产生影响。请你计算负载最重的处理器完成两倍负载（5%）和五倍负载（12.5%）时的加速比。此时其他处理器的利用率如何？

答案 | 如果一个处理器需要完成 5% 的并行负载，那么它必须执行 5% × 400 也就是 20 次加法，而另外 39 个处理器将平分剩余的 380 次加法。由于这些操作是同时进行的，我们可以将执行时间计算为两者的最大值：

$$\text{优化后的执行时间} = \max\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

加速比从 20.5 下降至 410*t*/30*t*=14。其余 39 个处理器的使用时间不到负载最重的处理器的一半：在等待负载最重的处理器完成时常为 20*t* 的任务时，其余处理器的计算时间仅为 380*t*/39=9.7*t*。

如果一个处理器需要完成 12.5% 的负载，那么它必须执行 50 次加法。上述公式变为：

$$\text{优化后的执行时间} = \max\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

加速比进一步降低为 410*t*/60*t*=7。其余处理器的使用时间不到 20%（9*t*/50*t*）。这个示例证明了负载均衡的重要性：只有一个处理器的负载是其他处理器的两倍时，加速比会降低三分之一，而当一个处理器的负载是其他处理器的五倍时，加速比几乎降低为原来的三分之一。—■

既然我们已经更好地理解了并行处理的目标和挑战，接下来就可以对本章的剩余部分进行概述了。6.3 节中介绍了一个比图 6-1 中更古老的分类方案。此外，该节还给出了两种支持在并行硬件上运行顺序应用程序的指令系统，即 SIMD 和向量（vector）。6.4 节介绍了多线程，这个术语经常与多处理器相混淆，部分原因在于它们都依赖于程序中相似的并发。6.5 节介绍了基本并行硬件的两种类型特征，它们的区别在于系统中所有的处理器是否依赖于单个的物理地址空间。如上文所述，这两种类型的常见形式分别为*共享内存多处理器*（SMP）和*集群*（cluster），6.5 节介绍前者。6.6 节描述来自图形硬件领域的相对较新的计算机类型，称为*图形处理单元*（GPU），GPU 也共享单个物理地址。（附录 B 更详细地描述了 GPU。）6.7 节描述了集群，这是具有多个物理地址空间的计算机的一个常见示例。6.8 节展示了用于将多个处理器（可以是集群中的多个服务器节点，也可以是微处理器中的多个核心）连接在一起的典型拓扑结构。6.9 节介绍了通过以太网在集群节点间进行通信的硬件和软件，展示了如何使用用户软件和硬件优化其性能。在 6.10 节中，我们将讨论找到并行基准测试的难度，其中还包括一个简单但很有启发意义的性能模型，该模型有助于设计应用程序和体系结构。我们将在 6.11 节中同时使用该模型和并行基准测试程序来比较多核计算机与 GPU。6.12 节揭示了加速矩阵乘法的最后也是最大的一个步骤。对于无法在 cache 中放下的矩阵，并行处理使用 16 个内核将性能提高了 14 倍。本章最后解析了一些常见的谬误和陷阱，并进行了总结。

在下一节中，我们会介绍一些你可能已经见过的缩略词，它们用于识别不同类型的并行计算机。

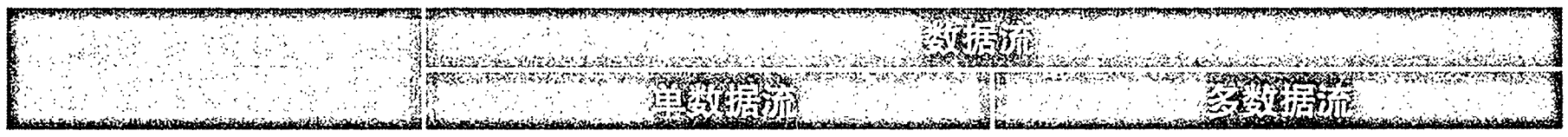
自我检测　判断题：强比例缩放不受 Amdahl 定律的约束。

6.3 SISD、MIMD、SIMD、SPMD 和向量机

本节介绍一种于 20 世纪 60 年代提出的并行硬件的分类方法，这种分类方法目前仍在使用。该分类基于指令流数量和数据流数量。图 6-2 展示了这种分类。传统的单处理器具有单个指令流和单个数据流，传统的多处理器具有多个指令流和多个数据流。这两个类别分别缩写为 SISD（Single Instruction stream，Single Data stream）和 MIMD（Multiple Instruction streams，Multiple Data streams）。

SISD：（单指令流单数据流的）单处理器。

MIMD：（多指令流多数据流的）多处理器。



		数据流	
		单数据流	多数据流
指令流	单指令流	SISD：Intel Pentium 4	SIMD：x86的SSE指令
	多指令流	MISD：目前为止还没有实例	MIMD：Intel Core i7

图 6-2　基于指令流数量和数据流数量的硬件分类实例：SISD、SIMD、MISD 和 MIMD

尽管可以在 MIMD 计算机上编写运行在不同处理器上的独立程序，但是，为了实现更宏大、更协调的目标，程序员通常会编写一个运行在 MIMD 计算机中所有处理器上的程序，不同的处理器通过条件语句执行不同的代码段。这种编程风格称为**单程序多数据流**（Single Program Multiple Data，SPMD），它是在 MIMD 计算机上编程的一种常用方法。

SPMD：单程序多数据流，是一种传统的 MIMD 编程模型，该模型中单个程序运行在所有处理器上。

最接近多指令流单数据流（MISD）的处理器应该是“流处理器”了，这种处理器在单数据流上以流水线方式执行一系列计算：解析来自网络的输入，分析数据，解压缩，查找匹配，等等。相比之下，与 MISD 相反的类型——SIMD 更受欢迎一些。SIMD（Single Instruction stream，Multiple Data streams）计算机对数据向量进行操作。例如，单个 SIMD 指令将 64 个数据流发送到 64 个 ALU 上，以在单个时钟周期内完成 64 次加法来将 64 个数字相加。我们在 3.6 节和 3.7 节中看到的子字并行指令是 SIMD 的另一个例子，事实上，SSE 这个缩写的中间字母 S 正是代表了 SIMD。

SIMD：单指令流多数据流。就像在向量处理器中那样，相同的指令应用于多个数据流上。

SIMD 的优点是所有的并行执行单元都是同步的，它们都响应自同一程序计数器（PC）中发出的同一指令。从程序员的角度看，这与他们已经很熟悉的 SISD 的概念非常接近。尽管每个单元将执行相同的指令，但是每个执行单元都有自己的地址寄存器，因此每个单元可以有不同的数据地址。根据图 6-1，一个顺序应用程序编译后，既可能在组织为 SISD 的串行硬件上运行，也可能在组织为 SIMD 的并行硬件上运行。

SIMD 的初衷是在数个执行单元上分摊控制单元的成本。因此，SIMD 的另一个优点是减少了指令带宽和空间——SIMD 只需要同时执行代码的一个副本，而消息传递的 MIMD 可能需要在每个处理器中都存有一个副本，而共享存储器的 MIMD 可能需要多个指令缓存。

SIMD 在处理 `for` 循环中的数组时效果最好。因此，为了在 SIMD 上并行运行，程序中必须存在大量相同结构的数据，这称为**数据级并行**（data-level parallelism）。SIMD 在处理 `case` 和 `switch` 语句时效果最差，在这些语句中，每个执行单元必须根据单元内存放的不同数据对这些数据执行不同的操作。存放有错误数据的执行单元必须被禁止执行，

数据级并行：通过对独立数据执行相同的操作来实现并行性。

以便存放有正确数据的执行单元可以继续工作。在有 n 个 case 语句的情况下，SIMD 处理器基本上只能以峰值性能的 $1/n$ 工作。

虽然促使 SIMD 类型产生的阵列处理器已经逐渐淡出历史（见 6.15 节），但是目前对 SIMD 的两种解释依然活跃。

6.3.1　x86 中的 SIMD：多媒体扩展

正如第 3 章所述，1996 年 x86 多媒体扩展（MMX）指令的灵感来源是窄整数数据的子字并行。随着摩尔定律的持续，更多的指令被添加进来，首先引入的是流式 SIMD 扩展（SSE）和高级向量扩展（AVX）。AVX 支持同时执行四个 64 位浮点数。操作和寄存器的宽度被编码在这些多媒体指令的操作码中。随着操作和寄存器的数据宽度的增加，多媒体指令的操作码数量也在增加，现在已经有数百条 SSE 和 AVX 指令（见第 3 章）。

6.3.2　向量机

正如我们将要看到的那样，对 SIMD 更加古老、更加优雅的解释被称为向量体系结构，这与 Seymour Cray 在 20 世纪 70 年代开始设计的计算机密切相关。这种体系结构与具有大量数据级并行的问题非常匹配。与早期的阵列处理器一样，64 个 ALU 并不是同时执行 64 次加法，而是采用向量体系结构流水化 ALU，从而以更低的成本获得良好的性能。向量体系结构的基本原理是从内存中收集数据元，将它们按顺序放入一大组寄存器中，使用流水化的执行单元在寄存器中依次对它们进行操作，然后将结果写回内存。向量体系结构的一个关键特性是拥有一组向量寄存器。这样，一个向量体系结构中可能具有 32 个向量寄存器，每个寄存器包含 64 个 64 位宽的数据元。

例题　将向量与常规代码进行比较

现在假设我们使用向量指令和向量寄存器来扩展 RISC-V 指令系统体系结构。向量操作使用与 RISC-V 操作相同的名称，但在后缀上添加了“V”。例如，`fadd.d.v` 这条指令将两个双精度向量相加。接下来添加 32 个向量寄存器 v0 ～ v31，每个寄存器包含 64 个 64 位数据元。向量指令既可以将两个向量（V）寄存器作为输入（使用指令 `fadd.d.v`），也可以将一个向量寄存器和一个标量寄存器作为输入（使用指令 `fadd.d.vs`）。在后一种情况下，标量寄存器中的值用作所有操作的输入——`fadd.d.vs` 这个操作将标量寄存器中的内容加到向量寄存器中的每个数据元上。操作 `fld.v` 和 `fsd.v` 表示向量加载与向量存储，这两条指令分别加载或存储整个双精度数据向量。其中一个操作数是要加载或存储的向量寄存器，另一个操作数是 RISC-V 的通用寄存器，用于给出存储器中向量的起始地址。

在简要描述后，答案中给出了下述语句的传统 RISC-V 代码和向量 RISC-V 代码：

$$Y = a \times X + Y$$

其中，X 和 Y 是 64 个双精度浮点数的向量，最初驻留在内存中，a 是一个标量双精度变量。（这个示例就是所谓的 DAXPY 循环，该循环组成 Linpack 基准测试程序的内部循环；DAXPY 正是 double precision a × X plus Y 的缩写。）假设 X 和 Y 的起始地址分别存放在 x19 和 x20 中。

答案　DAXPY 的传统 RISC-V 代码是：

```
        fld     f0, a(x3)       // load scalar a
```

```
        addi    x5, x19, 512   // end of array X
loop:   fld     f1, 0(x19)     // load x[i]
        fmul.d  f1, f1, f0     // a * x[i]
        fld     f2, 0(x20)     // load y[i]
        fadd.d  f2, f2, f1     // a * x[i] + y[i]
        fsd     f2, 0(x20)     // store y[i]
        addi    x19, x19, 8    // increment index to x
        addi    x20, x20, 8    // increment index to y
        bltu    x19, x5, loop  // repeat if not done
```

DAXPY 的假想向量 RISC-V 代码是：

```
        fld         f0, a(x3)       // load scalar a
        fld.v       v0, 0(x19)      // load vector x
        fmul.d.vs   v0, v0, f0      // vector-scalar multiply
        fld.v       v1, 0(x20)      // load vector y
        fadd.d.v    v1, v1, v0      // vector-vector add
        fsd.v       v1, 0(x20)      // store vector y
```

在上面的例题中，两个代码段之间存在一些有趣的对比。最引人注目的是，向量处理器大大降低了动态指令带宽。传统的 RISC-V 指令体系结构需要执行 500 条指令，相比之下，向量 RISC-V 只需要执行 6 条指令就完成了工作。这种减少是因为一次向量操作可以处理 64 个数据元，并且在 RISC-V 指令体系结构中近一半的循环开销指令在向量代码中无须存在。如你所料，指令取指和执行次数的减少的确可以节省能耗。

另一个重要的区别是流水线冒险（见第 4 章）的频率不同。在传统的 RISC-V 代码中，每个 `fadd.d` 指令必须等待 `fmul.d` 指令完成，每个 `fsd` 指令必须等待 `fadd.d` 指令完成，而且每个 `fadd.d` 指令和 `fmul.d` 指令还必须等待 `fld` 指令完成。在向量处理器中，每个向量指令只会停顿每个向量中的第一个数据元，然后后续数据元将顺利地沿着流水线流动。因此，每个向量操作仅需要一次流水线停顿，而不是每个数据元都会停顿一次。在本例题中，传统 RISC-V 的流水线停顿频率将是向量 RISC-V 版本的约 64 倍。通过循环展开可以消除传统 RISC-V 上的流水线停顿（见第 4 章）。但是，指令带宽的巨大差异依然不能减少。

因为向量中的各数据元是相互独立的，因此它们可以并行操作，这非常类似于 Intel x86 AVX 指令中的子字并行。所有现代向量计算机都具有向量功能单元，该单元具有多个并行流水线（我们称为向量通道，见图 6-2 和图 6-3），每个时钟周期可计算出两个或更多的结果。

详细阐述 在上面的例题中，循环次数与向量的长度恰好相等。当循环次数更小时，向量体系结构可以使用寄存器来减少向量操作的长度。当循环次数更大时，我们可以添加循环标记代码来进行全长度的向量循环操作，最后处理剩余部分。后一处理过程称为循环切分（strip mining/loop sectioning）。

6.3.3　向量与标量

与传统指令系统体系结构（在本节中被称为标量体系结构）相比，向量指令具有几个重要的属性：

- 单个向量指令指定了大量工作——相当于执行了完整的循环。正因为这样，指令取指和译码带宽大大减少。
- 通过使用向量指令，编译器或程序员确认了向量中的每个结果都是独立的，因此硬件

无须再检查向量指令内的数据冒险。

- 当程序中存在数据级并行时，相比使用 MIMD 多处理器，使用向量体系结构和编译器的组合更容易写出高效的应用程序。
- 硬件只需要在两条向量指令之间检查向量操作数之间的数据冒险，而无须检查向量中的每个数据元。减少检查的次数可以节省能耗和时间。
- 访问存储器的向量指令具有确定的访问模式。如果向量中的数据元位置都是连续的，则可以从一组存储器中交叉访问数据块，从而快速获取向量。因此，对整个向量而言，主存储器的延迟开销看上去只有一次，而不是对向量中的每个字都产生一次。
- 因为整个循环被具有已知行为的向量指令所取代，所以通常由循环引发的控制冒险不再存在。
- 与标量体系结构相比，节省的指令带宽和冒险检查以及对存储器带宽的有效利用，使得向量体系结构在功耗和能耗方面更具有优势。

综上所述，在同等数据量的前提下，向量操作可以比一组标量操作序列更快地完成。如果应用程序当中经常使用这些向量操作，设计者将更有动力在设计中加入向量单元。

6.3.4　向量与多媒体扩展

与 x86 AVX 指令中的多媒体扩展类似，向量指令也指定了多种操作。但是多媒体扩展通常只能表示几种操作，而向量指令可以指定几十种操作。与多媒体扩展不同，向量操作中数据元的数量不存放在操作码中，而是存放一个在单独的寄存器中。这种区别意味着仅通过改变该寄存器的内容就可以用不同数量的数据元实现不同版本的向量体系结构，从而保持二进制代码的兼容性。与之相反，在 x86 的 MMX、SSE、SSE2、AVX、AVX2 等多媒体扩展中，每当向量长度变化时，就需要添加一组新的指令操作码。

还有一点，与多媒体扩展不同，向量的数据传输不需要是连续的。向量支持步长访问（strided access）和索引访问（indexed access），前者是在存储器中每隔 n 个数据元加载一次数据，后者是按照数据项的地址将数据加载到向量寄存器中。索引访问也称为聚集 – 分散（gather-scatter），因为索引加载从主存将数据元收集为连续的向量元素，而索引存储将向量元素分散至内存中。

与多媒体扩展类似，向量体系结构可以灵活支持不同的数据宽度，因此可以使向量操作工作在 32 个 64 位数据元、64 个 32 位数据元、128 个 16 位数据元或者 256 个 8 位数据元上。向量指令的并行特性可以使其采用深度流水的功能单元、并行功能单元阵列或者并行功能单元与流水功能单元的组合来执行这些操作。图 6-3 说明了如何通过使用并行流水线执行向量加法指令来提高向量的性能。

向量算术指令通常仅允许一个向量寄存器的元素 N 与其他向量寄存器的元素 N 进行交互。这极大地简化了高度平行的向量单元的构造——可构造为多个平行的**向量通道**（vector lane）。与高速公路一样，我们可以通过添加更多车道来增加向量单元的峰值吞吐量。图 6-4 是一个四通道向量单元的结构图。将一通道增加为四通道可以使得每个向量指令的时钟周期数大约减少为原来的 1/4。为了能够利用多通道，应用程序和体系结构都必须支持长向量。否则，指令将很快执行完毕以至于没有足够的新指令可以被执行，第 4 章中的指令级并行技术可以提供足够的向量指令。

向量通道：一个或多个向量功能单元和一部分向量寄存器堆。受高速公路上的多车道提高交通速度的启发，利用多个通道同时执行向量操作。

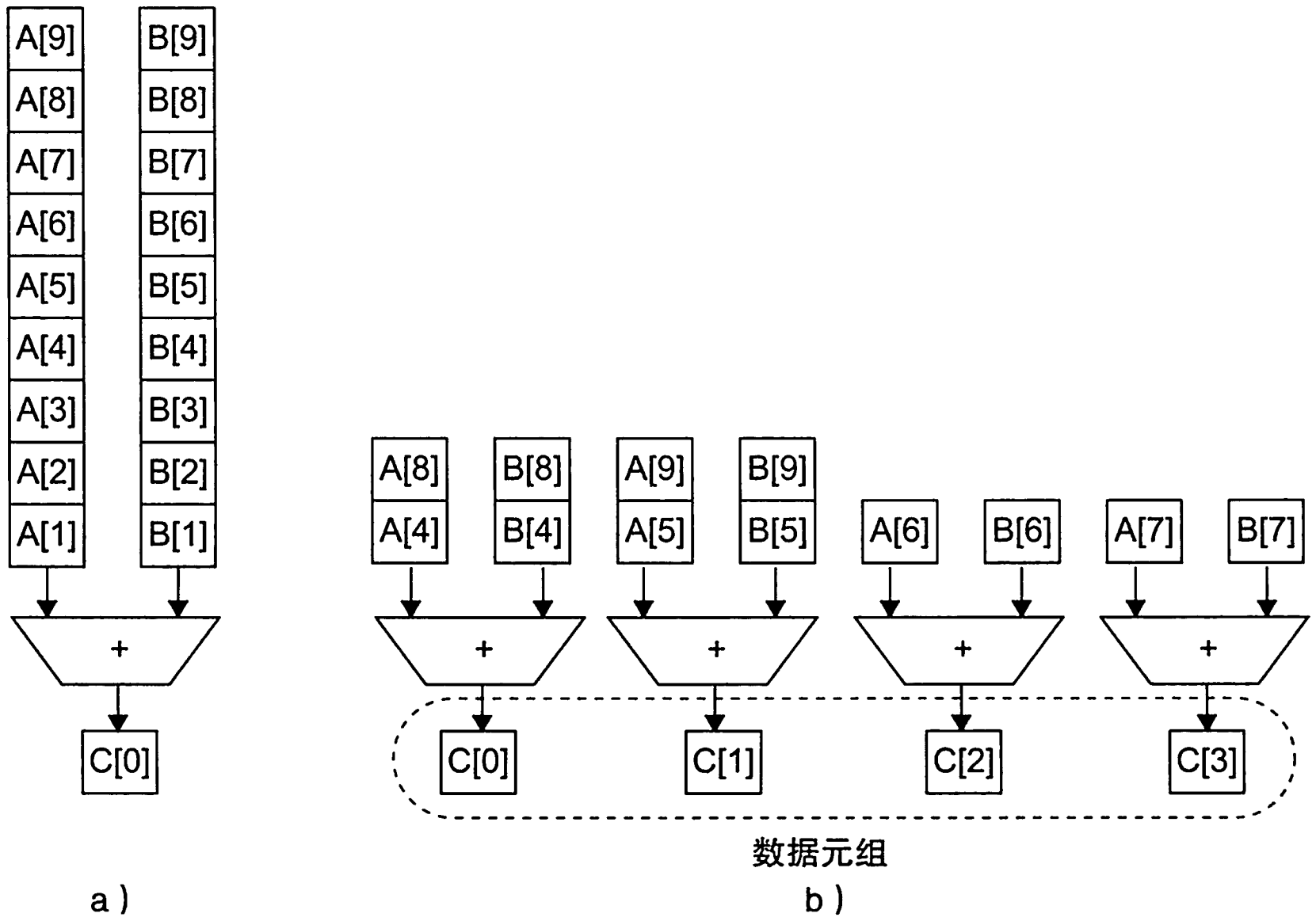


图 6-3 使用多个功能单元来提高单个向量加法指令 C=A+B 的性能。左侧的向量处理器（a）只有单个加法流水线，每个周期可以完成一次加法。右侧的向量处理器（b）有四个加法流水线，每个周期可以完成四次加法。单个向量加法指令中的数据元被分散在四个通道中

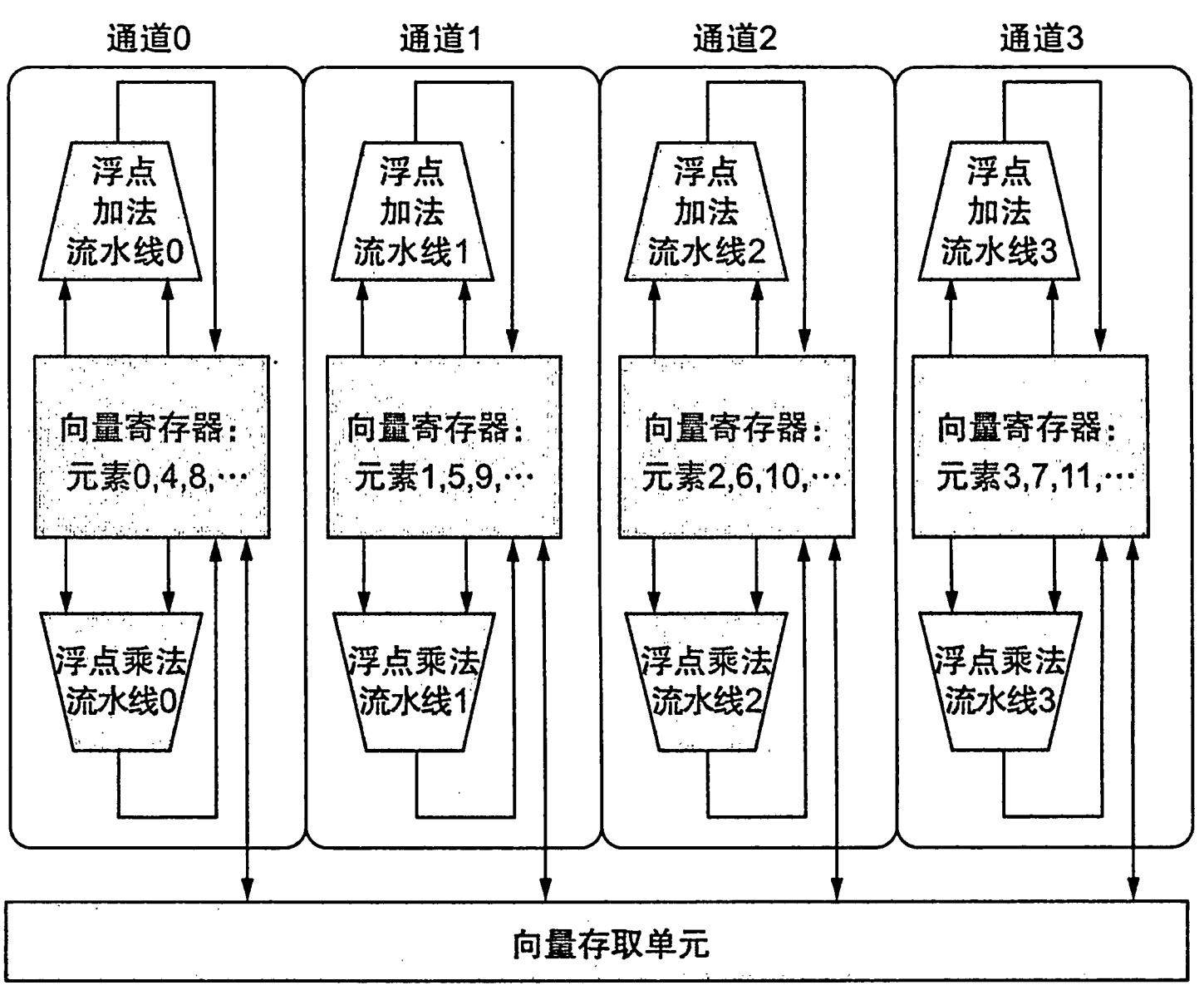


图 6-4 包含四个通道的向量单元的结构图。向量寄存器内存等量地分配给每个通道，每个向量寄存器中的数据元依次分配给每个通道。图中画出了三个向量功能单元：浮点加法单元、浮点乘法单元和存取单元。每个向量运算单元包含四个执行流水线，每个通道一个，它们协同工作以完成单个向量指令。请注意向量寄存器堆的每个部分是如何只为其对应通道的功能单元提供足够的读写端口的（见第 4 章）

总的来说，向量体系结构是执行数据并行处理程序的一种非常有效的方法。与多媒体扩展相比，向量体系结构更适合编译器技术，而且随着时间的推移，向量体系结构比 x86 体系

结构的多媒体扩展更容易进行改进。

给出这些经典分类后，接下来我们将了解如何利用指令的并行流来提高单个处理器的性能，我们还会将该方法应用到多处理器中。

详细阐述 既然向量体系结构有这么多优点，为什么还没有在除高性能计算之外的领域流行呢？主要原因是担心向量寄存器的巨大状态会增加上下文切换时间，以及加大处理向量加载和存储中的页错误的难度，并且 SIMD 指令已经实现了向量指令的部分优点。此外，只要指令级并行的进步可以提供摩尔定律所需的性能提升，就没有理由去改变体系结构的类型。

详细阐述 向量和多媒体扩展的另一个优点是，向量指令系统体系结构的扩展相对容易，可以提高数据并行操作的性能。

详细阐述 Intel 的 Haswell x86 处理器支持 AVX2 指令系统，该指令系统只有聚集操作而没有分散操作。

自我检测 判断题：以 x86 为例，多媒体扩展可以被认为是一种处理短向量的向量体系结构，这种体系结构仅支持连续的向量数据传输。

6.4 硬件多线程

从程序员的角度来看，**硬件多线程**（hardware multithreading）是与 MIMD 相关的一个概念。MIMD 依赖于多个**进程**（process）或**线程**（thread）使得多个处理器保持忙碌状态，而硬件多线程允许多个线程以重叠的方式共享单个处理器的功能单元，以有效地利用硬件资源。为了支持这种共享，处理器必须复制每个线程的独立状态。例如，每个线程都有一个寄存器堆和程序计数器的独立副本。内存本身可以通过虚拟内存机制实现共享，在多道程序编程中已经支持了这种方法。此外，硬件必须具有在线程之间快速切换的能力。特别是，线程切换应该比进程切换更加有效，线程切换可以是瞬时切换，而进程切换通常需要数百到数千个处理器周期。

硬件多线程：通过在一个线程停顿时切换到另一个线程来提高处理器的利用率。

线程：包括程序计数器、寄存器状态和栈。线程是一个轻量级的进程，线程通常共享一个地址空间，而进程则不共享。

进程：包括一个或多个线程、完整的地址空间和操作系统状态。因此，进程的切换通常需要调用操作系统，而线程切换则不用。

硬件多线程主要有两种实现方法。**细粒度多线程**（fine-grained multithreading）在每条指令执行后进行线程切换，导致了多线程的交叉执行。这种交叉执行通常以一种轮转方式完成，并跳过在该时钟周期停顿的任何线程。为了实现细粒度多线程，处理器必须能够在每个时钟周期切换线程。细粒度多线程的一个优点是可以隐藏由短期和长期停顿引起的吞吐量损失，因为当一个线程停顿时可以执行来自其他线程的指令。细粒度多线程的主要缺点是会减慢单个线程的执行速度，因为已经就绪的线程会因为执行其他线程的指令而延迟。

细粒度多线程：硬件多线程的一种版本，在每条指令之后切换线程。

粗粒度多线程：硬件多线程的另一种版本，仅在重大事件（例如末级 cache 失效）之后才切换线程。

粗粒度多线程（coarse-grained multithreading）是作为细粒度多线程的另一种可选项被发明的。粗粒度多线程仅在高开销的停顿上切换线程，例如末级 cache 失效时。这种改变降低了高速切换线程的要求，并且几乎不会减

慢单个线程的执行速度，因为只有在线程遇到高开销的停顿时才会发射来自其他线程的指令。然而，粗粒度多线程有一个严重缺点：降低吞吐量损失的能力有限，尤其是对于短停顿。这种限制源于粗粒度多线程的流水线启动开销。因为粗粒度多线程处理器从单个线程发出指令，所以当发生停顿时，必须清空或冻结流水线。在停顿之后开始执行的新线程必须在导致停顿的指令能够完成之前填充流水线。由于这种启动开销，粗粒度多线程对于降低高成本停顿的损失更为有用，因为在这种情况下，流水线重新填充的时间与停顿时间相比可以忽略不计。

同时多线程（Simultaneous Multithreading，SMT）是硬件多线程的一种变体，它使用多发射、动态调度流水线的处理器资源来挖掘线程级并行和指令级并行（见第4章）。提出SMT的主要原因是，多发射处理器中通常具有大多数单线程难以充分利用的并行功能单元。此外，通过寄存器重命名和动态调度（见第4章），可以发出来自相互独立的多线程的多条指令，而不需要考虑它们之间的依赖关系；可以通过动态调度能力来解决相关性的问题。

同时多线程：多线程的一个版本，通过利用多发射、动态调度的微体系结构的资源来降低多线程的成本。

因为SMT依赖于现有的动态机制，因此它不会在每个时钟周期切换资源。相反，SMT始终执行来自多个线程的指令，将资源分配交给硬件完成，这些资源是指令槽和重命名寄存器。

图6-5概念性地说明了在不同的处理器配置下对超标量资源利用能力的差别。上半部分展示了四个线程如何在不支持多线程的超标量处理器上独立执行。下半部分展示了三个不同的多线程选项下，四个线程如何组合以在单个处理器上更高效地执行。这三个选项是：

- 支持粗粒度多线程的超标量
- 支持细粒度多线程的超标量
- 支持同时多线程的超标量

在不支持硬件多线程的超标量处理器中，发射槽的使用受到指令级并行的限制。此外，诸如指令cache失效之类的绝大多数停顿，都可能使整个处理器处于空闲状态。

在粗粒度多线程超标量处理器中，通过切换到使用该处理器资源的另一个线程，可以部分隐藏长停顿。尽管这样做可以减少完全空闲的时钟周期的数量，但是流水线的启动开销依然会带来空闲的时钟周期，并且ILP的限制意味着并非所有的发射槽都能得到充分利用。在细粒度多线程的情况下，线程的交叉执行可以基本消除空闲的时钟周期。但是，由于在给定的时钟周期内只有一个线程发出指令，因此指令级并行的限制仍会导致某些时钟周期内出现空闲的发射槽。

在SMT中，线程级并行和指令级并行都得到了充分利用，多个线程在单个时钟周期中使用发射槽。理想情况下，发射槽的使用仅受到多个线程之间资源需求不平衡和资源可用性的限制。实际上，还有其他因素可能会限制使用的发射槽的数量。虽然图6-5大大简化了这些处理器的实际操作，但是它确实说明了多线程在一般情况下的潜在性能优势，特别是SMT。

图6-6给出了Intel Core i7 960单处理器上多线程的性能和能耗优势，该处理器的硬件支持两个线程。平均加速比为1.31，这对于有少量额外资源执行硬件多线程的情况来说不算坏。平均能耗提高1.07，这非常好。总之，在能耗基本不变的前提下获得性能提升是使人感到高兴的。

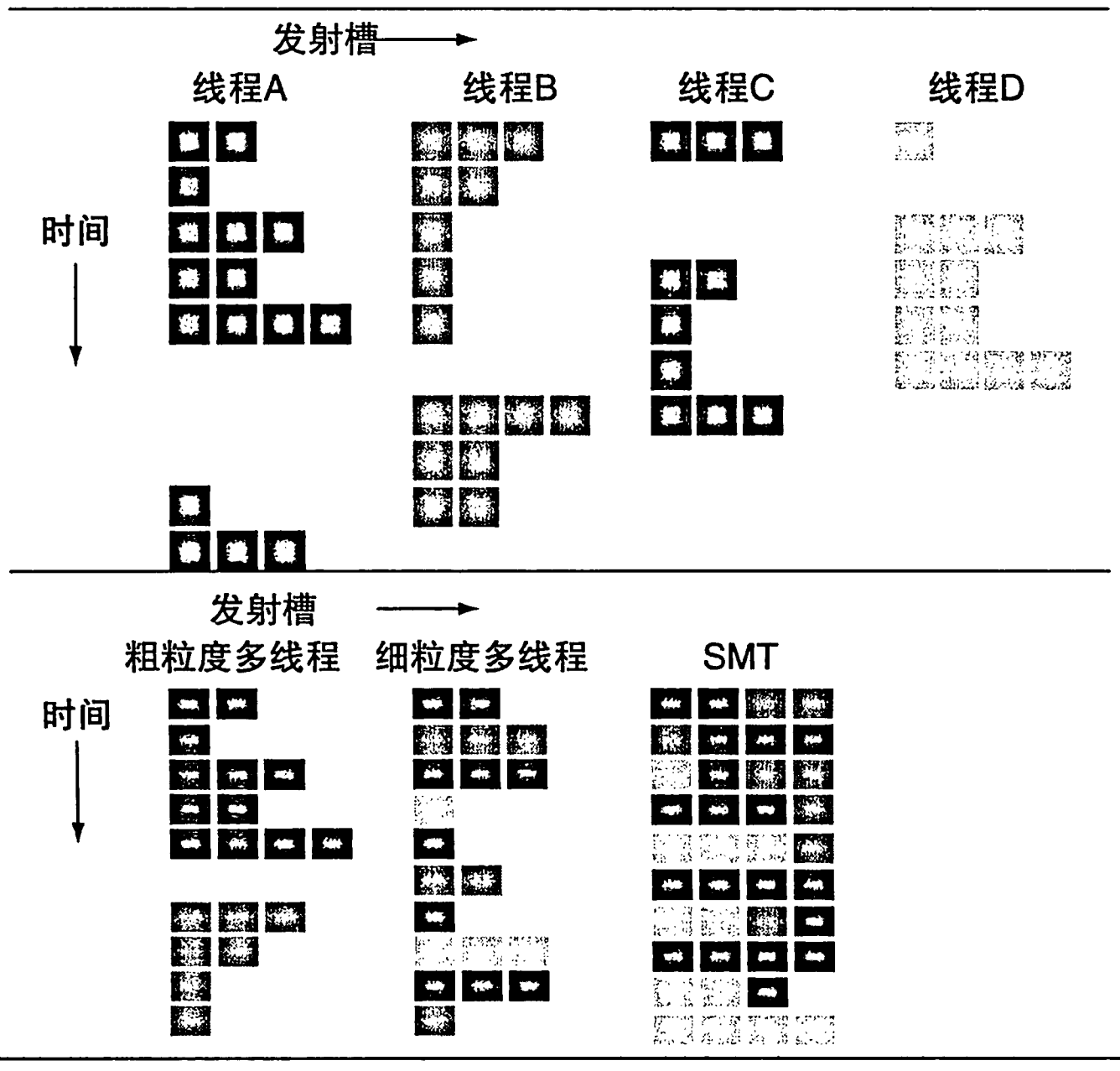


图 6-5 在不同方法中，四个线程如何使用超标量处理器的发射槽。上半部分的四个线程显示了如何在不支持多线程的标准超标量处理器上单独执行每个线程。下半部分的三个示例分别显示了在三个多线程选项中它们是如何一起执行的。水平维度表示每个时钟周期中的指令发射能力。垂直维度表示一系列时钟周期。空白的位置表示在该时钟周期中未使用的相应发射槽。灰色阴影对应于多线程处理器中的四个不同线程。粗粒度多线程的额外流水线启动损失（在图中未画出）将导致粗粒度多线程的吞吐量产生进一步损失

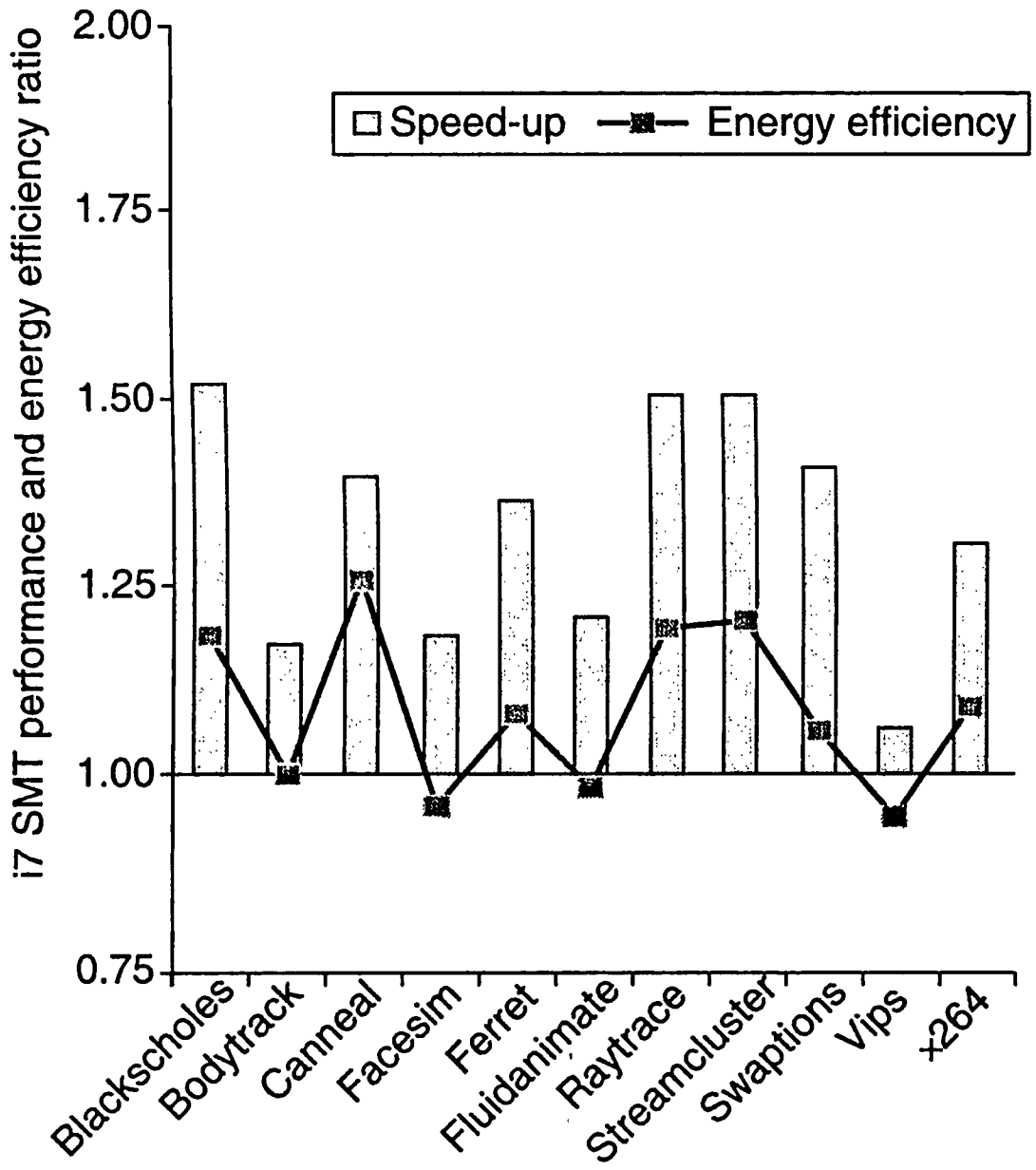


图 6-6 在 i7 处理器的一个核心上对 PARSEC 基准测试程序（见 6.9 节）使用多线程的平均加速比为 1.31，能效改善比例为 1.07。Esmaeilzadeh 等人收集并分析了这些数据 [2011]

现在我们已经看到如何通过多个线程更有效地利用单个处理器的资源，接下来我们将看到如何通过多线程来利用多处理器的资源。

自我检测

1. 判断题：多线程和多核都依赖并行以从芯片中获得更高的效率。
2. 判断题：同时多线程（SMT）使用多个线程来提高动态调度乱序处理器的资源利用率。

6.5　多核及其他共享内存多处理器

尽管硬件多线程已经用很小的代价提升了处理器效率，但是在过去的十几年中，一个巨大的挑战是：如何通过有效地编程来利用单个芯片上数量不断增长的处理器，以发挥出摩尔定律呈现出的性能潜力。

考虑到重写旧程序使其能在并行硬件上良好运行是困难的，一个自然的问题是：计算机设计者该如何简化该任务？一种方法是为所有处理器提供一个共享的统一物理地址空间，使得程序无须考虑数据的存放位置，只需考虑如何并行执行。在这种方法中，程序的所有变量对其他任何处理器都是随时可见的。另一种方法是每个处理器采用独立的地址空间，这就必须进行显式共享，我们将在 6.7 节描述这种情况。当共享物理地址空间时，硬件通常提供 cache 一致性，以保证共享内存的一致性（见 5.8 节）。

如上所述，*共享内存多处理器*（SMP）为所有处理器提供统一物理地址空间——对多核芯片几乎总是如此——尽管更准确的术语是共享地址多处理器。处理器通过存储器中的共享变量进行通信，所有处理器都能够通过加载和存储指令访问任意存储器位置。图 6-7 是 SMP 的典型结构。请注意，即使这些系统共享物理地址空间，它们仍可在自己的虚拟地址空间中运行独立程序。

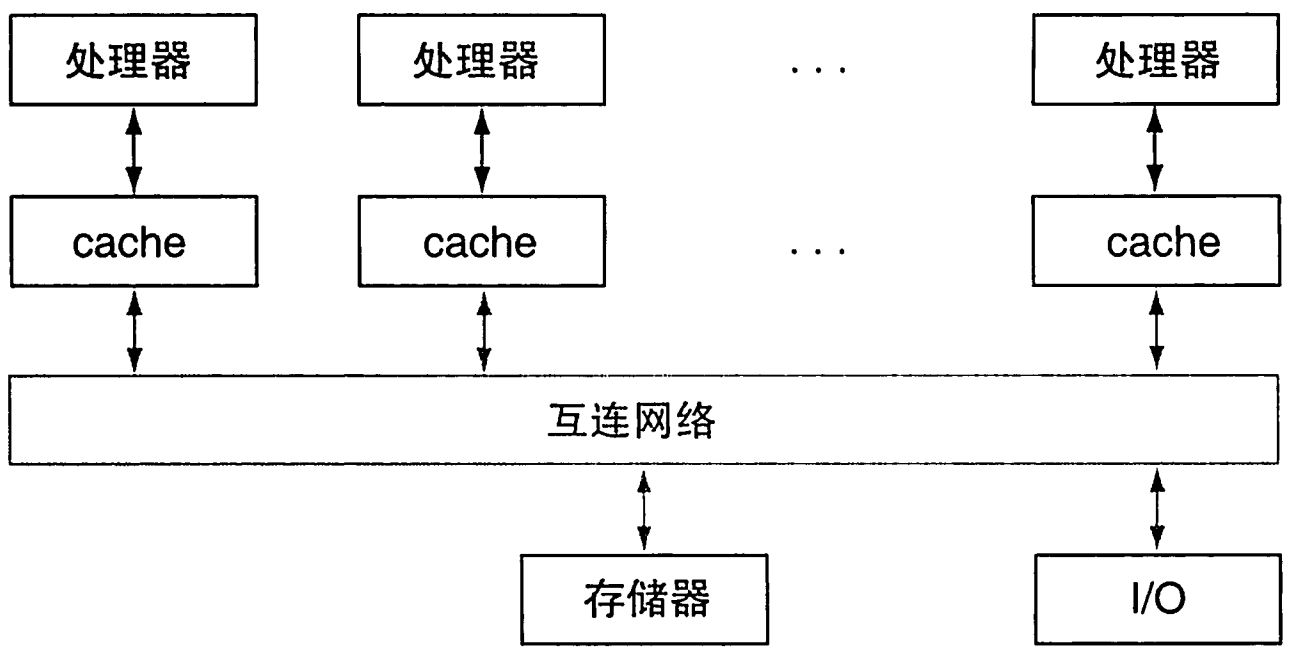


图 6-7　一个共享内存多处理器的典型结构

单地址空间多处理器有两种类型。在第一种类型中，访存延迟不依赖于是哪个处理器提出的请求。这种机器称为**统一内存访问**（UMA）多处理器。在第二种类型中，一些存储器的访问会比其他存储器快很多，这取决于是哪个处理器访问哪个存储。这通常是因为主存储器被划分，并分配给不同的微处理器或同一芯片上的不同内存控制器。这种机器称为**非统一内存访问**（NUMA）多处理器。如你所料，NUMA 多处理器的编程难度高于 UMA 多处理器，但 NUMA 机器可以扩展到更大规模，并且 NUMA 在访问附近的内存

统一内存访问：一种多处理器，无论哪个处理器访问存储器，存储器的访问延迟都大致相同。

非统一内存访问：一种单地址空间多处理器，存储器的访问延迟各不相同，具体取决于哪个处理器访问哪个存储。

时具有较低的延迟。

处理器并行执行时通常需要共享数据，因此在操作共享数据时需要进行协调；否则，一个处理器可能会在另一个处理器还没有对共享数据完成操作之前就开始处理该数据。这种协调被称为**同步**（synchronization），我们在第 2 章中讨论过。当统一地址空间支持共享时，必须提供一套独立的同步机制。一种方法是为共享变量提供**锁**（lock）。同一时刻只能有一个处理器可以获得锁，其他想要操作共享数据的处理器必须等待，直到该处理器解锁共享变量为止。2.11 节描述了 RISC-V 中关于锁操作的指令。

同步：协调两个或多个进程行为的过程，这些进程可能在不同的处理器上运行。

锁：一种同步机制，同一时刻仅允许一个处理器访问数据。

例题 | 一个共享地址空间的简单并行处理程序

现在假设我们要使用一台具有统一内存访问时间的共享内存多处理器计算机，对 64 000 个数字求和。假设该计算机有 64 个处理器。

答案 | 首先需要确保每个处理器负载均衡，因此我们先将这组数字划分为相同大小的子集。我们不能将这些子集分配至不同的内存空间，因为这台计算机只有一个统一内存空间；我们只为每个处理器提供不同的起始地址。Pn 为处理器的编号，介于 0 至 63 之间。所有处理器通过运行一个循环程序来对其数字子集求和：

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
  sum[Pn] += A[i]; /*sum the assigned areas*/
```

（注意，在 C 语言中，`i+=1` 是 `i=i+1` 的简写形式。）

下一步是将这 64 个子集的和继续求和。这一步称为**归约**（reduction），我们采用分治的方式求和。首先，1/2 的处理器将成对的子集和做求和，之后 1/4 的处理器继续将成对的新子集和做求和，以此类推，直到得到一个最终的总和。图 6-8 说明了本次归约的层次结构。

归约：一种处理数据结构并返回单个值的函数。

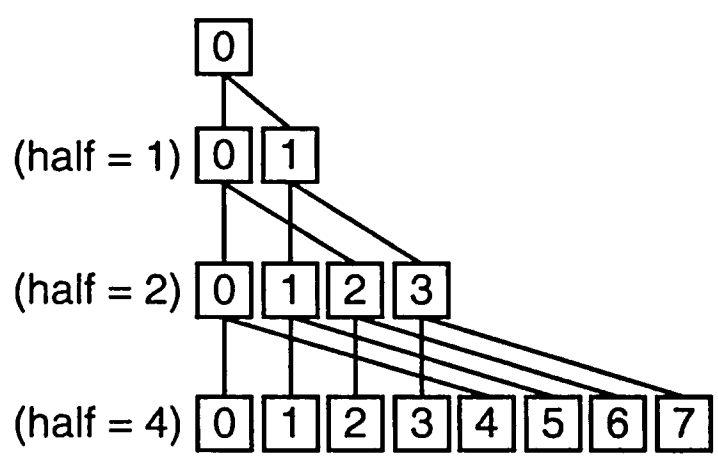


图 6-8　对每个处理器上的数据进行求和的归约过程的最后四级结构，归约过程自下向上进行。对于所有编号 i 小于 half 的处理器，将编号为（i+half）的处理器中产生的和加到该处理器上

在本例中，在“消费者”处理器试图从“生产者”处理器上读取结果之前必须进行同步。否则，“消费者”可能会读取到数据的旧值。我们希望每个处理器都拥有自己的循环变量 i，因此需要指出 i 是一个“私有”变量。以下是相应的代码（half 也是私有变量）：

```
half = 64; /*64 processors in multiprocessor*/
do
   synch(); /*wait for partial sum completion*/
```

```
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1];
        /*Conditional sum needed when half is
        odd; Processor0 gets missing element */
        half = half/2; /*dividing line on who sums */
        if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*exit with final sum in Sum[0] */
```

硬件/软件接口 由于人们长久以来对并行编程都有着浓厚的兴趣，迄今为止已经出现了数百次构建并行编程系统的尝试。一个有局限但常用的例子是OpenMP。它只是一个应用程序编程接口（Application Programmer Interface，API），带有一组可以对标准编程语言进行扩展的编译器制导、环境变量和运行时库。它为共享内存多处理器提供了可移植、可扩展且简单的编程模型。它最初的目标是循环并行化并进行归约。

OpenMP：一套支持共享内存多处理的API，它在UNIX和Microsoft平台上运行，支持C/C++/Fortran语言。它包括编译器制导、库和运行时指导。

大多数的C语言编译器已经支持OpenMP。在UNIX C编译器上使用OpenMP API的指令是：

```
cc -fopenmp foo.c
```

OpenMP使用编译制导对C语言进行扩展，编译制导就是C宏指令，如#define和#include。如上例所示，如果要将处理器数量设置为64，需要使用如下命令：

```
#define P 64 /* define a constant that we'll use a few times */
#pragma omp parallel num_threads(P)
```

这些指令表明，运行时库应该使用64个并行线程。

为了将顺序for循环转换为并行for循环（通过该循环在所有线程之间平均分配任务），只需将代码写为如下形式（假设sum初始化为0）：

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
    for (i = 0; 1000*Pn; i < 1000*(Pn+1); i += 1)
      sum[Pn] += A[i]; /*sum the assigned areas*/
```

要进行归约，可以使用另一个命令告诉OpenMP归约运算符是什么，以及需要使用哪个变量来放置归约的结果。

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
    FinalSum += sum[i]; /* Reduce to a single number */
```

注意，从现在开始需要由OpenMP库来查找使用64个处理器对64个数字进行求和的效率最高的代码。

尽管使用OpenMP可以使基本并行代码的编写变得简单，但它并不能对调试起到帮助，因此许多程序员在使用比OpenMP更加复杂的并行编程系统，正如今天许多程序员在使用比C语言更加高效的编程语言那样。

现在我们已经了解了经典的MIMD硬件和软件，接下来我们将探索更新奇的MIMD架构，它继承于另一体系结构，也因此在另一个方向上对并行编程提出了挑战。

详细阐述 一些作者将SMP作为对称多处理器（symmetric multiprocessor）的缩略词，

以说明对所有处理器而言，访问存储器的时延都大致相同。这种转变是为了与大规模 NUMA 多处理器做区分，因为这两者都使用统一地址空间。由于集群比大规模 NUMA 多处理器更为常见，因此在本书中我们将 SMP 恢复为其原始含义（即共享内存多处理器），并将其与使用多个地址空间的集群进行对比。

详细阐述 共享物理地址空间的一种替代方案是：使用独立的物理地址空间，但共享一个公共的虚拟地址空间，由操作系统负责处理通信。这种方法已经被尝试过，但是需要非常大的开销，以至于不能为注重性能的程序员提供一个实际可用的共享内存抽象。

自我检测 判断题：共享内存多处理器无法利用任务级并行。

6.6 GPU 简介

最初将 SIMD 指令添加到现有体系结构中的理由是：许多微处理器被用于 PC 和工作站中的图形显示器上，以至于越来越多的处理时间被用于图形上。随着处理器上的晶体管数量根据摩尔定律持续增长，改进图形处理效率逐渐成为一个值得研究的问题。

改进图形处理的主要驱动力来自计算机游戏行业，包括 PC 和索尼 PlayStation 等专用游戏主机。快速增长的游戏市场鼓励许多公司加大了在开发更快速的图形硬件上的投资，这种正反馈使得图形处理的改进速度比主流微处理器中的通用处理快得多。

由于图形和游戏社区与微处理器开发社区的目标不同，因此它发展出了自己独特的处理风格和术语。随着图形处理单元计算能力的增加，它们被命名为图形处理单元（Graphics Processing Unit）或 GPU，以区别于 CPU。

只需要几百美元，任何人都可以购买到带有数百个并行浮点单元的 GPU，这使得普通人进行高性能计算更容易了。当这种潜力与编程语言相结合时，GPU 变得更易于编程，而大众对 GPU 计算的兴趣也在逐渐增长。因此，今天的许多科学和多媒体应用的程序员开始犹豫究竟是使用 GPU 还是 CPU 编程。

（本节重点介绍如何使用 GPU 进行计算。想要了解 GPU 计算是如何与 GPU 的传统功能——图形加速相结合的，请参阅附录 B。）

下面是一些 GPU 区别于 CPU 的关键特性：

- GPU 是作为 CPU 补充的加速器，因此不需要能够执行 CPU 上的所有任务。这种角色定位同时也允许了 GPU 将所有的资源都用于图形。GPU 在执行一些任务时表现很差甚至完全不能执行，这也是被允许的，因为在一个同时拥有 CPU 和 GPU 的系统中，CPU 可以根据需要执行这些（GPU 不能执行的）任务。
- GPU 的问题规模通常为几百 MB 到 GB，而不是几百 GB 到 TB。

下面是一些导致（GPU 和 CPU）体系结构风格不同的差异：

- 也许最大的区别在于 GPU 不依赖于多级 cache 来消除内存的长延迟，但 CPU 依赖。与此相反，GPU 依靠硬件多线程（见 6.4 节）来隐藏内存延迟。也就是说，在存储器发出请求与数据到达之间的时间里，GPU 执行了数百或数千个独立于该请求的线程。
- 因此，GPU 的存储器面向带宽而不是延迟。甚至还有用于 GPU 的特殊图形 DRAM 芯片，这种芯片比用于 CPU 的 DRAM 芯片宽度更大，并且具有更高的带宽。此外，GPU 存储器通常比传统微处理器的主存更小。在 2013 年，GPU 存储器的大小一般为

4 ～ 6GiB 甚至更低，而 CPU 存储器的大小为 32 ～ 256GiB。最后，请记住，对于通用计算来说，需要将 CPU 内存和 GPU 内存之间传输数据的时间也计算在内，因为毕竟 GPU 是协处理器。

- 考虑到 GPU 需要通过多线程来获取良好的内存带宽，除多线程外，GPU 还可以容纳许多并行处理器（MIMD）。因此，每个 GPU 处理器都比传统 CPU 拥有更多的线程，并且它们拥有更多的处理器。

硬件 / 软件接口 虽然 GPU 是为小众的图形应用程序而设计的，但是一些程序员想要通过某种形式特化他们的应用程序，使其能够挖掘 GPU 潜在的高性能。在厌倦了尝试使用图形 API 和语言后，他们开发了类 C 语言的编程语言，这种语言支持直接为 GPU 编写程序。一个例子是 NVIDIA 的 CUDA（Compute Unified Device Architecture），它使程序员能够编写在 GPU 上运行的 C 程序，尽管会有一些限制。附录 B 中给出了 CUDA 的示例代码。（OpenCL 是由多个公司发起的可移植编程语言，可提供 CUDA 中的许多功能。）

NVIDIA 决定将所有形式的并行都定义为 CUDA 线程。使用这种最底层的并行作为编程原语，编译器和硬件可以将数千个 CUDA 线程组合在一起，以利用 GPU 内的各种形式的并行：多线程、MIMD、SIMD 和指令级并行。以 32 个为一组，这些线程被一起阻塞并一起执行。GPU 内部的多线程处理器执行这些线程块，一个 GPU 由 8 到 32 个这样的多线程处理器组成。

6.6.1　NVIDIA GPU 体系结构简介

我们使用 NVIDIA 系统作为示例，因为它们是 GPU 体系结构的代表。具体来说，我们使用 CUDA 并行编程语言的术语，并以 Fermi 体系结构作为示例。

与向量体系结构一样，GPU 仅适用于数据级并行问题。这两种体系结构都具有聚集 – 分散数据传输，不过 GPU 处理器比向量处理器拥有更多的寄存器。与大多数向量体系结构不同，GPU 还依赖于单个多线程 SIMD 处理器中的硬件多线程来隐藏存储器延迟（见 6.4 节）。

多线程 SIMD 处理器类似于向量处理器，但是前者有许多并行功能单元，而不是像后者一样只有少数深度流水的功能单元。

正如上文所述，GPU 中包含一个多线程 SIMD 处理器的集合；也就是说，GPU 是由多线程 SIMD 处理器组成的 MIMD。例如，NVIDIA 的 Fermi 架构有四种不同价位的具体实现，分别包括 7、11、14 或 15 个多线程 SIMD 处理器。为了在具有不同数量的多线程 SIMD 处理器的 GPU 模型之间提供透明的可扩展性，线程块调度器硬件为多线程 SIMD 处理器分配线程块。图 6-9 是多线程 SIMD 处理器的简化框图。

让我们再向下深入一层细节，硬件创建、管理、调度和执行的机器对象是一个 SIMD 指令的线程，也称为 SIMD 线程。它是一个传统的线程，但只包含 SIMD 指令。这些 SIMD 线程有自己的程序计数器，它们运行在多线程 SIMD 处理器上。SIMD 线程调度器包含一个控制器，该控制器知道 SIMD 指令的哪些线程已准备好运行，然后将它们发送到调度单元，以便在多线程 SIMD 处理器上运行。SIMD 线程调度器与传统多线程处理器中的硬件线程调度器相同（见 6.4 节），区别仅在于它是调度 SIMD 指令的线程。因此，GPU 硬件有两级硬件调度器：

1. 线程块调度器，用于多线程 SIMD 处理器分配线程块。
2. SIMD 线程调度器，位于 SIMD 处理器内部，可在 SIMD 线程运行时进行调度。

这些线程的 SIMD 指令宽度为 32，因此每个 SIMD 指令线程都将计算 32 个计算元素。

由于线程由 SIMD 指令组成，因此 SIMD 处理器必须具有并行功能单元才能执行操作。我们称其为 SIMD 通道，这与 6.3 节中的向量通道非常类似。

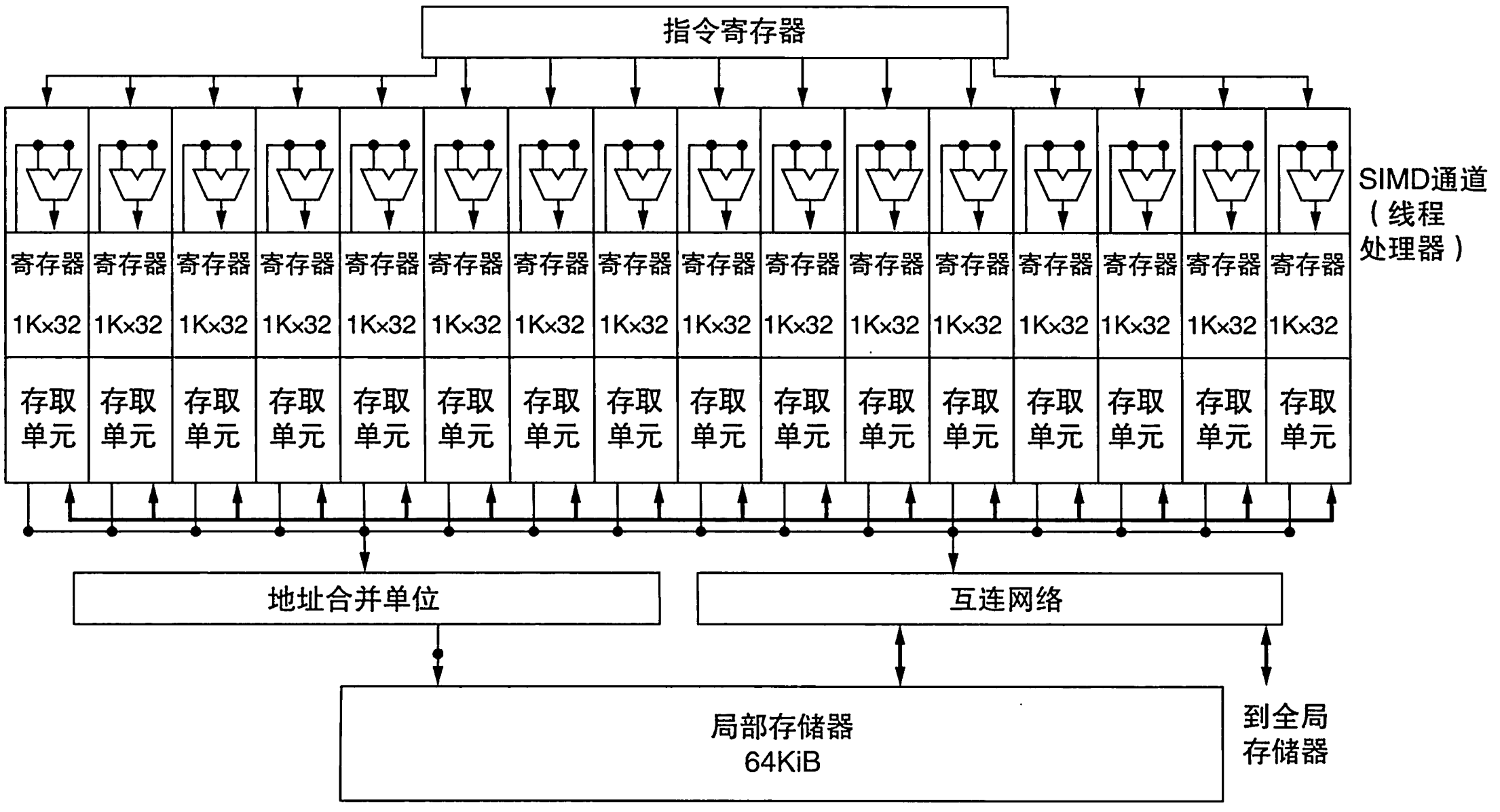


图 6-9　一个多线程 SIMD 处理器数据通路的简化框图。它有 16 个 SIMD 通道。SIMD 线程调度器中有许多相互独立的 SIMD 线程，它们可以被选择运行在此处理器上

详细阐述　每个 SIMD 处理器的通道数量因 GPU 的版本而异。在 Fermi 中，每个宽度为 32 的 SIMD 指令线程都被映射到 16 个 SIMD 通道，因此 SIMD 指令线程中的每条 SIMD 指令都需要两个时钟周期才能完成。SIMD 指令的每个线程都是同步执行的。让我们继续将 SIMD 处理器与向量处理器做类比，你可以说它有 16 个通道，并且向量长度为 32。这个宽而浅的性质是我们使用术语 SIMD 处理器而不是向量处理器的原因，因为 SIMD 处理器这个术语更直观。

因为根据定义，SIMD 指令线程是独立的，所以 SIMD 线程调度器可以选择任何一个准备好的 SIMD 指令线程，并且不需要在一条指令执行后继续坚持执行该线程内的下一条 SIMD 指令。因此，如果使用 6.4 节中的术语来描述，SIMD 处理器使用细粒度多线程。

为了保存存储元素，Fermi SIMD 处理器具有令人印象深刻的 32768 个 32 位寄存器。就像向量处理器一样，这些寄存器在向量通道（这里是 SIMD 通道）上进行逻辑划分。每个 SIMD 线程限制为不超过 64 个寄存器，因此你可以认为 SIMD 线程至多有 64 个向量寄存器，每个向量寄存器中具有 32 个元素，每个元素为 32 位宽。

Fermi 有 16 个 SIMD 通道，因此每个通道包含 2048 个寄存器（32768/16=2048）。每个 CUDA 线程获得每个向量寄存器的一个元素。注意，CUDA 线程只是 SIMD 指令的线程的垂直切割，一个 SIMD 线程对应执行一个元素。请注意，CUDA 线程与 POSIX 线程非常不同，你不能任意地在 CUDA 线程中进行系统调用或同步。

6.6.2　NVIDIA GPU 存储结构

图 6-10 展示了 NVIDIA GPU 的存储结构。我们称每个多线程 SIMD 处理器本地的片上