第1章

体系结构发展对编译技术的影响

在过去的数十年里，摩尔定律一直支配着半导体行业的发展路线，随着晶体管尺寸的不断变小，单个芯片上集成的晶体管数量越来越多。最新的 Nvidia H100 GPU 单个芯片集成了 800 亿个晶体管，而苹果公司的 M1 Ultra 片上系统（System on Chip，SoC）中的晶体管数量更是达到了惊人的 1140 亿个。晶体管数量的增加允许芯片设计厂商可以在单个芯片上实现更多的功能和更高的计算能力。也正是日益丰富的功能和日渐增强的处理能力，使得软件开发人员可以不断地开发新的应用，从而驱动着整个信息技术产业不断向前发展。

在现代计算机系统中，编译器已经成为一个必不可少的基础软件工具。程序员通过高级语言对底层硬件进行编程，而编译器则负责将高级语言描述转换为底层硬件可以执行的机器指令。编译器在将应用程序翻译到机器指令的过程中，还需要对程序进行等价变换，从而让程序能够更加高效地在硬件上执行。在特定硬件平台和编程语言的双重约束条件下，应用程序的性能主要依赖于程序员编写并行代码的能力和编译器的优化能力。编译器还需要充分弥合上层编程模型与底层硬件的巨大鸿沟，尽可能地降低程序员的编程难度。也正因如此，编译技术的发展始终紧随着硬件架构的演化，并且扮演着越来越重要的角色。

1.1 面向经典体系结构的性能优化

从经典体系结构的角度，提升硬件性能主要有三类方法：第一类方法是将更多的精力用于发掘各种并行性；第二类方法是引入新的存储层次来缓解访存速度和存储容量之间的矛盾；第三类方法则是通过定制化的方法来提升硬件处理领域相关应用的性能。

1.1.1 并行性发掘

应用程序中的并行性可以分为数据级并行（data level parallelism）和任务级并行（task level parallelism）。其中，数据级并行是指同时操作多个数据，任务级并行则是通过多个并行而且独立的任务处理多个数据。在此基础上，计算机硬件可以通过指令

级并行（instruction level parallelism）、单指令多数据并行（single instruction multiple data parallelism）、线程级并行（thread level parallelism）和请求级并行（request level parallelism）四种方式来实现数据级并行和任务级并行。

1. 指令级并行

一个处理器核心通常是指能够独立地从至少一个指令流获取和执行指令的处理单元，通常包含取指单元、译码单元、执行单元、访存单元和程序计数器和寄存器文件等逻辑单元。指令级并行是指在单个处理器核心中，通过同时执行多条指令的方式来提高处理速度。通过巧妙地设计流水线结构，可以大幅度地提升单个处理器核心的指令级并行度。然而，受限于硬件复杂度，处理器中正在执行的指令总数与流水线的深度和个数成正比。处理器中正在运行的指令总数决定了处理器的复杂度。

指令级并行是传统系统结构领域中较为成熟的一种并行方式，代表性的技术包括流水线（pipeline）、多发射（multiple issue）、同时多线程（simultaneous multithreading）和超长指令字（very long instruction word）等。其中，流水线并行是将每个功能单元通过分时复用的方式在不同的指令之间共享，是一种时间上的并行；而多发射则是在每个时钟周期发射多条指令到多个流水线中并行执行，是一种空间上的并行；将时间和空间维度的并行组合起来就形成了同时多线程技术和超长指令字技术。

2. 单指令多数据并行

单指令多数据并行是一种空间维度上的并行处理模式，典型的实现方式是在硬件中集成向量运算部件，用单条向量指令同时处理多个数据，例如 Intel 的 SSE/AVX、ARM 的 NEON/SVE 等。这种并行方式不仅可以充分挖掘程序中潜在的并行性，还可以精简指令序列。向量化不需要对硬件结构特别是流水线结构做大量的修改，通常只需要修改数据通路的位宽。对于图 1.1 所示的代码，如果运行在普通的标量处理器上，编译器需要生成一个循环，通过 16 次迭代来完成整个计算任务，一次迭代完成一个元素的自增运算。如果运行在向量宽度为 4 的向量处理器上，编译器只需要生成 4 条向量加法指令即可完成计算，不需要插入任何分支跳转指令。

```
1 for (i = 0; i < 16; i++)
2   x[i] = x[i] +1;
```

图 1.1　循环级并行的基本模式

3. 线程级并行

线程级并行既可以在单处理器核心内实现，也可以在多处理器核心内实现。在支持硬件多线程技术的单处理器中，可以通过硬件多线程或者同时多线程等技术来提高资源利用率和计算效率。但是，在设计复杂度和功耗墙问题的共同约束下，提升单处理器的

性能已经非常困难。一个自然的想法就是在单个芯片上集成多个处理器核心，通过挖掘多个处理器核心之间的线程级并行能力来提升总体的计算能力。

在线程级并行模式中，不同执行单元的指令流相互独立，可以以同步或者异步执行的方式处理各自的数据。主要的实现方式有对称多处理器（Symmetric Multi-Processing, SMP）结构、分布式共享存储（Distributed Shared Memory, DSM）结构、大规模并行处理器（Massively Parallel Processing, MPP）结构、集群（cluster）。与线程级并行紧密相关的是被 GPU 架构普遍采用的单指令多线程技术，即多个线程以锁步的形式执行相同的指令，但是每个线程处理不同的数据。

4. 请求级并行

线程级并行要求多个线程或进程在紧耦合的硬件系统中通过协作的方式完成一个计算任务，而请求级并行则是让多个独立并且可以并行工作的线程或进程完成各自的计算任务。请求级并行在 Flynn 分类法[32] 中属于多指令多数据（Multiple Instructions Multiple Data，MIMD）并行。与指令级并行、单指令多数据并行和线程级并行相比，请求级并行的粒度更粗，而且通常需要程序员和操作系统的密切配合才能实现，而前面三种并行方式则更多地依赖编译器和底层硬件的支持。

1.1.2　存储层次结构

随着体系结构技术的飞速发展，处理器执行指令的速度远远超过了主存的访问速度。这种日益拉大的速度差异对计算机体系结构的发展产生了巨大的影响。为了弥合这种巨大的鸿沟，处理器中必须要设置由不同容量和不同访问速度的存储器构成的存储层次。存储层次可以提高程序性能的原因是保证了 CPU 大部分数据的访问时延都比较低。随着处理器计算速度和访存速度的差距越来越大，存储层次结构的设计显得越来越重要。

在典型的现代处理器中，每个 CPU 核有私有的 L1 Cache，一组 CPU 核有一个共享的 L2 Cache，而 L3 Cache 通常会被所有的处理器核共享，Cache 的访问速度往往与容量成反比。容量越大，访问速度越低。由于程序中的时间局部性和空间局部性，Cache 可以作为一个有效的硬件结构将经常使用的数据保持在离处理器较近的位置。为此，Cache 需要尽可能多地保留最近使用的数据，在容量有限的前提下尽可能地替换出最近很少使用的数据。现代体系结构中的 Cache 通常是由硬件自动管理的，但是为了极致优化性能，程序员和编译器仍需要知道存储层次的相关信息。除了硬件自动管理的 Cache 以外，现代体系结构往往还有软件自主管理的便签式存储器（scratchpad memory）。

存储层次设计一个非常重要的方面是对数据一致性的维护。便签式存储器的管理和一致性维护通常由软件完成，而对 Cache 的管理和一致性维护则由硬件完成。为了降低硬件成本，简化存储管理和数据一致性的维护，经典体系结构的存储层次通常设计成金字塔形。在金字塔形存储层次中，越靠近运算部件的存储器，容量越小，速度越快，成

本越高；位于金字塔不同层次的存储器，只和它相邻的上下层存储器交换数据；存储层次之间的数据一致性由 Cache 一致性协议来保证。

然而，在一些领域专用架构中，金字塔形存储层次往往无法满足具体应用的访存需求。例如，对于大规模流式计算来说，因为数据的空间局部性和时间局部性较差，Cache 的访存加速效果也会大打折扣。另外，对于同时存在多条具有不同特征的数据流的应用场景，对于流不敏感的金字塔形存储层次也不能很好地满足不同数据流的访存需求。例如，在深度神经网络的前向计算过程中，卷积层和全连接层的权值为常量而且数据量较小，而不同的输入和输出神经元则是变量，而且数据量通常很大，本层的输出神经元又会成为下一层的输入神经元。从宏观上看，领域专用架构的存储层次仍然是金字塔形结构，由片外的低速、大容量存储器和片上的高速、小容量存储器组成；但是，在微观结构上，通常还会设计一些非金字塔形的存储结构来更好地适应具体领域的计算和访存模式。

1.1.3 领域专用架构

领域专用架构（domain-specific architecture）专用性强，反而可以使得逻辑设计更加趋于简单，在设计思路上也更加开放，不用受到传统指令集和生态因素的制约，已经在大数据处理、数字信号处理、密码学、高性能计算、图形学、图像处理和人工智能等领域获得了广泛的应用。领域专用处理器一般会根据应用的具体特点，定制运算单元，简化控制逻辑，设计与领域计算特征相适应的存储结构和数据通路，虽然牺牲了通用性和灵活性，却获得了较高的性能和能效比。特别是在第三次人工智能浪潮中，涌现出大量的面向人工智能应用的领域专用处理器，这些人工智能处理器在功耗、性能和集成度等方面较传统的 CPU、GPU 和 FPGA 都有很大的优势。

领域专用架构体现出领域的专注性。以引爆第三次人工智能浪潮的深度学习算法为例，其主要特点是计算量和输入与输出（Input and Output, IO）数据量都比较大，而且并行度较高。这要求面向人工智能应用的领域专用处理器（简称人工智能处理器）在存储结构、带宽和算力配置以及互连结构上做大量的定制化设计。为了满足海量数据在计算单元和存储单元之间的高速传输需求，人工智能处理器不仅要具备与计算模式匹配的存储结构，还要在计算单元和存储单元之间具有高速的通信链路。为了满足深度学习的算力需求和计算模式需求，人工智能处理器不仅要集成大规模的并行计算单元，还要能够高效地处理深度学习算法中常见的卷积、全连接和池化等操作。为了适应深度学习算法的典型计算模式，人工智能处理器在结构设计时还要考虑将不同的计算单元和存储模块有机地结合在一起，尽量降低相关操作之间的数据共享开销。

可以说深度学习算法既是计算密集的，又是访存密集的。其中计算密集的代表是卷积运算，而访存密集的代表则是全连接运算。

卷积神经网络（Convolutional Neural Network, CNN）的主要计算量也都来自卷积层。以 GoogleNet 为例，卷积计算量会占到总计算量的 90% 左右。因此，加速卷积运

算是提升深度学习应用性能的核心任务。卷积层的输入是神经元和权值，经过图 1.2 所示的 N 重循环处理后，得到输出神经元。

```
//HO 表示输出神经元的行数
for (row=0;row<HO;row++){
  //WO 表示输出神经元的列数
  for (col=0;col<WO;col++){
    //CO 表示输出神经元的通道数
    for (co=0;co<CO;co++){
      //CI 表示输入神经元的通道数
      for (ci=0;ci<CI;ci++){
        //KH 表示卷积核的行数
        for (i=0;i<KH;i++){
          //KW 表示卷积核的列数
          for (j=0;j<KW;j++){
            Out[co][row][col]+=
                W[co][ci][i][j]*In[ci][stride*row+i][stride*col+j];
          }
        }
      }
    }
  }
}
```

图 1.2　卷积运算的循环表示 (1)

一个卷积层包含 CO 个 CI×KH×KW 的卷积核，共计 CO×CI×KH×KW 个权值，每个卷积核对应输出神经元的一个通道。每个输出神经元涉及 CI×KH×KW 次乘累加运算，每个卷积层总的乘累加运算次数为 HO×WO×CO×CI×KH×KW。

卷积运算是一种典型的 Stencil 运算，Stencil 运算的特点是输入输出数据组织成一个多维网格，一个输出数据点的值只与邻近点的输入点有关，虽然每个输出点都可以独立计算，但是运算密度较低，依赖关系复杂。Stencil 计算模式在流体力学、元胞自动机、天气预报、粒子模拟仿真、电磁学等领域的数值计算中都有广泛的应用。为了在领域专用处理架构上优化这类程序的性能，需要合理地组织计算次序和数据布局，充分发掘数据局部性和计算并行性。

作为访存密集型的代表运算，全连接运算本质上是矩阵向量运算，可以用图 1.3 所示的两层嵌套循环来表示。全连接运算的特点是，R 和 C 比较大，因而权值的 IO 数据量比较大；二维权值中的每个元素都只参与一次乘累加运算，权值局部性比较差，整个算法的计算密度较低。

1. 领域专用架构的存储层次

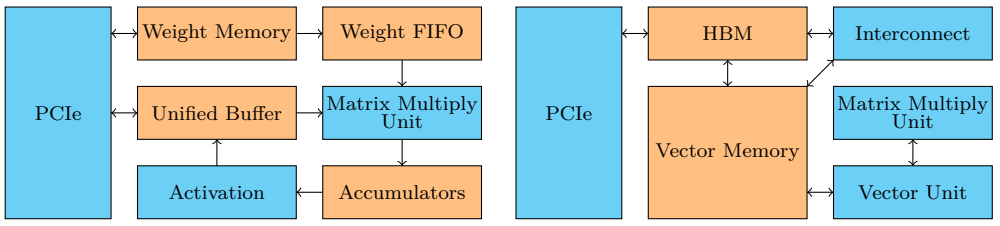
卷积和全连接运算本质上都是在执行乘累加运算，而且都可以转换为矩阵运算。对于全连接运算，可以将 N 维向量看成是 $N \times 1$ 的矩阵；对于卷积运算，可以利用

```
for (row=0;row<R;row++){
  Out[row] = 0; /*S1*/
  for (col=0;col<C;col++){
    Out[row] += w[row][col]*In[col]; /*S2*/
  }
}
```

图 1.3 卷积运算的循环表示 (2)

Img2Col 操作[23] 转换为矩阵运算。除了乘累加运算，深度学习中还有大量的向量对位运算，例如，偏置运算和激活运算。面向人工智能应用的领域专用架构为了能够高效地处理矩阵乘累加运算和向量对位运算，通常会集成专用的矩阵运算单元和向量运算单元，例如华为的达芬奇架构[54] 和 Google 的 TPU[41]。

为了能够高效地处理深度神经网络中的典型运算，面向深度学习应用的领域专用架构还需要设计适应运算特点的存储层次。图 1.4 给出了 Google TPU 的组织结构，作为 TPU 的核心功能部件的矩阵乘单元（matrix multiply unit），分别从权值队列（weight FIFO）和神经元缓冲区（unified buffer）读取参与矩阵运算的只读权值和输入神经元数据，矩阵运算的结果首先被保存在片上的累加器缓冲区（accumulators）中，经过激活和池化等操作后再写回神经元缓冲区，这样就完成了一轮卷积运算。权值队列从片外的权值存储器（weight memory）读取只读的权值数据，神经元缓冲区则从主机侧读取参与运算的神经元数据。显然，TPU 的存储层次已经不再是经典体系结构中的金字塔形结构，而是根据权值和神经元的运算特点，分别设计了专门的数据通路和专属的存储器。类似的设计在其他面向人工智能应用的领域专用架构中也普遍存在，而对于这类非金字塔形存储层次的管理通常由程序员或者编译器完成。



(a) TPU v1 的存储层次　　(b) TPU v2/v3 的存储层次

图 1.4 Google TPU 的组织结构

2. 领域专用的计算功能部件

除了像 Google TPU 这样的领域专用芯片，在具有传统金字塔形存储层次结构的 GPU 上也为深度学习提供了专用的计算功能部件。Nvidia 最先在 V100 GPU 上集成了用于实现矩阵乘累加运算功能的 Tensor Core，可以用于实现形如 $\boldsymbol{D} = \boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{C}$ 的

矩阵乘累加操作。在 CUDA 编程模型中可以使用 wmma（warp-level matrix multiply and accumulate) 系列接口来操纵 Tensor Core，每个 wmma 接口的内部则是通过调用一系列基础的 mma（matrix multiply and accumulate）操作来实现矩阵块的乘累加运算。为了将 Tensor Core 融合到 SIMT 的编程模型中，CUDA 将一个 Warp 中的 32 个线程分成 8 组，每组称为一个线程组（thread group），每两个线程组称为一个线程组对（thread group pair），每个线程组对中的 8 个线程协作完成一次 Tensor Core 上的 mma 操作。一次 mma 所需要的线程和对应矩阵元素的分布示意如图 1.5 所示。其中，带颜色的方块表示矩阵的一个元素，带颜色的圆圈表示一个线程；一个方块上的数字表示该矩阵元素由哪个线程执行，圆圈内的数字表示该线程的编号。

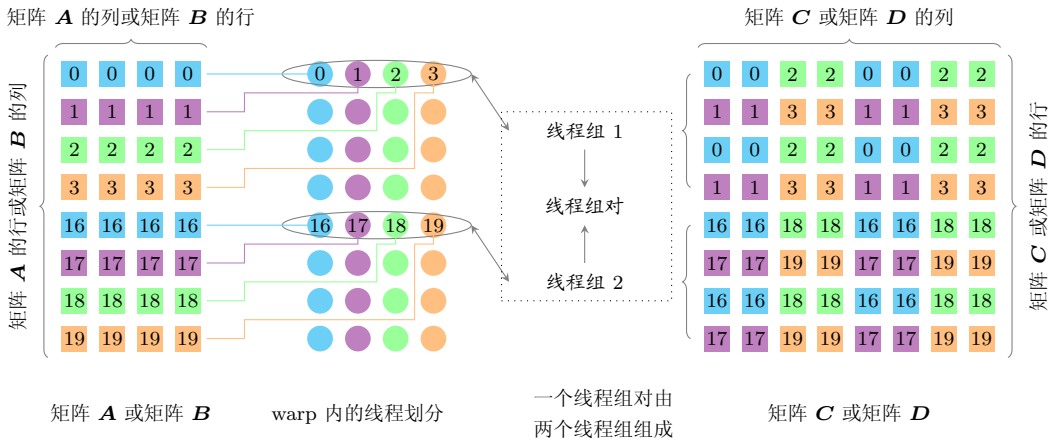


图 1.5　一次 mma 所需要的线程和对应矩阵元素的分布示意

操作矩阵 $\boldsymbol{A}$ 或矩阵 $\boldsymbol{B}$ 的一行或一列内相邻的数据元素（在图中用相同的颜色表示）会被批量载入同一个线程的寄存器中，类似地，矩阵块 $\boldsymbol{C}$ 或矩阵 $\boldsymbol{D}$ 中的相邻元素也会被保存在同一个线程的寄存器中。对于任意规模的矩阵乘加操作，由编译器完成从任意规模的输入矩阵到 wmma 和 mma 操作支持的矩阵规模的分解和变换。

1.2　编译器面临的挑战

前面从体系结构的角度介绍了提升硬件性能的三类主要方法，即发掘并行性、优化存储层次和设计领域专用架构。但是，体系结构设计仅仅是确定了软件的执行平台，而在确定的执行平台上如何优化程序，则是程序员和编译器的职责。为了充分利用硬件的处理能力，软件开发人员和编译器都需要充分了解底层体系结构的特点。对于程序员来说，编写正确而且高效的串行程序是非常困难的，编写正确而且高效的并行程序则更加困难，其难度会随着并行粒度的降低而增长。为了减轻程序员的工作，编译器需要担起性能优化的重担。特别地，随着非金字塔形存储结构在各个领域专用架构（例如，人工

智能芯片、网络处理器和数据处理器）的普遍应用，以及各种定制化加速器部件的持续集成，程序员手工编写高性能程序变得越来越困难，而编译器则成为上层应用与体系结构之间一个至关重要的工具。

狭义的程序优化是指尽可能地发掘程序的并行性和硬件的计算能力，使得代码在具体硬件上的执行速度达到最高。而广义上的程序优化是指，编译器为了改进应用程序的某项技术指标（例如，执行时间、代码尺寸和内存占用），在不改变程序语义的前提下，修改其内部数据结构或所生成代码的过程。优化可以是架构无关的，也可以是架构相关的。为了把程序员使用高级语言编写的与架构无关的程序转换为可以在具体硬件上高效执行的机器指令，编译器需要在程序分析、代码优化和代码生成这三个核心步骤持续引入各种新的技术和方法。常规的标量优化技术包括循环不变量外提、常量传播、公共表达式删除、运算强度削弱、循环展开、软件流水、指令调度等。

并行性、局部性和领域专用设计为编译优化带来了巨大的挑战，这些挑战在以深度学习为优化目标的背景下同样适用，并且如何在这些优化目标之间进行权衡取舍的问题也显得日益尖锐。而这三个问题都与循环有着紧密的联系，因为大多数的计算机程序几乎将所有运行时间都花费在循环内，因此为循环优化付出的努力往往能得到最高的回报。但是，对应用程序的优化往往需要程序员和编译器的共同努力。例如，程序员需要对循环嵌套结构进行拆分和重写等操作，来帮助编译器更好地识别和优化循环。

1.2.1 并行性发掘

并行性发掘可以通过软件静态完成，也可以通过硬件动态地进行。完全依赖硬件管理其并行性的机器称为超标量处理器，而完全依赖编译器来管理其并发性的机器称为超长指令字。在一些成本或者功耗敏感的嵌入式应用场景（例如物联网）中，通常依赖编译器来发掘并行性，而在桌面计算、服务器和云计算场景中，虽然硬件实现了强大的调度机制，也需要在编译器的配合下才能充分发挥硬件的指令级并行性能力。

编译器开发指令级并行性的基本思路是结合硬件的结构特点，通过指令调度的方法来提高硬件资源的利用率，常用的提高指令级并行性的方法如表 1.1 所示。

表 1.1 编译器常用的提高指令级并行性的方法

技　术	目　标
静态分支预测	减少控制依赖造成的流水线停顿
延迟槽调度	避免流水线空泡
静态指令调度	解决数据依赖造成的流水线停顿
循环展开	解决控制依赖造成的流水线停顿
软件流水	用计算时间隐藏访存时间

流水线的结构冲突、数据依赖和控制依赖会在处理器流水线中引入空泡。为了避免因流水线空泡导致的性能损失，通常需要编译器对指令序列进行静态调度。例如，静态

分析预测可以结合硬件的推测执行功能减少因控制流引入的流水线空泡，延迟槽调度则是用一些与控制流无关的指令来填充分支跳转指令的延迟槽，常规的静态指令调度则是将一些没有数据依赖的指令安排在一起，以避免因数据依赖引入的流水线空泡。

指令级并行在过去数十年中已经被深度发掘，并成为 LLVM、GCC 等传统编译器的基础功能。编译器支持指令级并行的基本方法是在基本块内实现简单的指令调度。然而，在一个基本块级别开发并行性的收益并不大。为了进一步发掘并行性，必须着力于开发更高层级的并行。

本书中重点介绍的多面体模型主要用于开发单指令多数据并行性和线程级并行性，而对请求级并行的开发则不在本书的讨论范围。对单指令多数据并行和线程级并行的优化通常是以循环为研究对象的，因此也称为循环级优化。循环级优化技术尝试从多个循环迭代中寻找并行性，通过重复执行多个循环体中的代码来提高程序的并行性。循环级并行优化可以显著提升程序的整体性能，因为程序中运行时间较长的代码片段通常是具有很多次迭代的循环。

循环级优化的首要目标是充分发挥硬件提供的计算能力，以减少延迟（尽可能快地完成计算任务）或者提高吞吐率（在相同的时间内完成更多的任务）。循环级优化的基础方法包括循环展开、软流水、并行化和向量化。其中，循环展开是通过复制循环指令的方式来避免循环的控制开销；软流水主要是对原始循环进行重新组合，将无依赖的访存和计算指令组合在一次循环迭代中；向量化和并行化的目的是要对处理大量数据或者可划分后能并行执行的程序改善性能。

循环向量化是编译器将循环中的标量运算转换为向量运算的过程，最终生成的向量运算指令会在同一个硬件单元上执行。并行化则是把循环拆分成多个可以并行执行的子任务，在不同的硬件单元上并行执行每个子任务。循环并行化需要将循环进行划分，并让多个处理器分别执行其中的一部分。但是为了保证性能，还必须把处理器之间的通信量减少到最小。通信量实际上与数据局部性紧密相关，对于处理器经常访问的数据，在任务或数据划分时应当让它离处理器更近，否则会引入大量的通信和同步操作。

然而，循环向量化和循环并行化通常是两个互相冲突的优化目标，因为循环并行化通常在外层循环进行，而循环向量化则必须在内层循环进行。自动并行化和自动向量化是现代编译器面临的主要困难，原因是编译器很难有效地收集和分析向量化或者并行化所需要的数据相关性和控制相关性等信息。由于编译器在自动向量化和自动并行化方面遇到了困难，软件开发人员不得不自己发掘应用的并行性，并且处理共享资源的访问冲突，这就要求编译器和编译语言能够提供一种易于描述并行性的编程模型。

与此同时，在以大量的矩阵乘法和卷积运算为中心的深度神经网络中，循环嵌套的维度也随着增加，不同运算循环嵌套结构之间可能存在较大差异。在开发外层循环的并行化和内层循环的向量化时，优化编译器的任务不再是简单地识别某一个或多个循环维度上的并行性，而是要在一个循环嵌套内多个循环维度上的不同并行性内寻找到能够最大限度地提升程序性能的并行执行方式，这就需要优化编译器在原有的识别并行性的基础上，还要具备同时组合多种循环变换的能力。

1.2.2 局部性发掘

存储器的容量越大，则访问速度越慢。幸好程序访问的数据通常具有空间局部性和时间局部性。时间局部性通常是指某个数据被访问后，常常会在较近的时间再次被访问；空间局部性则是指某个数据被访问以后，常常会在较近的时间再访问与其相邻的数据。可以在内存之上设置多个存储层级来利用数据的局部性改进程序性能。性能调优一个非常重要的前提就是理解计算机系统的存储层次，并且结合硬件的存储层次和计算特征，进行特定的访存优化从而平衡访存和计算。

图 1.2 所示的卷积过程可以通过循环变换转换成不同的实现方式和数据、时间和空间局部性特征，而不同的实现在不同的硬件上性能差异也非常大。我们以图 1.3 所示的矩阵与向量乘法运算 $\boldsymbol{b} = \boldsymbol{A}^{\mathrm{T}}_{M\times N}\boldsymbol{x}$ 为例，当 $M << N$（M 远小于 N）时，每计算一个值 $b[\mathrm{col}]$，会导致列向量 $\boldsymbol{x}$ 和矩阵 $\boldsymbol{A}$ 被不断地换入和换出，严重影响性能。经过分析可以发现，在整个计算过程中，矩阵 $\boldsymbol{A}$ 中的每个元素只会被使用一次。因此，最理想的情况下，$\boldsymbol{A}$ 的数据元素只需要被加载到 Cache 一次，而列向量 $\boldsymbol{x}$ 则会被反复使用 M 次。Cache 优化的最终目标就是确保矩阵 $\boldsymbol{A}$ 的数据元素只被加载到 Cache 一次，$\boldsymbol{x}$ 的元素只被加载一次。因此，可以将图 1.3 所示的源代码转换为图 1.6 所示的形式。

```
1   for (n=0; n < N/K; n++){
2     for (row=0; row<M; row++){
3       for (k=0; k < K; k++) {
4         int col = n*K + k;
5         if (row==0)
6           b[col]=A[row][col]*x[col];
7         else
8           b[col] += A[row][col]*x[col];
9       }
10    }
11  }
```

图 1.6 在 CPU 上实现矩阵与向量乘的优化方法

访存局部性优化比较成熟的手段是循环分块（loop tiling）和循环融合（loop fusion）。循环分块的基本思想是通过将大块的循环迭代拆解成若干较小的循环迭代块，减少一个内存单元的数据重用周期，这种重用周期既可以是时间上的也可以是空间上的。循环融合则是将多个具有“生产–消费”关系的循环合并在一起，从而缩短了每个中间数据的重用周期，避免中间结果被换出高速缓存。循环分块和循环融合都可以确保某个内存地址单元被加载到 Cache 以后，该内存地址单元会尽量保留在 Cache 中，这样就可以达到减少片外访存开销的目的。然而，循环分块和循环融合也是两个相互冲突的优化目标，因为循环融合会导致循环分块的约束增加，灵活性变差。

特别地，循环分块和循环融合在当前的领域专用架构上扮演着至关重要的作用。与