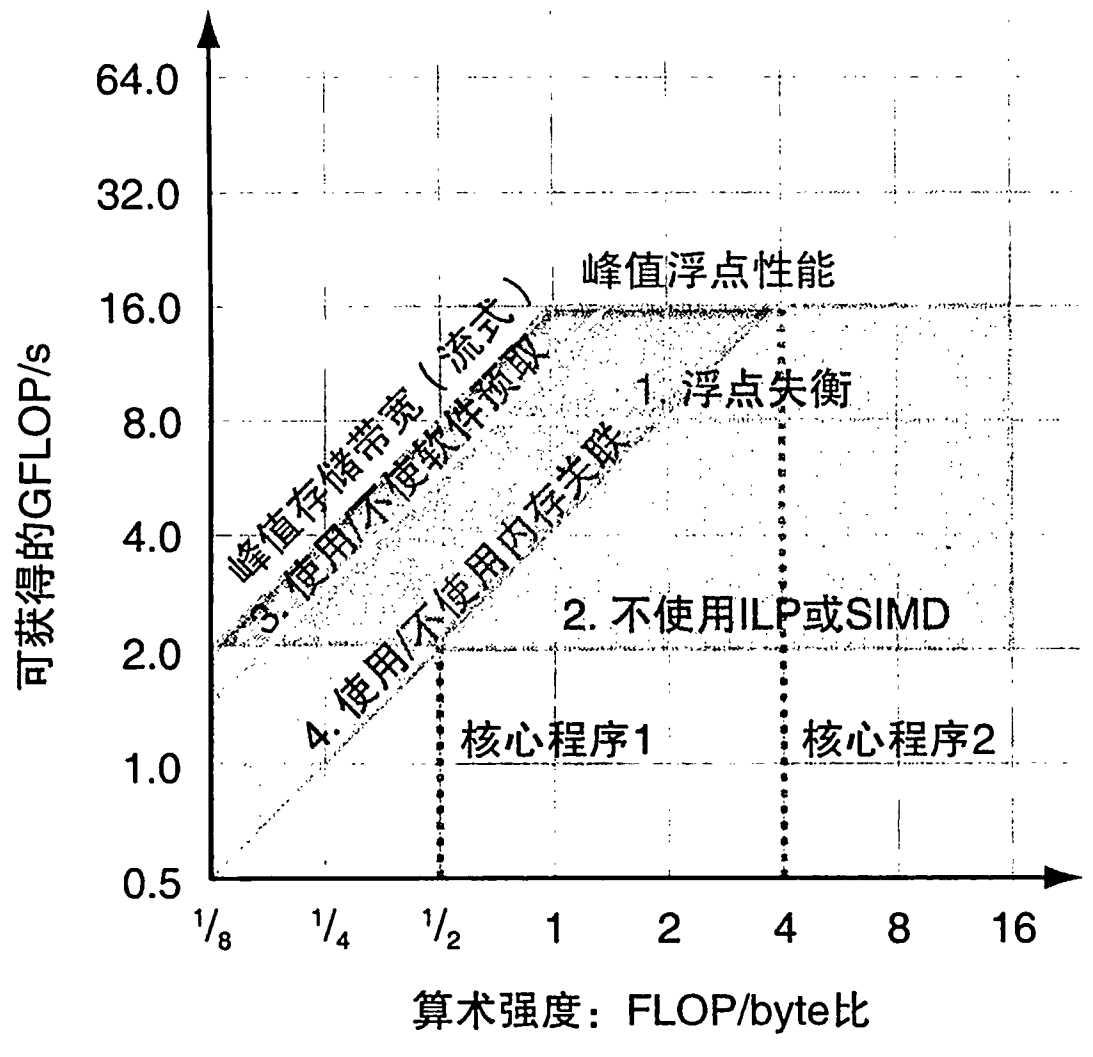

图 6-21　将图 6-18 中两图重叠的 Roofline 模型。算术强度落在右边梯形的核心程序应该着重于计算优化，而处于三角形区域的核心程序应当着重于存储带宽优化。处于平行四边形区域的核心程序两种优化都应当考虑。例如核心程序 1 落在中间的平行四边形中，可尝试优化 ILP 和 SIMD、内存关联、软件预取等。核心程序 2 落在右边的梯形区域，可尝试优化 ILP 和 SIMD 以及浮点运算均衡

到目前为止，我们一直假设算术强度是固定的，但事实并非如此。首先，有些核心程序的算术强度会随问题规模增长，例如密集矩阵和 N 体问题（见图 6-17）。实际上，这就是程序员处理弱比例缩放比强比例缩放更成功的原因之一。其次，存储层次结构的有效性会影响到访存次数，因此提高 cache 性能的优化也会提高算术强度。一个例子是通过展开循环，并将使用相近地址的语句分组在一起来改善时间局部性。许多计算机都有特殊的 cache 指令，故可以先将数据分配到缓存中，而不用先从存储器中填充，因为它可能很快被改写。这两种优化都降低了访存流量，因此可以将算术强度乘以一个系数（如 1.5）以向右移动，这种右移会使核心程序移到一个不同的优化区域。

虽然上面的例子展示了如何帮助程序员提高程序的性能，但同时架构师也可以使用该模型来决定硬件的哪些部分应该优化，以提升他们认为重要的核心程序的性能。

下一节将使用 Roofline 模型来分析多核微处理器和 GPU 的性能差异，并了解这些差异是否反映了真实程序的性能。

详细阐述　天花板是分层次的，最低的天花板是最容易优化的。显然，程序员可以按

任意顺序优化，但是遵从建议的顺序可以避免将时间浪费在因其他约束而无效的优化上。和3C模型类似，只要模型进行了抽象，就会存在一些理想的假设。例如，Roofline模型是假定程序在所有处理器间负载均衡的。

详细阐述 一种替换流式基准测试的方案是使用原始DRAM带宽作为Roofline。虽然原始带宽肯定是一个硬上限，但是存储器的实际性能往往与此相差甚远，因此可用性不高。也就是说，没有程序可以接近该上限。使用流式基准测试的缺点是非常仔细的编程有可能获得超过流的结果，因此内存Roofline不像计算Roofline一样坚实。我们坚持使用流式基准测试是因为很少有程序员能够做到这一点。

详细阐述 尽管Roofline模型适用于多核处理器，但它显然也适用于单处理器。

自我检测 对与错：评测并行计算机的传统方法的主要缺点是确保公平性的同时压制了创新。

6.11 实例：评测Intel Core i7 960和NVIDIA Tesla GPU的Roofline模型

一组Intel研究人员发表了一篇论文[Lee et al.，2010]，比较了带有多媒体SIMD扩展的四核Intel Core i7 960与前一代的GPU（NVIDIA Tesla GTX 280）。图6-22列出了两个系统的特点。这两款产品都是在2009年秋季购买的。Core i7采用Intel的45纳米半导体工艺，而GPU采用TSMC的65nm工艺。虽然由中立机构或对两种产品都感兴趣的机构进行比较可能更公平，但本节的目的不是说明一种产品比另一种产品快多少，而是尝试理解这两种截然不同的结构特性。

	Core i7-960	GTX 280	GTX 480	比值280/i7	比值480/i7
处理单元的数量（核或流式微处理器数）	4	30	15	7.5	3.8
时钟频率（GHz）	3.2	1.3	1.4	0.41	0.44
晶片尺寸	263	576	520	2.2	2.0
工艺	Intel 45nm	TSMC 65nm	TSMC 40nm	1.6	1.0
功耗（芯片，不是模组）	130	130	167	1.0	1.3
晶体管数量	700 M	1400 M	3030 M	2.0	4.4
内存带宽（GByte/s）	32	141	177	4.4	5.5
单精度SIMD宽度	4	8	32	2.0	8.0
双精度SIMD宽度	2	1	16	0.5	8.0
单精度向量运算峰值性能（GFLOP/s）	26	117	63	4.6	2.5
单精度SIMD运算峰值性能（GFLOP/s）	102	311～933	515或1344	3.0～9.1	6.6～13.1
（SP 1中加法器或乘法器）	不支持	(311)	(515)	(3.0)	(6.6)
（SP 1中融合乘加部件）	不支持	(622)	(1344)	(6.1)	(13.1)
（双发射SP包括融合的乘加部件和乘法部件）	不支持	(933)	不支持	(9.1)	—
双精度SIMD运算峰值性能（GFLOP/s）	51	78	515	1.5	10.1

图6-22 Intel Core i7-960、NVIDIA GTX 280和GTX 480的规格。最右边的两列展示了Tesla GTX 280和Fermi GTX 480与Core i7的比值。尽管例子研究的是Tesla 280和i7，但是我们也给出了Fermi 480与Tesla 280的对比，因为本章中介绍到了这点。注意这里的内存带宽比图6-23中的要高，因为这里是DRAM引脚的带宽，图6-23中的是通过测试程序得到的处理器访存的带宽（本表来自Lee et al.[2010]中的表2）

图 6-23 的 Core i7 960 和 GTX 280 的 Roofline 模型展示了两款计算机的区别。GTX 280 不仅具有更高的内存带宽和双精度浮点性能，而且其双精度浮点脊点也更靠左。GTX 280 的双精度浮点性能脊点在 0.6，而 Core i7 的在 3.1。前面曾提到，Roofline 的脊点越靠左，越容易达到峰值性能。对于单精度性能，这两个计算机的脊点都更靠右一些，因此要达到单精度性能的峰值会更难。请注意，算术强度是基于访存的字节数，而不是基于访问 cache 的字节数。因此，就像之前提到的，若大部分访存请求都访问 cache，那么 cache 可以改变特定计算机上的核心程序的算术强度。再次注意，对于这两种结构，我们计算内存带宽时都采用的是单位步长的访问。但是我们即将看到，真实的存储访问模式（聚集 – 分散）在 GTX 280 和 Core i7 上会更慢一些。

通过分析最近提出的四个基准测试程序的计算特性和访存特性，研究人员选择基准测试程序，并制定了计算吞吐量的核心程序集（throughput computing kernels）以捕获这些特征。图 6-24 展示了性能结果，数字越大表明速度越快。Roofline 模型帮助说明了本例中的相对性能。

原始的性能配置参数变化范围很大，GTX 280 的时钟频率是 Core i7 的 1/2.5，而核数是 Core i7 的 7.5 倍。同时性能评测结果变化范围也很大，对于 Solv 程序，GTX 280 是 Core i7 的 1/2，而对于 GJK 程序，GTX 280 比 Core i7 快 15.2 倍，Intel 的研究人员决定找出其中的原因。

- 内存带宽。GPU 的内存带宽是 Core i7 的 4.4 倍，这解释了为什么 GPU 运行 LBM 和 SAXPY 的速度比 Core i7 快 5.0 和 5.3 倍；由于这些程序的工作集是几百兆字节，因此数据不能在 Core i7 的 cache 中全放下（为了密集地访存，他们特意不使用第 5 章中的 cache 分块技术）。因此，Roofline 模型的斜率反映了它们的性能（这是 log-log 图的斜率）。SpMV 也有一个很大的工作集，但 GTX 的运行速度只比 Core i7 快了 1.9 倍，这是因为 GTX 280 的双精度浮点运算性能只是 Core i7 的 1.5 倍。
- 计算带宽。剩下的核心程序中有 5 个是受限于计算强度的：SGEMM、Conv、FFT、MC 和 Bilat。GTX 运行这 5 个程序时分别比 Core i7 快 3.9、2.8、3.0、1.8 和 5.7 倍。其中前三个是单精度浮点运算，GTX 280 单精度浮点运算比 Core i7 快 3 ～ 6 倍。MC 是双精度浮点运算，因为 GTX 280 的双精度浮点运算性能仅比 Core i7 快 1.5 倍，这解释了为什么它只比 Core i7 快 1.8 倍。Bilat 使用 GTX 280 直接支持的超越函数。而 Core i7 执行 Bilat 时，三分之二的时间用来计算超越函数，所以 GTX 280 比 Core i7 要快 5.7 倍。这些观察指出由硬件支持工作负载的特定操作（双精度浮点运算、超越函数）很有意义。
- 从 cache 获得的好处。GTX 上运行 Ray Cast（RC）只比 Core i7 快 1.6 倍，这是因为 Core i7 使用的 cache 分块技术使得 RC 不受内存带宽限制（见 5.4 节和 5.14 节），在 GPU 上也不受内存带宽限制。cache 分块技术还有助于 Search 程序。如果索引树小到可以在 cache 中装下，那么 Core i7 的速度会是 GTX 的 2 倍。较大的索引树会使程序受限于内存带宽。总体而言，GTX 280 在运行 Search 程序时比 Core i7 快 1.8 倍。cache 分块技术也有助于 Sort 程序。虽然大多数程序员不会在 SIMD 处理器上运行 Sort，但可以用“split”的 1 位 Sort 原语编写。split 算法执行的指令数远多于标量 Sort。所以，Core i7 运行 Sort 程序的速度是 GTX 280 的 1.25 倍。注意，cache 分块技术也使得 SGEMM、FFT 和 SpMV 成为受限于计算的程序，所以 cache 对于 Core i7 上的其他核心程序也有帮助。这一观察再次强调了第 5 章中 cache 分块技术的重要性。

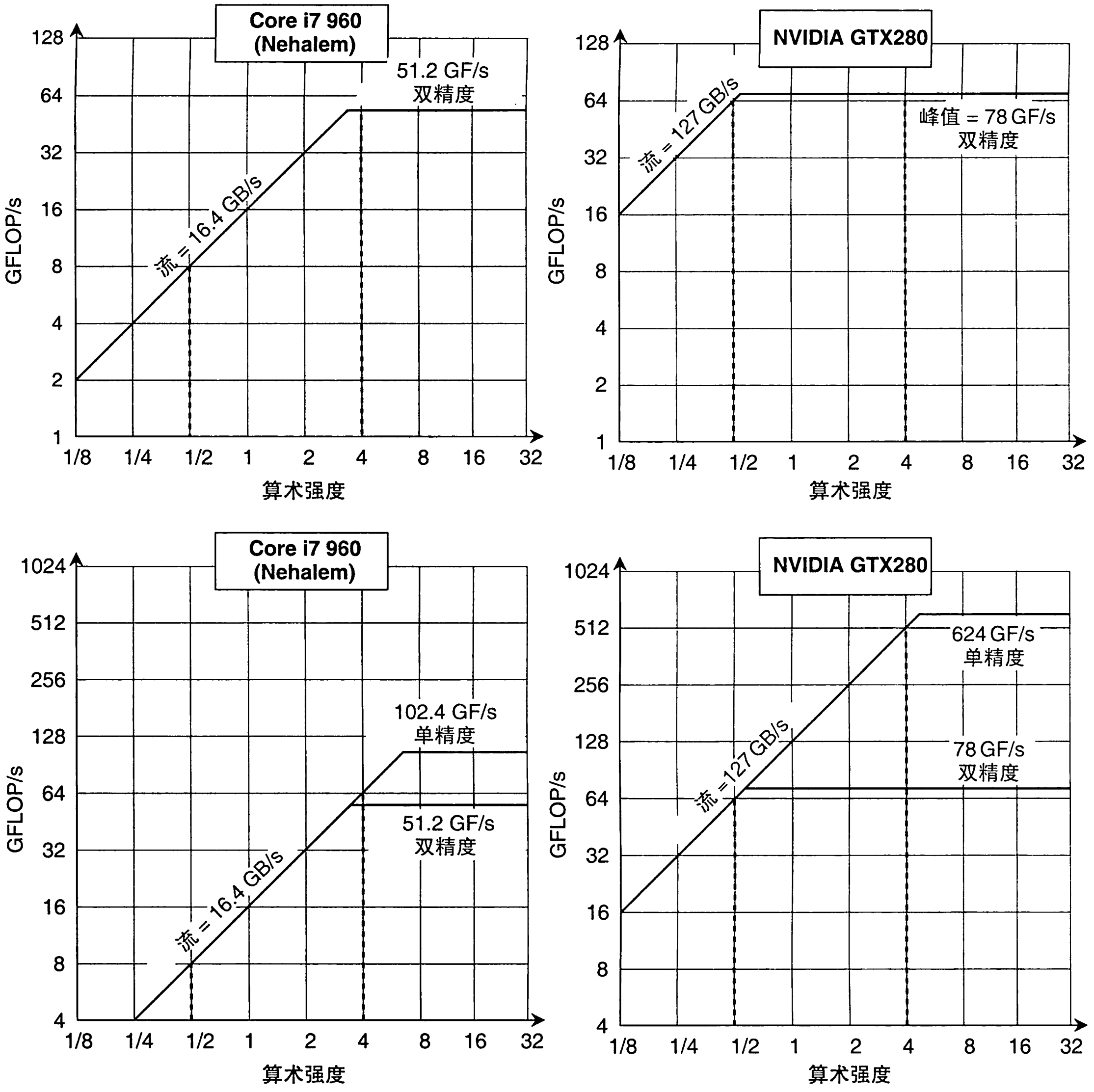

图 6-23 Roofline 模型 [Williams，Waterman，and Patterson，2009]。前两张图给出了双精度浮点性能，后面两张图给出了单精度浮点性能。（双精度浮点性能的 Roofline 同时也在单精度浮点性能的图中画出了，并给出对比。）左边 Core i7 960 的双精度浮点性能峰值为 51.2GFLOP/s，单精度浮点性能峰值为 102.4GFLOP/s，内存带宽峰值为 16.4GB/s。NVIDIA GTX 280 的双精度浮点性能峰值为 78GFLOP/s，单精度浮点性能峰值为 624GFLOP/s，内存带宽峰值为 127GB/s。图中左侧的垂直虚线表示 0.5FLOP/byte 的算术强度。Core i7 的浮点性能受内存带宽的限制不超过 8GFLOP/s（对于双精度浮点和单精度浮点都如此）。图中右侧的垂直虚线表示 4FLOP/byte 的算术强度，在 Core i7 上限制在 51.2GFLOP/s（双精度浮点性能）到 64GFLOP/s（单精度浮点性能）之间，在 GTX 280 上限制在 78GFLOP/s（双精度浮点性能）到 624GFLOP/s（单精度浮点性能）之间。为了在 Core i7 上达到最高的运算率，需要使用全部 4 个核和 SSE 指令，并且乘法器和加法器的数量相等。对于 GTX 280，你需要在所有多线程的 SIMD 处理器上使用混合的乘加指令

物理程序	单位	Core i7 960	GTX 280	GTX 280 / i7 960
SGEMM	GFLOP/s	94	364	3.9
MC	十亿条路径/秒	0.8	1.4	1.8
Conv	百万像素/秒	1250	3500	2.8
FFT	GFLOP/s	71.4	213	3.0
SAXPY	GByte/秒	16.8	88.8	5.3
LBM	百万次查找/秒	85	426	5.0
Solv	帧/秒	103	52	0.5
SpMV	GFLOP/s	4.9	9.1	1.9
GJK	帧/秒	67	1020	15.2
Sort	百万个元素/秒	250	198	0.8
RC	帧/秒	5	8.1	1.6
Search	百万次查询/秒	50	90	1.8
Hist	百万像素/秒	1517	2583	1.7
Bilat	百万像素/秒	83	475	5.7

图 6-24 两个平台测量得到的原始和相对性能数据。在这项研究中，SAXPY 只用来对内存带宽进行测量，所以右边的单位是 GByte/s 而不是 GFLOP/s（基于 Lee et al.[2010] 中的表 3）

- 聚集 – 分散。如果数据分散在内存中的各个地方，多媒体 SIMD 扩展几乎没有什么用途。只有访问的数据是 16 字节对齐时才会获得最佳性能。因此，GJK 在 Core i7 上从 SIMD 中获得的好处很少。就像前面提到过的，GPU 提供的向量结构支持聚集 – 分散技术，而大部分 SIMD 扩展不支持。内存控制器会成批地访问同一个 DRAM 页的请求（见 5.2 节）。这两点使得 GTX 280 运行 GJK 的速度是 Core i7 的 15.2 倍，这个数字比图 6-22 中的任何物理参数都要大。这一观察强调了聚集 – 分散技术对于向量结构和 GPU 结构的重要性（而 SIMD 扩展还不支持）。
- 同步。尽管 Core i7 有硬件取改写（fetch-and-increment）原子指令，但原子更新仍然占据了 Core i7 总执行时间的 28%，所以同步的性能受到原子更新的限制。因此，Hist 程序在 GTX 280 上只比 Core i7 快 1.7 倍。Solv 程序是一个约束求解问题，它通过在一段简单的计算之后插入同步操作来实现。尽管不是所有之前的访存指令都执行结束，我们也可以保证结果是正确的，所以 Core i7 可以从原子存储一致性模型和原子指令中获益。由于没有存储一致性模型，当 GTX 280 从系统处理器发出一批操作时，它的性能只是 Core i7 的 0.5 倍。这一观察指出了同步的性能对一些数据并行问题的重要性。

由 Intel 研究人员选择的这些核心程序所发现的 Tesla GTX 280 的不足，在 Tesla 的后继版本中不断得到解决，改进效率令人震惊：Fermi 有更高的双精度浮点性能、更快的原子操作和更快的 cache。同样有趣的是，对向量结构支持聚集 – 分散机制早于 SIMD 指令几十年就出现了，并且有些人在这次比较之前已经预测到聚集 – 分散技术对 SIMD 扩展的有效执行非常重要。Intel 研究人员指出，如果对 Core i7 提供足够的聚集 – 分散支持，14 个核心程序中的 6 个可以更有效地利用 SIMD。本次比较当然也证实了 cache 分块技术的重要性。

现在我们看到了评测不同多处理器得出的很多结果，接下来看看我们需要对 DGEMM 程序的 C 代码进行多大的修改才可以发挥多处理器的性能优势。

6.12　加速：多处理器和矩阵乘法

本节我们将调整 DGEMM 使其适于 Intel Core i7（Sandy Bridge）底层硬件，这是性能提升的最后一个步骤，也是优化效果最显著的步骤。每个 Core i7 有 8 个核，我们使用的计算机有 2 个 Core i7。所以我们有 16 个核来运行 DGEMM。

图 6-25 给出了使用这 16 个核的 OpenMP 版本的 DGEMM 程序。注意，第 30 行是相对于图 5-47 增加的唯一一行代码，这行代码使程序可以运行在多处理器上。OpenMP 的 pragma 语句告诉编译器对最外层 for 循环使用多线程技术。这样，计算机会将最外层 for 循环的任务分配给所有线程去执行。

```
 1 #include <x86intrin.h>
 2 #define UNROLL (4)
 3 #define BLOCKSIZE 32
 4 void do_block (int n, int si, int sj, int sk,
 5                 double *A, double *B, double *C)
 6 {
 7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
 8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
 9       __m256d c[4];
10       for ( int x = 0; x < UNROLL; x++ )
11         c[x] = _mm256_load_pd(C+i+x*4+j*n);
12       /* c[x] = C[i][j] */
13       for( int k = sk; k < sk+BLOCKSIZE; k++ )
14       {
15         __m256d b = _mm256_broadcast_sd(B+k+j*n);
16       /* b = B[k][j] */
17         for (int x = 0; x < UNROLL; x++)
18           c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                  _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20       }
21
22       for ( int x = 0; x < UNROLL; x++ )
23         _mm256_store_pd(C+i+x*4+j*n, c[x]);
24         /* C[i][j] = c[x] */
25     }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32     for ( int si = 0; si < n; si += BLOCKSIZE )
33       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34         do_block(n, si, sj, sk, A, B, C);
35 }
```

图 6-25　图 5-47 中 DGEMM 的 OpenMP 版本。第 30 行是唯一一条 OpenMP 语句，它使最外层的 for 循环并行执行。这一行代码是图 6-25 与图 5-47 唯一的不同之处

图 6-26 画出了一个经典的多处理器加速比图，它展示了随着处理器的线程数增加，其性能相对于单线程的提升。从图中很容易看出，强比例缩放比弱比例缩放面临更大的挑

战。如果所有数据都可以放入一级数据 cache 中，增加线程数实际上会降低性能，例如对于 32 × 32 矩阵，16 个线程的性能只是单线程的一半。相反，如图 6-26 最上面的两条线所示，最大的两个矩阵在使用 16 个线程时，性能提升了 14 倍。

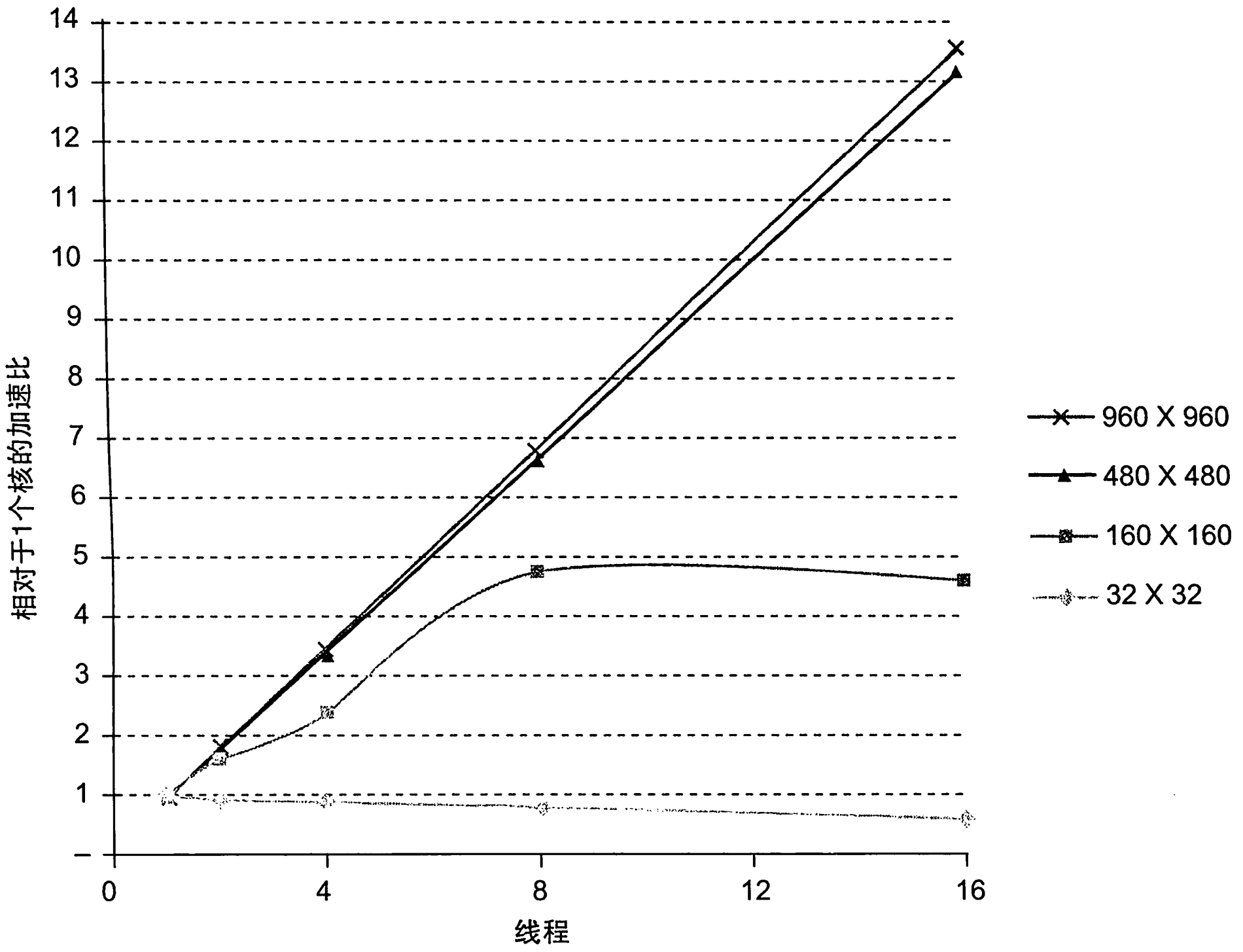

图 6-26　与单线程相比，当线程数增多时的性能提升。我们采用的是最客观的方法：将多线程的性能与最优的单线程性能相比。与本图做对比的是图 5-47 中没有使用 OpenMP 的 pragma 语句的代码

当线程数从 1 个增加到 16 个，图 6-27 给出了绝对性能增长。对于 960960 的矩阵，DGEMM 程序的运行速度是 174GFLOPS。而图 3-22 给出的未经任何优化的 C 语言 DGEMM 程序的运行速度是 0.8GFLOPS，也就是说通过第 3 ～ 6 章根据底层硬件对代码进行的优化，性能提升了 200 多倍！

接下来我们将给出多进程的谬误和陷阱。正是因为忽略了这些谬误和陷阱，计算机体系结构中的很多并行项目遭受了失败。

详细阐述　上述这些结论是在关闭 Turbo 模式的情况下得到的。在这个系统中，我们使用的是双芯片，所以如果打开 Turbo 模式，无论我们使用 1 个线程（只使用一个芯片上的一个核）还是 2 个线程（每个芯片上各使用一个核），都能得到完整的 Turbo 加速（3.3/2.6=1.27）。但是当我们增加线程数时（活跃的核数也增加），从 Turbo 模式获得的收益将减少，因为消耗在这些核上的功耗变少了。对于 4 线程、8 线程和 16 线程，平均 Turbo 加速比分别为 1.23、1.13 和 1.11。

详细阐述　虽然 Sandy Bridge 支持每个核 2 个硬件线程，但当使用 32 个线程时，一个 AVX 硬件被同一个核上的两个线程所共享，我们无法得到更多的性能提升。所以为每个核分配 2 个线程会引入复用核的代价，使性能降低。

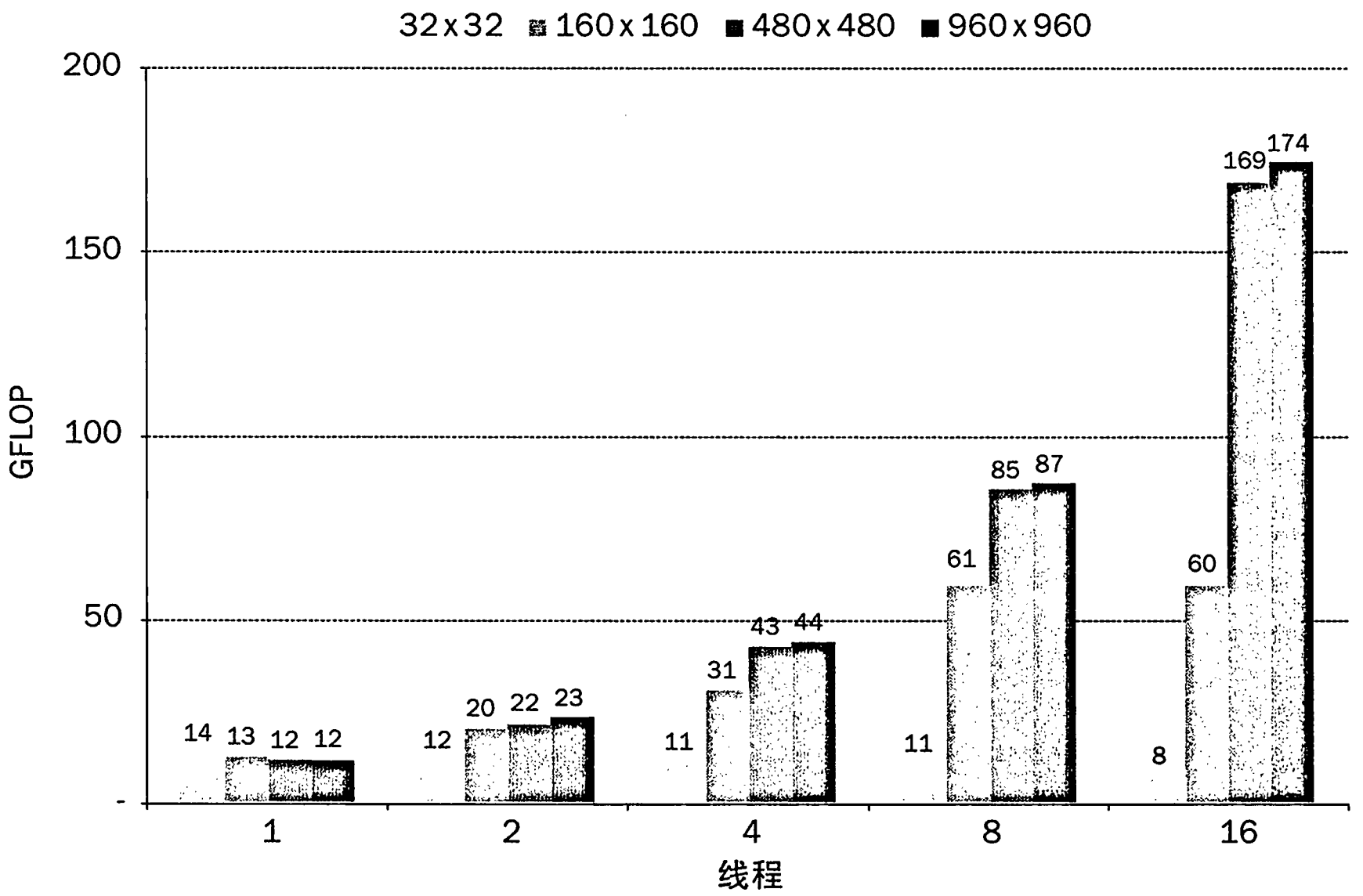

图 6-27 对于 4 个不同大小的矩阵，运行在多线程处理器上的 DGEMM 程序的性能。对于 960 × 960 矩阵，与图 3-21 中的未经任何优化的代码相比，在使用 16 个线程时，性能提升了 212 倍

6.13 谬误与陷阱

对并行处理的许多争论中揭示了诸多谬误和陷阱。这里列举其中的四个。

谬误：并行计算机不遵循 Amdahl 定律。

1987 年，一个研究组织的负责人声称多处理器不遵循 Amdahl 定律。为了理解这个报道的理论基础，我们看看报道中对 Amdahl 定律的解释 [1967, p. 483]：

> 此时可以得出一个明显的结论：提高并行处理的速度所付出的代价很可能是种浪费，除非同时将顺序处理的速度提高到相近的数量级。

这种说法的确是真实的，被忽略的那部分程序必然限制性能。对 Amdahl 定律的一种解释可得到下面的引理：每个程序都有一部分是顺序的，因此必然有一个经济的处理器数量上限，比如 100。通过给出 1000 个处理器也可以达到线性增长，证明该引理是错误的，因而得出了 Amdahl 定律被打破的结论。

> 十多年来，一些先知激烈地争论着，认为单处理器的组成方式已经达到了性能极限，只能通过将多台计算机互连以支持协同工作，来获得性能的真正改进……事实证明单处理器提升性能的方法一直在生效……
>
> *Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", Spring Joint Computer Conference, 1967*

这些研究人员的方法是使用弱比例缩放：不是在相同的数据集上将速度提高 1000 倍，而是在可比较时间内将计算量提高 1000 倍。但是对于他们的算法，程序中顺序执行的比例是常数，与问题的输入规模无关。而其余部分则是完全并行的，因此，使用 1000 个处理器时依然为线性增长。

Amdahl 定律显然适用于并行处理器。这项研究却指出了更快的计算机的主要用途之一是完成更大规模的问题。只要确保用户真的关心这些问题，而不是为了使很多处理器保持运转而购买更贵的计算机。

谬误：峰值性能可代表实际性能。

超级计算机业界在市场中曾经使用该度量方法，并且该谬误在并行机中更加严重。市场营销人员不仅在单处理器节点使用几乎无法达到的峰值性能，而且还将它与处理器总数相乘，假定并行机可以达到完美的加速！ Amdahl 定律已经指出达到任一个峰值是多么困难，将两者相乘就更是错上加错。Roofline 模型有助于正确看待峰值性能。

陷阱：不开发软件来利用和优化多处理器体系结构。

在很长一段时间里并行软件的发展一直落后于并行硬件，可能是因为软件的问题更加困难。我们用一个例子说明问题的微妙之处，其实可供选择的例子还有很多！

把为单处理器设计的软件移植到多处理器环境时，经常会遇到这样一个问题。例如，Silicon Graphics 操作系统最初假定页的分配不频繁，只通过一个锁来保护页表。在单处理器中，这不会引起性能问题。但在多处理器中，这种设计对某些程序会成为主要的性能瓶颈。正如 UNIX 为静态分配的页所做的事情，程序在启动时需要初始化大量的页。假设该程序被并行化了，所以需要为多个进程分配页。由于页的分配需要使用页表，而页表在每次使用时被上锁。如果多个进程试图同时请求分配页，即使操作系统内核支持多线程，这些请求也会被串行执行（这就是我们在初始化时所预期的情况）。

页表请求的串行化处理影响了初始化时的并行性，并对整个并行性能产生很大的影响。这种性能瓶颈也会出现在任务级并行中。例如，假设我们将并行处理程序分为若干个独立的作业，并在每个处理器上运行一个作业，不同作业之间没有任何共享。（这恰好是一个用户的做法，因为他合乎情理地相信性能问题是由于应用程序中无意的共享或冲突所造成的。）不幸的是，锁机制依然将所有工作串行化——即使对于互相独立的工作，性能也会很低。

该陷阱说明软件运行在多处理器上时，这种虽然微小但对性能有极大影响的缺陷会显现出来。像许多其他关键软件一样，多处理器上的操作系统算法和数据结构需要重新考虑。对页表的更小单位加锁可以有效地避免这个问题。

谬误：在不提升内存带宽的前提下可以得到不错的向量计算性能。

从 Roofline 模型中可以看到，内存带宽对所有体系结构都非常重要。DAXPY 的每个浮点运算需要 1.5 次访存，对于很多科学计算，这是一个很标准的比例。即使浮点运算不花费时间，Cray-1 计算机也无法提升 DAXPY 向量计算的性能，因为它受到内存限制。若编译器采用分块技术，使数据可以保存在向量寄存器中，Cray-1 运行 Linpack 的性能有了跳跃式提升。这个方法降低了每个浮点运算的访存次数并使性能提升了将近两倍！这样，对于之前有高带宽需求的循环，Cray-1 的内存带宽就足够了，这正是 Roofline 模型所预言的。

6.14　本章小结

> 我们正在将未来产品的开发专注于多核设计。我们相信这对工业界是一个重要转折点……这不是一场竞争，这是计算的翻天覆地的变化……
> *Paul Otellini，Intel 总裁，Intel 开发者论坛，2004*

从计算机发展的早期开始，人们就梦想着通过简单地集成若干处理器来构建计算机。但是构建并充分有效地利用并行处理器的过程是缓慢的。一方面是受软件困难的限制，另一方面是，为了提高可用性和效率，多处理器体系结构在不断改进。本章中我们讨论了很多软件方面的挑战（例如，如何编写根据 Amdahl 定律可获得高加速比的程序）。不同的体系结构之间存在巨大差异，而且过去很多并行体系结构的生命周期非常短暂，所取得的性能提升也非常有限，这些因素使得软件更加困难。网上的 6.15 节讨论了这些多处理器的历史。要对本章所述的主

题有更深入的理解，请参阅《计算机体系结构：量化研究方法》（第 5 版）第 4 章中更多关于 GPU 以及 CPU 和 GPU 之间对比的内容，还有第 6 章关于 WSC 的内容。

正如第 1 章所述，历史漫长而坎坷，但是现在信息技术业的未来与并行计算紧密联系在一起。像过去一样，尽管可能会失败，依然有很多理由让我们对并行充满希望：

- 显然，软件即服务（SaaS）的重要性日益增加，并且集群已被证明能很好地提供这种服务。通过提供高层次的冗余（包括地理分布的数据中心），此类服务可以为全球的客户提供 24 × 7 × 365 的可用性。
- 我们相信仓储级计算机（Warehouse-Scale Computer，WSC）正在改变服务器设计的目标和原则，就像移动客户端的需求正在改变微处理器设计的目标和原则一样。这两者同样也在革新软件行业。每美元的性能和每焦耳的性能驱动着移动客户端硬件和 WSC 硬件的发展，并行是实现这些目标的关键。
- 在后 PC 时代多媒体应用程序扮演着更重要的角色，而 SIMD 和向量操作更适合多媒体应用程序。与经典的并行 MIMD 编程相比，SIMD 和向量运算对程序员来说更简单，而且能效性也更好。为了认识到 SIMD 与 MIMD 的重要性，图 6-28 给出了随着时间的增长 MIMD 中的核数，以及 x86 计算机中 SIMD 模式下每时钟周期的 32 位和 64 位操作个数。对于 x86 计算机，我们预计每两年芯片增加两个核，并且每四年 SIMD 宽度增加一倍。基于这些假设，在未来十年内，SIMD 的并行加速会是 MIMD 并行的两倍。由于 SIMD 对多媒体应用程序非常有效，并且多媒体应用程序在后 PC 时代日益重要，这个推论可能是正确的。因此，理解 SIMD 并行与 MIMD 并行一样重要，尽管后者已经得到了更多的关注。

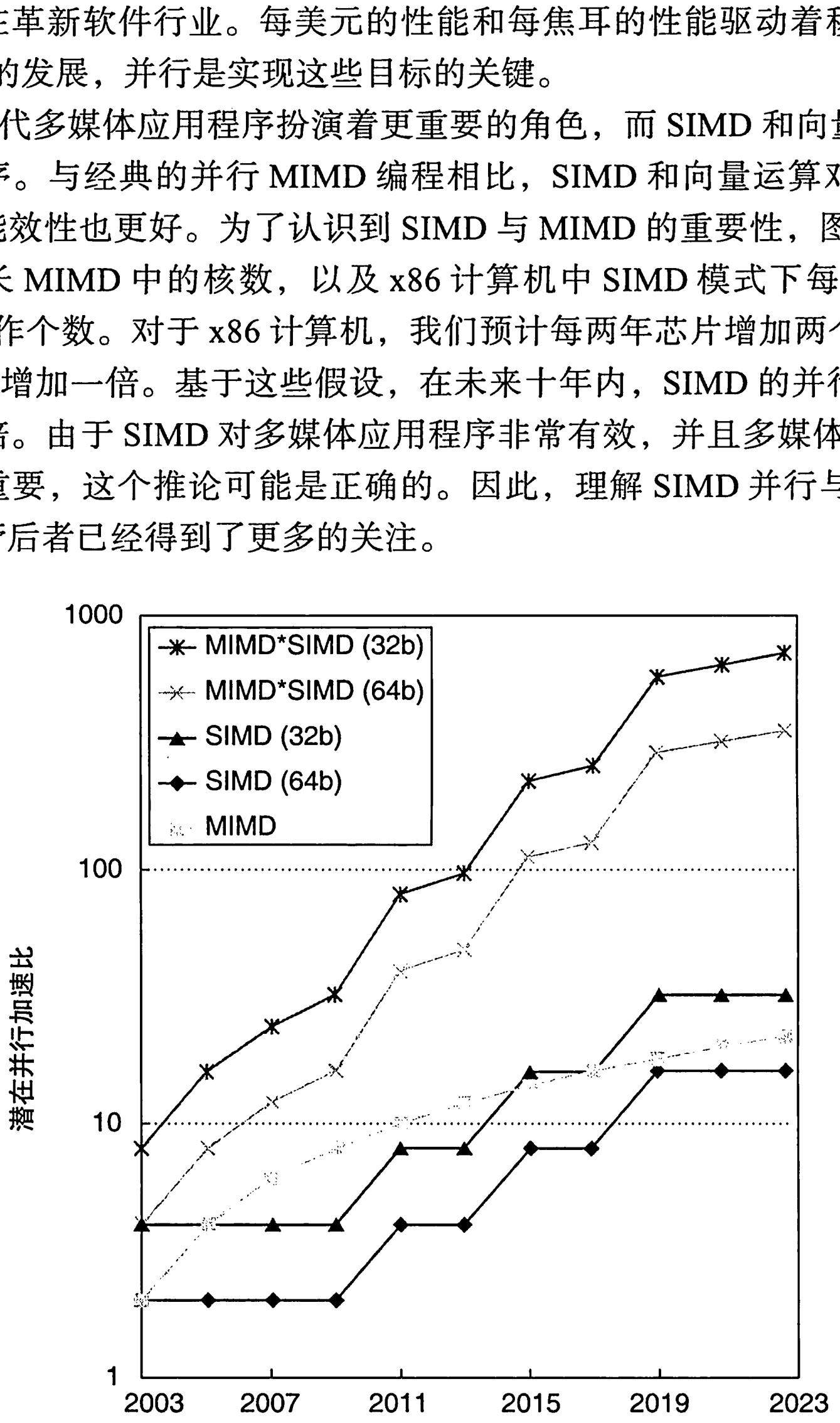

图 6-28　MIMD 和 SIMD 的潜在并行加速比，以及 x86 计算机的 MIMD 和 SIMD 随着时间的变化。该图假设每两年芯片上 MIMD 的核数增加两个，并且每四年 SIMD 操作的数目增加一倍

- 在科学和工程计算的领域中并行处理被广泛使用。这类应用程序对计算能力的需求几乎没有上限。还有一些应用程序存在大量天然的并行性。集群再次占据了这类应用程序。例如，在 2012 年 Top 500 报告中，集群占了 500 个最快的 Linpack 的 80%。
- 为了获得高性能，所有桌面和服务器微处理器制造商都在构建多处理器，顺序应用程序不会像过去一样再有获取高性能的途径。正如我们之前所说，顺序程序现在的速度很慢。因此，需要更高性能的程序员必须将自己的代码并行化，或编写新的并行处理程序。
- 过去，微处理器和多处理器对于成功的定义是不同的。当扩展单处理器性能时，如果单线程性能随芯片面积的平方根增长，微处理器设计者会感到很满意。也就是说，他们满足于性能随资源数量的亚线性增长。多处理器的成功在过去被定义为与处理器数量相关的线性加速比函数，并假定 n 个处理器的购买成本或管理成本是单处理器的 n 倍。既然并行发生在芯片上的多核之间，我们可以使用已经获得成功的传统微处理器标准来获得亚线性的性能提升。
- 即时编译和自动运行的成功使得软件更容易适应每芯片核数的增长，提供了静态编译器所不能提供的灵活性。
- 与过去不同，开源运动已成为软件行业的一个关键部分。这种运动可以改善工程解决方案，促进开发者之间的知识共享。它也鼓励创新，在改变旧软件时欢迎新的语言和软件产品。这种开放式的文化必将有益于目前日新月异的时期。

为了使读者接受这场革命，我们通过快速浏览第 3 ～ 6 章来展示如何通过 Intel Core i7（Sandy Bridge）处理器发掘矩阵乘法的潜在并行：

- 第 3 章的数据级并行通过使用 256 位的 AVX 指令并行执行 4 个 64 位浮点运算，使性能提高了 3.85 倍，证明了 SIMD 的价值。
- 第 4 章通过 4 次展开循环发掘指令级并行，为乱序执行的硬件提供了更多的指令去调度，这又使性能提升了 2.3 倍。
- 第 5 章的 cache 优化使用分块技术来减少 cache 失效，这将不能放进 L1 cache 的矩阵性能提升了 2.0 ～ 2.5 倍。
- 本章通过使用多核芯片上的所有 16 个核，利用了线程级并行，将无法放入 L1 cache 矩阵的性能提高了 4 ～ 14 倍，证明了 MIMD 的价值。我们是通过添加一行 OpenMP pragma 语句实现的。

使用本书中的方法并根据该计算机对软件进行改造，为 DGEMM 程序添加了 24 行代码。对于 32×32、160×160、480×480 和 960×960 的矩阵，通过这几行代码和本书的方法得到的总性能加速比为 8、39、129 和 212！

硬件 / 软件接口的并行变革可能是过去 60 年来所面临的最大挑战。正如我们快速浏览各章节时所展示的，你也可以将其视为一个绝佳的机会。这场变革在 IT 界内外提供了大量新的研究和商业前景，并且使专注于多核和单处理器的公司区分开来。在理解了底层硬件的发展趋势，以及学会了如何使软件适应它们之后，也许你就会抓住其中的机会，成为创新者中的一员。我们期待从你的发明创造中受益！

6.15　历史视角和拓展阅读

本节在网上主要给出了近 50 年来多处理器的发展历史。

参考文献

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with YCSB, In: Proceedings of the 1st ACM Symposium on Cloud computing, June 10–11, 2010, Indianapolis, Indiana, USA, doi:10.1145/1807128.1807152.

G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. IEEE Computer, 37(11):48–58, 2004.

6.16 练习

6.1 首先写一个每天你通常需要完成的日常活动的列表。例如，你可能会起床、淋浴、穿衣服、吃早饭、弄干头发、刷牙。确保列表中至少包含 10 项活动。

6.1.1 [5] <6.2> 考虑哪些活动已经利用了某种形式的并行性（例如，是同时刷多颗牙还是一次只刷一颗牙，是一次只带一本书到学校，还是将所有书装到背包里一次“并行”携带）。对每个活动都分析是否已经并行工作，如果没有，分析其原因。

6.1.2 [5] <6.2> 接下来考虑哪些活动可以并发执行（例如，吃早餐和听新闻）。对每个活动都分析哪些活动可以与其配对并发执行。

6.1.3 [5] <6.2> 对于上题，可以通过改变现有系统（例如，淋浴设备、衣服、电视机、汽车等）中的什么让我们并行执行更多的任务？

6.1.4 [5] <6.2> 如果你能尽可能多地并行执行任务，估计完成这些任务可以缩短的时间是多少？

6.2 假设需要你制作 3 块蓝莓蛋糕。蛋糕的配料如下：

- 1 杯黄油，软化后再用
- 1 杯糖
- 4 个大鸡蛋
- 1 茶匙香草精
- 0.5 茶匙盐
- 0.25 茶匙肉豆蔻
- 1.5 杯面粉
- 1 杯蓝莓

蛋糕的制作流程如下：

- 第 1 步：烤箱预热至 160（325）。在烤盘上抹黄油和一层薄薄的面粉。
- 第 2 步：在一只大碗中使用搅拌器以中速将奶油和糖混合在一起，直到变成稀松的糊状。再加鸡蛋、香草精、盐和肉豆蔻，搅拌到完全混合。将搅拌器降到低速，一次加入 0.5 杯面粉，搅拌到完全混合。
- 第 3 步：最后慢慢加入蓝莓，将蛋糕均匀地放在烤盘中，烘烤约 60 分钟。

6.2.1 [5] <6.2> 你的任务是尽可能高效率地完成 3 块蛋糕。假定只有一个能容纳一块蛋糕的烤箱、一个大碗、一个烤盘、一个搅拌器，请做出合理的调度以尽可能快地完成任务，并分析瓶颈所在。

6.2.2 [5] <6.2> 假设你现在有 3 个碗、3 个蛋糕盘子和 3 个搅拌器。你拥有这些增加的资源后，现在的工序加快了多少？

6.2.3 [5] <6.2> 假设现在有两个朋友可帮你烹饪，并且你有一个可容纳 3 个蛋糕的大烤箱。这些将使 6.2.1 中的计划有何改变？

6.2.4 [5] <6.2> 将制作蛋糕与并行计算机中的循环迭代进行类比。分析制作蛋糕的循环中存在的数据级并行和任务级并行。

6.3 许多计算机应用程序需要在一组数据中进行搜索和对数据进行排序。为了减少这些任务的执行时间，几种高效的搜索和排序算法已被设计出来。在本练习中，我们考虑如何尽可能地将这些任务并行化。

6.3.1 [10] <6.2> 请看下面的二进制搜索算法（一种经典的分治算法），该算法可以在已经排序的 N 元素数组 A 中搜索值 X，并返回匹配项的索引号：

```
BinarySearch(A[0..N-1], X) {
    low = 0
    high = N -1
    while (low <= high) {
        mid = (low + high) / 2
        if (A[mid] >X)
            high = mid -1
        else if (A[mid] <X)
            low = mid + 1
        else
            return mid // found
    }
    return -1 // not found
}
```

假设 BinarySearch 运行在具有 Y 个核的多核处理器上，且 Y 远远小于 N。请问对于 Y 和 N 的不同取值，预期的加速比是多少？请画图表示。

6.3.2 [5] <6.2> 接下来，假设 Y 与 N 相等，这会对你前面的结论有何影响？如果要求你获得尽可能高的加速比（例如，强比例缩放），请问该如何修改代码？

6.4 请看下面的 C 代码片段：

```
for (j=2;j<=1000;j++)
    D[j] = D[j-1]+D[j-2];
```

与之对应的 RISC-V 代码如下所示：

```
        li      x5, 8000
        add     x12, x10, x5
        addi    x11, x10, 16
LOOP:   fld     f0, -16(x11)
        fld     f1, -8(x11)

        fadd.d  f2, f0, f1
        fsd     f2, 0(x11)
        addi    x11, x11, 8
        ble     x11, x12, LOOP
```

一条指令的延迟是指，在这条指令与使用其结果的指令之间必须存在的周期数。假设浮点指令的延迟如下（周期数）：

fadd.d	fle	fsd
4	6	1

6.4.1 [10] <6.2> 执行这段代码需要多少时钟周期？

6.4.2 [10] <6.2> 重新对这段代码排序以减少停顿（提示：通过改变 fsd 指令的偏移量，可以消除由其带来的停顿）。现在，执行这段代码需要多少时钟周期？

6.4.3 [10] <6.2> 在循环中，如果后一次迭代的指令会依赖于前一次迭代的指令产生的结果，我们说

循环迭代之间存在循环进位相关性（loop-carried dependence）。请分析上面代码中的循环进位相关性，识别其中相关的程序变量和汇编级寄存器。可忽略循环变量 j。

6.4.4 [15]<6.2>重写这段代码，用寄存器来传递迭代之间的数据（而不是从内存中读写该数据）。指出重写后的这段代码哪里会发生停顿，并计算执行这段代码所需的时钟周期数。注意，关于这个问题，你可能需要使用汇编伪指令 fmv.d rd, rs1，该指令将浮点寄存器 rs1 的值写入浮点寄存器 rd。假设 fmv.d 指令在一个周期内完成。

6.4.5 [10]<6.2>第 4 章描述了循环展开。对上述代码进行循环展开并优化，展开后的每次迭代处理原来循环的三次迭代。指出展开后的代码在哪里停顿，并计算执行这段代码需要多少时钟周期。

6.4.6 [10]<6.2>由于我们刚好想要多个三次迭代的组合，上一题循环展开后的代码可以工作得很好。但是如果迭代次数在编译时无法得知呢？如何处理这样的循环：原始的迭代次数不是展开后循环的迭代次数的整数倍？

6.4.7 [15]<6.2>考虑将上述代码运行在一个 2 节点的基于消息传递的分布式存储系统中。假定我们采用 6.7 节描述的消息传递机制，操作 send (x, y) 向节点 x 发送值 y，操作 receive () 等待正在发送的数。再假定 send 操作的发射需要 1 个周期（也就是说，同一节点的后续指令可在下个周期执行），而接收节点需要多个周期接收。接收指令会阻塞接收节点上指令的执行，一直等到接收节点完成消息接收为止。你能用这样一个系统为上述代码加速吗？如果可以，能容忍的最大接收延迟是多少？如果不能，请说明理由。

6.5 考虑下面的归并排序算法（另一种经典的分治算法）。归并排序由 John von Neumann 于 1945 年首先提出。其基本思想是将含有 m 个元素的未排序序列 x 分为两个子序列，其中每个序列长度都大约是原来的一半。然后对每个子序列重复类似的动作，直到每个子序列的长度均为 1。再从长度为 1 的子序列开始，将两个子序列“归并”为一个排序的序列。

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
        return result
```

下面的代码实现归并步骤：

```
Merge(left,right)
    var list result
    while length(left) >0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) >0
```

```
    append rest(left) to result
if length(right) >0
    append rest(right) to result
return result
```

6.5.1 [10] <6.2> 假设 MergeSort 运行在具有 Y 个核的多核处理器上，且 Y 远远小于 m（长度）。请问对于 Y 和 m 的不同取值，预期的加速比是多少？请画图表示。

6.5.2 [10] <6.2> 接下来，假设 Y 与 m 相等，这会对你前面的结论有何影响？如果要求获得尽可能高的加速比（例如，强比例缩放），请问该如何修改代码？

6.6 矩阵乘在大量应用中都扮演着重要角色。两个矩阵可以相乘的条件是第一个矩阵的列数和第二个矩阵的行数相同。

假设我们有一个 $m \times n$ 的矩阵 A，还有一个 $n \times p$ 的矩阵 B 与之相乘。乘法结果为一个 $m \times p$ 的矩阵 AB（或 $A \cdot B$）。如果令 $C=AB$，$C_{i,j}$ 代表在矩阵 (i, j) 位置处的值，则 $1 \leqslant i \leqslant m$ 且 $1 \leqslant j \leqslant p$，$C_{i,j}=\sum_{k=1}^{n} a_{i,k} \times b_{k,j}$。现在我们考虑是否可以将 C 的计算并行化。假设矩阵在存储中的存放顺序为 $a_{1,1}$，$a_{2,1}$，$a_{3,1}$，$a_{4,1}$，…。

6.6.1 [10] <6.5> 假设我们分别在单核 / 四核共享存储的系统计算 C，忽略关于存储的问题，请问四核相对于单核的预期加速比是多少？

6.6.2 [10] <6.5> 如果对 C 的多次更新发生在一行中连续的元素时，会引发伪共享带来的 cache 失效，重新计算 6.6.1 中的问题。

6.6.3 [10] <6.5> 怎样消除可能出现的伪共享问题？

6.7 下面的代码来自两个不同的程序，它们同时运行在一个包含 4 个处理器的 SMP（对称多核处理器）中。假设在开始运行之前，x 和 y 的初值均为 0。

- 核 1：x=2
- 核 2：y=2
- 核 3：w=x+y+1
- 核 4：z=x+y

6.7.1 [10] <6.5> w、x、y、z 所有可能的结果分别是什么？对每种可能的情况，通过分析指令的交错情况，解释其产生的原因。

6.7.2 [5] <6.5> 怎样能让执行变得更有确定性，以便只产生一种结果。

6.8 哲学家就餐问题是一个经典的同步和并发问题。该问题假设就座于一个圆桌周围的哲学家可以做两件事之一：吃饭或思考。吃饭时不能思考，反之亦然。在圆桌中心有一碗通心粉。每两个哲学家之间有一把叉子，这样每个哲学家左边有一把叉子，右边也有一把叉子。按照吃通心粉的方式，哲学家需要两把叉子才能吃到，而且只能使用紧挨着他左右的两把叉子。哲学家不能和其他人说话。

6.8.1 [10] <6.7> 请描述没有任一哲学家能吃到通心粉的情景（例如，饿死）。什么样的事件序列会导致该问题发生？

6.8.2 [10] <6.7> 如何通过引入优先级的概念来解决这一问题？可以平等地对待所有哲学家吗？请解释原因。

现在假定我们增加一个服务员负责为哲学家们分配叉子。只有在服务员允许之下他们才可以拿起叉子。服务员也知道所有叉子的状态。并且我们要求所有哲学家总是先请求拿起左边的叉子，再请求拿起右边的叉子，这样可以避免死锁。

6.8.3 [10] <6.7> 我们可以将哲学家向服务员发出的请求放入一个队列，也可以让请求周期性地重