后摩尔时代 高性能软件开发的挑战

关键词：摩尔定律　后摩尔时代　高性能开发

P. (Saday) Sadayappan
犹他大学

大规模集成电路的后摩尔时代

摩尔定律 (Moore's Law) 是指当价格不变，集成电路上可容纳的元器件的数目，每隔 18~24 个月就会增加一倍，性能亦将提高一倍。自诞生之日起，摩尔定律就成为计算机领域最重要的定律之一，预测和指导了整个信息技术产业的蓬勃发展。然而，当前全球半导体行业已经逐渐认可一直在推动 IT 行业前进的摩尔定律正在走向终结。那么在后摩尔时代，高性能软件的开发将面临哪些棘手的挑战呢？

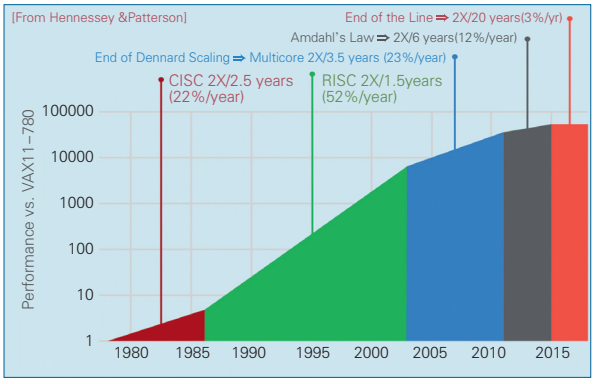

图 1　摩尔定律的终结

图 1 展示了过去近四十年计算机芯片处理速度的增长趋势。在 20 世纪 80 年代，芯片设计和制造工艺的进步和高级编程语言的问世极大程度地推动了指令集的发展和完善。精简指令集 (RISC)、多级缓存和先进编译器（特别是在寄存器分配方面）等技术创新使得处理器性能每年以高达 50% 的速度增长，由此，开创了计算机架构的黄金时代。随着处理器性能的不断攀升和晶体管体积的缩小，能耗成为架构设计的关键约束条件。进入 21 世纪后，预言晶体管能耗随着其面积的缩小而降低的登纳德缩放定律 (Dennard Scaling) 开始显著放缓，单核速度性能提升的比例逐年下降，芯片多核时代随之来临。然而，多核技术并不能解决登纳德缩放比例定律结束所带来的能耗问题。这是因为阿姆达尔定律 (Amdahl's Law) 指出并行计算机的加速比严重受制于串行执行部分的速度。2011 年至 2015 年，晶体管数量每 6 年才翻一番。更不幸的是，自 2015 年起，单核速度性能提升的比例由上世纪 80 年代的每年 22% 降低至现在的 3%。随着摩尔定律失效、登纳德缩放比例定律减速、阿姆达尔定律继续发挥影响，我们将如何加快处理速度？

摩尔定律的终结给我们带来了很多启示。首先，既然晶体管的数量无法增长，那么我们需要关心的问题就是如何更高效地利用芯片上有限的晶体管。专用硬件构建模块被认为是主要的解决方案，例如谷歌的 TPU、Volta 的 Tensor Cores 等。除此以外，我们可以对层次存储中的数据移动进行更多更明确的控制。当然，我们还可以使用非冯·诺伊曼 (Non von Neumann) 的体系结构。然而，硬件体系结构

的变化将使得高性能软件的开发工作比以往更加困难。因此，一个很有潜力的研究方向是通过编译器有效利用这些日益复杂的定制架构。如此一来，我们将发挥出不同类型架构的特性，获得软件高性能的同时，提高应用程序设计与实现的高生产率和可移植性。

目前编译器最大的挑战在于优化数据移动 (data movement)。硬件的浮点计算能力已经十分充足，而数据移动的时间开销和能耗开销却占比巨大。因此，如何为数据移动建模并进行优化，已然成为一个巨大的难题。不同于算法的计算复杂度，许多算法的数据移动复杂度是未知的。

数据移动的代价

数据移动的代价远远超过计算本身的代价。众所周知，芯片内部工艺尺寸变小，单位晶体管的数量会随之增加。由于我们只需要相同数量的晶体管来执行浮点运算，因此浮点数计算的开销随着工艺的进步而显著降低。如图 2 所示，11 纳米工艺下的双精度浮点操作指令功耗降低到了 45 纳米工艺的十分之一。与计算相比，数据移动的开销明显要高出数个量级。而且，随着工艺的提高，数据移动的能耗虽然有所下降，但降幅比较小。例如，11 纳米工艺与 45 纳米工艺相比，off-chip/DRAM 上的功耗只降低到原来的 42.5%。通过观察，我们认为 FLOPs 的能耗极低，而数据移动的开销才是性能的瓶颈。因此只有控制数据移动的量，才能控制总的功耗。

2008 年塞缪尔·威廉姆斯 (Samuel Williams) 等人提出了通用的 Roofline 性能模型，使用运算强度 (Operational internsity)、运算性能峰值和带宽峰值（如内存带宽）三个基本参数来分析浮点数算法性能的理论极限。图 3 展示了关于英伟达不同代际 GPU 的 Roofline 图，包括 Fermi、Maxwell、Kepler、Pascal、Volta。操作密度 (operational intensity) 是指对于每个移动到内存以便处理器处理的字节，它所需要执行的操作数量。当操作密度为 1 时，即对于要移至内存的每个字节数据，都需要执行一个操作。峰值性

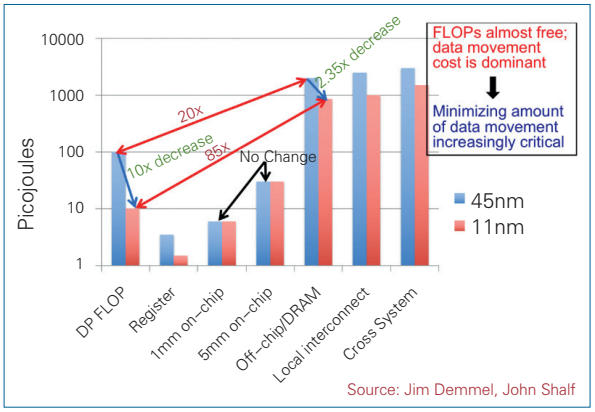

图 2　芯片的能耗趋势

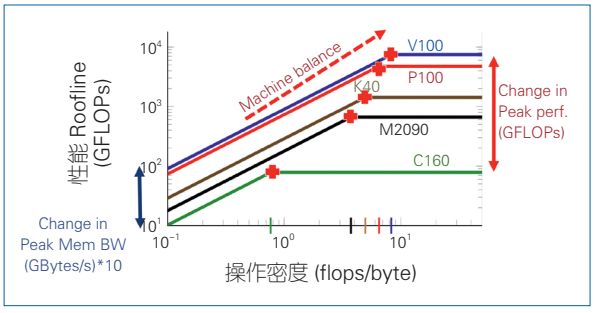

图 3　计算和数据移动的性能趋势

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j];
```

(a) 原始版本

```
for(it = 1; it<N−1; it +=B)
  for(jt = 1; jt<N−1; jt +=B)
    for(i = it; i < min(it+B, N−1); i++)
      for(j = jt; j < min(jt+B, N−1); j++)
        A[i][j] = A[i−1][j] + A[i][j−1];
```

(b) 循环分块版本

图 4　数据移动的计算复杂度

能决定了“屋顶”的高度，即水平轴，而内存的峰值带宽决定了“房檐”的斜率。在图 3 中，通过增加操作密度，我们能够发现，由于内存的带宽峰值不同，V100 等高端显卡的 GFLOPs 显著高于其他显卡。因此，浮点计算的性能峰值增长快于内存的带宽峰值。

判断算法受限于内存带宽还是处理器的处理能力需要引入机器平衡点 (machine balance) 这一概念。简而言之，机器平衡点等于峰值运算能力与峰值带宽的比值。为了避免受限于内存带宽峰值，我们需要为计算提供比机器平衡点更高的操作密度，这意味着对于每个操作，需要更少的数据移动。针对现在的发展趋势，我们需要进行更加严格的控制，更好地优化数据移动。

需要指出的是，数据移动的复杂度在表征和理解上比计算复杂度要困难得多。如图 4(a) 和图 4(b) 所示，相同功能操作的代码计算复杂度虽然相同，但是其数据移动的复杂度却完全不一样。图 4(a) 中的红色部分代码是一个简单的双层嵌套循环，目的是为了通过将 A[i][j-1] 和 A[i-1][j] 相加，产生 A[i][j]。而图 4(b) 中的蓝色部分代码，是红色部分代码的展开版本。循环分块 (tiling) 是众所周知的良好转换，它的基本思路是，利用在小块数据的访问，提高二级高速缓存的使用。这两个版本的计算复杂度是一样的，都是 (N-2) × (N-2)。

循环分块是为了减少缓存未命中的情况，从而减少数据移动。图 5 是对这两个版本代码缓存未命中次数的统计。x 轴是机器的缓存大小，y 轴是 read 操作未命中的次数。这里我们使用了一个简单的最近很少使用 (Least Recently Used, LRU) 缓存。可以看出，蓝色循环分块版本代码数据移动的开销和红色代码的不同。而对于同一版本的代码，数据移动的开销和机器的缓存大小有关。当缓存足够大时，红色版本和蓝色版本代码的未命中率没有差别。相反的，算法的计算复杂度本质上不依赖于系统参数。因此对于循环的各种近似逼近的版本，算法复杂度是不变的，而数据移动会有很大的差别。

那么，是否有版本能够达到比分块版本更低的缓存未命中率呢？一种可能的办法是通过遍历所有可能的版本，逐个分析缓存未命中率。由此引发了另一个问题，对于所有可能的等价版本，可达到的最低的数据移动量是多少？当前的性能分析工具似乎还不能解决这个问题。

通过上述对比我们可以发现，当前的体系结构中数据移动是主要的限制，那么，我们可以做些什么来改善它呢？几十年前，有项开创性的工作曾尝试解决这个问题 [1]。他们的基本想法是：首先把数据的推理抽象为计算有向无环图 (Computational Directed Acyclic Graph, CDAG)，然后再对其进行分析和优化。图 6 描述了当 N=6 时，图 4 中算法的一个完整计算流程，对于执行操作的每个计算实例，都对应 CDAG 图中的一个顶点。CDAG 中的边是从一个操作到另一个操作的数据流。

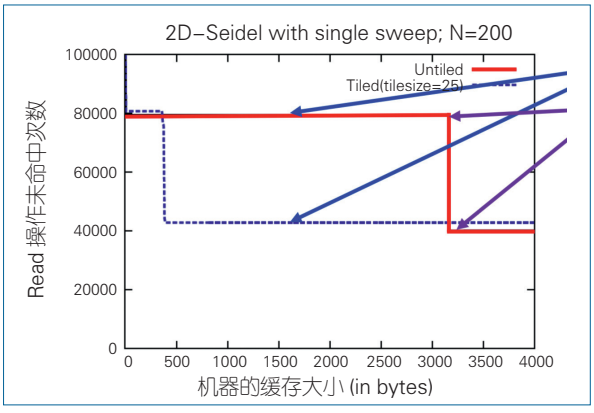

图 5　不同版本代码 Read 操作未命中缓存次数

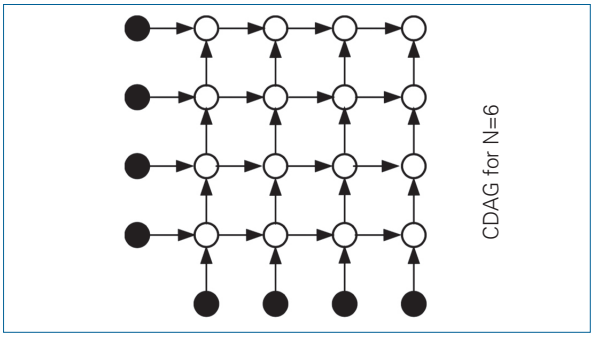

图 6　CDAG 图 (N=6)

对于图 4(b) 中的蓝色版本代码，该 CDAG 图首先利用 A[1][0] 和 A[0][1] 的初始值，得到 A[1][1]，

由 A[1][1] 和 A[0][2] 得到 A[1][2]，由 A[2][0] 和 A[1][1] 得到 A[2][1]，接下来利用 A[2][1] 和 A[1][2] 的值，得到 A[2][2]。对于图 4(a) 中的红色版本代码，会先后计算 A[1][0] 和 A[0][1]、A[1][1] 和 A[0][1]、A[1][2] 和 A[0][2]、A[1][3] 和 A[0][3]……以此类推。显而易见，这两个版本的代码有着不同的操作执行顺序。然而，这两种不同的执行顺序会影响对内存的读取效率。

```
for (i=0; i<N; i++)
  for (j=0;j<N;j++)
    for (k=0;k<N;k++)
      C[i][j] += A[i][k]*B[k][j];
```

图 7　矩阵乘法

如果我们能找出所有有效的执行顺序，那么就可以找到数据移动最少的那一个顺序。显然，找出所有有效的执行顺序是困难的，因此，确定最少的数据移动是难以实现的。即便如此，我们可以找出数据移动的下界。对于图 7 中展示的矩阵乘法算法，洪 (Hong) 和孔 (Kung) 等人指出对于任何有效的矩阵乘法的实现，若系统的缓存大小为 C，则主存和缓存之间需要 $\frac{N^3}{\sqrt{C}}$ 体积的数据移动。这一工作发表在 1981 年的 ACM 计算理论年会 (STOC) 上[1]。埃诺里 (Irony)[4] 等人 2004 年发表在《并行和分布式计算学报》(*JPDC*) 上的工作推算出了更低的下界 $\frac{1}{2\sqrt{2}}\frac{N^3}{\sqrt{C}} - C$。2008 年唐加拉 (Dongarra) 等人发表在 JFOCS 的工作得到的下界为 $1.83 * \frac{N^3}{\sqrt{C}} - C$。史密斯 (Smith)、范德盖恩 (Van de Geijn) 和朗古 (Langou)[5] 则认为下界是 $2 * \frac{N^3}{\sqrt{C}} - C$。研究者对下界提出了越来越紧的常量系数，对于任何高效循环分块优化，$2 * \frac{N^3}{\sqrt{C}} - C$ 都是这些算法数据移动量的下界。然而，与矩阵乘法相比，其他一些算法的数据移动下界还是未知的，这是一个还没有解决的问题。

图 8 总结了四种不同算法所需的浮点运算能力、数据移动量下界和操作密度上界。这三个指标之间存在着这样一种关系：用 FLOPs 除以 I/O 的下界，可得到操作密度的上界。如果我们设图 7 中的数据格式为 8 字节，那么数据移动量下界即为 $16 * \frac{N^3}{\sqrt{C}} - C$ 字节。可以看出，操作密度的上界是一个关于缓存大小的函数。

Algorithm	#Float-Ops	I/O Lower Bound	OI Upper Bound	
Matrix Multiplication	$2N^3$	$16*N^3/C^{1/2}$ bytes	$(1/8)*C^{1/2}$	☺
FFT	$2*N*\log_2 N$	$16*N*\log_2 N/\log_2 C$	$(1/8)*\log_2 C$	☹
Conj. Gradient (2D Heat Eqn.)	$20*N^2*T$	$48*N^2*T$	5/12	☹
Jacobi 2D	$9*N^2*T$	$48*N^2*T/C^{1/2}$	$(3/16)*C^{1/2}$	☺

图 8　不同算法的操作密度上界

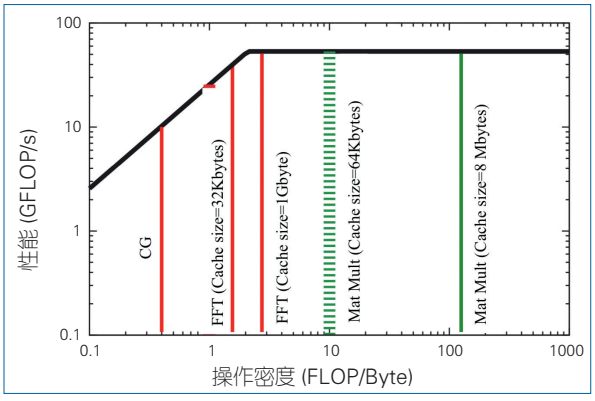

图 9　操作密度对性能的影响

对于不同算法，随着缓存的增加，算法性能的增幅表现不同。从图 9 可以看出矩阵乘法 (matrix multiplication) 和雅可比矩阵 (Jacobi 2D) 能够很快超越机器平衡点，得到更好性能。

数据移动的复杂度在表征和理解上比计算复杂度要困难得多，还有另一方面原因：计算复杂度具有可加性，而数据移动却没有这个性质。当计算是组合操作时，计算复杂度往往可由子操作的计算复杂度累加而得到。但是数据移动复杂度却没有这个性质，因为组合操作的数据移动可能少于子操作数据移动的累加。点乘操作就是典型的例子。

我们面临着优化计算和数据移动的挑战。对于前者，假设浮点运算是昂贵的且没有数据移动，那

```
Parameters: N, T
Inputs: In[N]; Outputs: A[N]
for (i=0; i<N; i++)
  A[i] = In[i];                    S1
for (t=0;t<T;t++) {
 for(i=1;i<N-1;i++)
   B[i] = A[i-1]+A[i]+A[i+1];  S2
 for(i=1;i<N-1;i++)
   A[i] = B[i]; }                  S3
```

图 10　代码样例

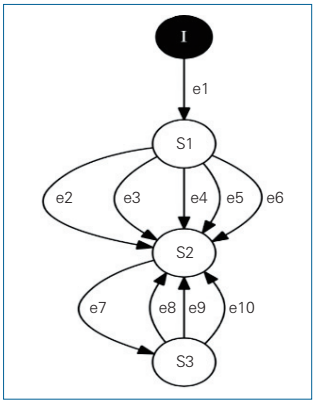

图 11　数据依赖图

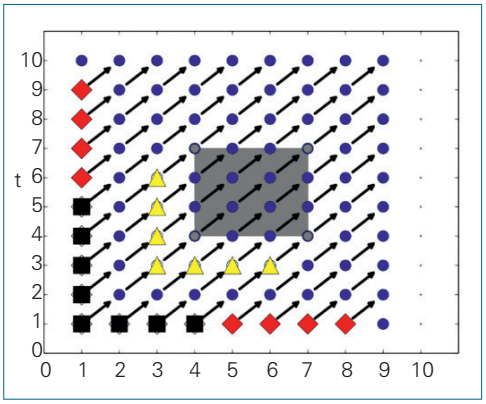

图 12　CDAG 图

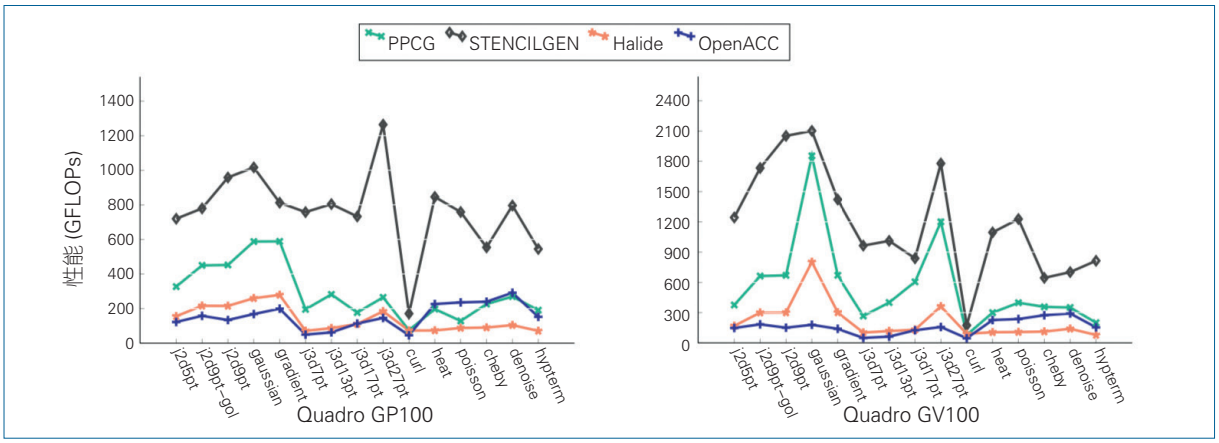

图 13　不同编译结果的性能对比

么优化就是简单的组合问题，因为计算复杂度具有可加性。一个组合操作的优化，可以看成是对多个子操作的优化，而显然，特定子操作的优化我们是可以完成的。对于多进程并行的运算，浮点运算的开销如果远大于数据移动开销，那么并行计算的主要问题就是处理器内核的负载均衡问题，如果子操作是负载均衡的，组合操作也会是负载均衡的。

但不幸的是，现实系统中浮点运算开销通常并不大，而数据移动却是昂贵的。正如上面提到的，数据移动不具有可加性，无法拆分为子问题的优化再进行组合。

埃兰戈 (Elango)、拉斯泰洛 (Rastello) 等人 2015 年在 POPL 上发表了一篇文章 [2]，其目的是在编译器中分析仿射循环程序 (affine loop programs) 数据移动量的下界。例如，对于图 10 中的代码，我们可以将其看作三个操作 S1、S2 和 S3 的组合。图 11 是这段代码的数据依赖图 (data dependence graph)，图 11 中的环对应图 12 中的 CDAG 射线。求解数据移动的下界有两个关键步骤。首先，通过对 Loomis-Whitney 几何不等式的一般化，得到多维空间中点的数量的上限，这个上限等于投影到低维空间里点的数量的乘积；其次，分析可执行操作（投影点）的最大数量和访问数据元素数量之间的关系。

多面体循环优化

邦杜古拉 (Bondhugula)、哈多诺 (Hartono) 等人 2008 年发表在 PLDI 上的工作 [3] 提出了多面体循环优化 (polyhedral loop optimization) 方法。该论文于 2018 年获得了 PLDI 最有影响力论文奖。我们知道，传

Benchmark	N	T	k	FPP	Benchmark	N	T	k	FPP	Benchmark	N	T	k	FPP
j2d5pt	8192^2	4	1	10	j3d7pt	512^3	4	1	13	heat	512^3	4	1	15
j2d9pt-gol	8192^2	4	1	18	j3d13pt	512^3	4	2	25	poisson	512^3	4	1	21
j2d9pt	8192^2	4	2	18	j3d17pt	512^3	4	1	28	cheby	512^3	4	1	39
gaussian	8192^2	4	2	50	j3d27pt	512^3	4	1	54	denoise	512^3	4	2	62
gradient	8192^2	4	1	18	curl	450^3	1	1	36	hypterm	300^3	1	4	358

N: Domain Size, T: Time Tile Size, k: Stencil Order, FPP: FLOPs per Point

图 14　测试基准

统的编译技术会为用户编写的程序生成抽象语法树 (AST)，这是一种中间表达 (IR)。与此不同的是，多面体编译器使用一种便于语义分析和翻译的中间表达。这种多面体循环优化对非完美嵌套循环有着统一、强大的抽象，对带有参数的循环边界有着统一和强大的处理能力。任何循环变换都能等价为仿射调度函数，使之前不可行的变换和等价的非完美嵌套循环能够执行。例如 cholesky 分解的六种置换变体。

那么，多面体循环优化有哪些限制呢？多面体循环优化受限于仿射计算。对于图 14 的测试基准，可以从图 13 的结果中看出，对于稀疏矩阵和稀疏张量的计算，多面体循环优化的性能远远低于人工优化的版本，也低于领域专用语言 (DSL) 生成的代码的性能。

影响多面体循环优化的原因是什么呢？ (1) 在多面体编译器中使用的性能模型是线性代价函数，对于指导循环变换的能力有限；(2) 解耦合循环变换和分块大小选择；(3) 从巨大的选择空间中选择最优的解决方案十分困难，例如主循环的变换就有置换 (permutation)、分块 (tiling)、融合 (fusion) 等多种方法，而数据放置的变换也有多种方案。

特定领域优化

为什么一个特定领域 (domain-specific) 的优化会比其他编译优化性能更高呢？我们将以张量收缩为例进行说明，该计算方法是矩阵乘法的高维模拟。如图 15 所示，假设我们有 2 个 4D 的矩阵 A 和 B，二者相乘得到矩阵 C。请注意 A 和 B 的下标共有 6 个变量，因此简单的实现方法是利用 6 层循环。但

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
   for (l=0; l<N; l++)
    for (m=0; m<N; m++)
     for (n=0; n<N; n++)
      C[i][j][k][l] += A[i][m][k][n]*B[j][n][l][m];
```

$$C_{ijkl} = \sum_{mn} A_{imkn} \cdot B_{jnlm}$$

图 15　6 层循环代码样例

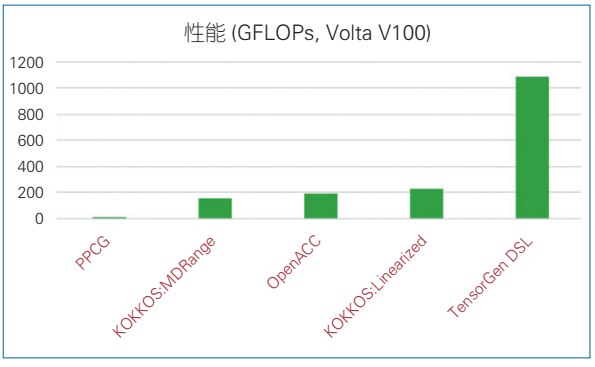

图 16　程序在 GPU 上的性能

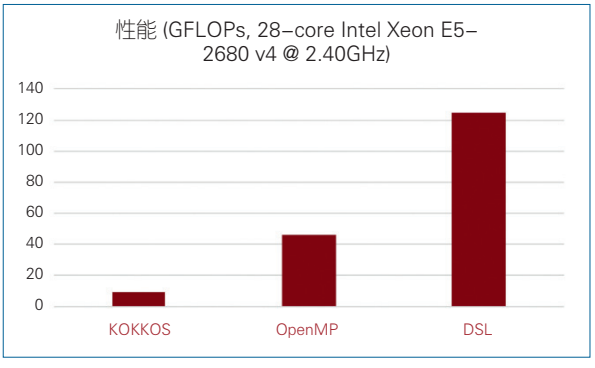

图 17　程序在 CPU 上的性能

是这样的实现性能较差。

循环置换 / 分块的选择，再加上分块大小的选择，令这个需要搜索的选择空间十分巨大，要从中得出最优的方案是非常具有挑战性的非线性优化问

题。对于通用型他编译器来说，搜索最优的方案是十分耗时的，但是对于特定领域的编译器，只需指出张量定义中的属性，DSL 编译器就能利用其完成张量收缩的优化。

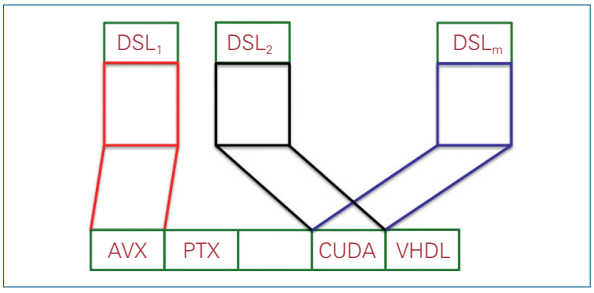

图 18　DSL 编译

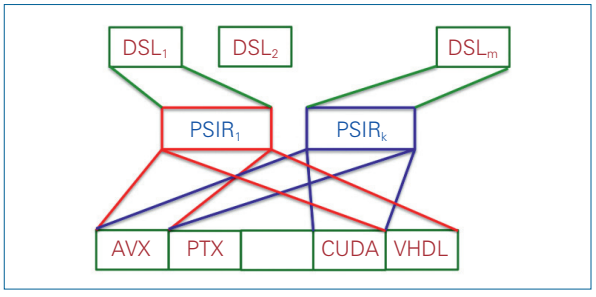

图 19　为 DSL 增加中间表达

我们使用 PPCG、KOKKOS、OpenACC、TensorGen DSL、OpenMP 等不同编译器编译 CCSD(T) 的代码，将得到的可执行程序在 CPU 和 GPU 上分别运行。CCSD(T) 是一种用于量子化学和分子动力学领域的计算密集型应用。我们可以发现，TensorGen DSL 编译器编译得到的程序性能远超其他通用型编译器（见图 16、17）。那么，特定的优化器能否被采用到通用型的优化编译器内呢？

多目标的 DSL 拥有高性能、可移植性和高生产率等特性。特定领域语言的中间表达可以使计算或数据的映射和调度更容易（见图 18、19）。并且多目标的 DSL 将上层的目标独立决策和下层的特定平台选择分离。特定平台的代码模式由主要性能因子驱动。但是现在的每一个 DSL（无论是编译器还是库）都是一个独立的系统，很少复用，相同功能重复实现。归根结底都是因为目前还没有一个通用的基础架构来支持开发 DSLs。

如果有基于模式的中间层存在会怎么样呢？如何为不同目标生成高效代码？如何设计出可复用的“负载均衡”“高效重叠计算与传输”“最小化数据移动”等编译器组件呢？是否有少量的具有广泛覆盖的中间表达呢？这些都是我们值得思考和需要解决的难题。

编译优化中的开放问题

我们罗列了一些编译器优化方面有趣的开放问题，如果能够得到解决，将会使我们在有限的资源上挖掘出更多的空间和价值。(1) 以数据结构为中心的访问模式与其他的数据访问模式存在根本的不同。虽然数量很少，却意义重大。因此，能否在关键算法中以较大的覆盖度来识别少量的以数据结构为中心的访问模式成为了第一个开放问题。(2) 特定模式的优化是否能无缝融入一个更通用的编译器框架？我们希望这些特定模式的优化能应用到更通用的编译器框架内，这样编译器就能在遇到实例时对其进行优化。(3) 广泛的标准中间表达 (IRs) 和接口 (APIs) 能否被确定，以便编译器框架即插即用？这是另一个重要的要求，因为只有这样才能通过广大的社区力量来实现更好的进步。目前，围绕着 LLVM 这一大型项目的社区为可互操作框架作出贡献。(4) 强大的证明器（如 SAT、SMT）能否被用于连接基于 ILP 的优化器，从而克服现有的多维编译器的限制？(5) 神奇的机器学习方法被广泛应用于解决很多计算机领域或非计算机领域的难题。那么，机器学习方法能否被高效地利用到性能模型和编译器优化？我们认为，很多性能建模、模型分析都无法拟合实际情况。因此，黑盒的机器学习模型可能也无法完成。

总结

预测超大规模集成电路发展的摩尔定律的失效，意味着我们需要对硬件资源进行高度的定制化和更明确的控制，以实现对有限资源更加高效的利用。这些需求和变化为开发高性能软件带来极大的挑战。然而，编译器及编译技术应该发挥更重要的作用，以达到更

高的性能、生产率和可移植性。但是，编译器面临着一个巨大的挑战——数据运输的优化。虽然围绕这个问题的研究开展得如火如荼，但仍未得到根本的解决。宏大的命题令人胆怯，但是针对数据结构或者特定模式的优化是一个很有希望的方向。范式重用 (canonical reuse) 模式覆盖了大部分机器学习算法。高效的算法和体系结构协同设计存在着大量的空间，两个层次迭代设计空间探索让编译器变得十分重要：对外，是指体系结构参数；对内，是指优化体系结构的映射和调度。轩尼诗 (Hennessey) 和帕特森 (Patterson) 在 2018 年 ACM 图灵奖颁奖典礼的报告上提到，随着摩尔定律的结束，我们进入了一个计算机体系结构的黄金时代。对编译器研究来说，这也将是令人兴奋和有影响力的一段时间！ ■

作者：

P. (Saday) Sadayappan

犹他大学计算机科学与工程系教授，IEEE Fellow。主要研究方向为高性能计算的性能优化和编译器 / 运行系统。

整理者：

李　诚

CCF 专业会员，CCF 体系结构、系统软件专委通信委员。中国科学技术大学特任研究员。主要研究方向为大规模、实时、高可靠分布式系统。

chengli7@ustc.edu.cn

许冠斌

CCF 学生会员。中国科学技术大学硕士生。主要研究方向为分布式机器学习系统的可扩展性、容错性。

xugb@mail.ustc.edu.cn

参考文献

[1] Hong J W , Kung H T . I/O Complexity: The Red-Blue Pebble Game[C]// Proceedings of the 13th Annual ACM Symposium on Theory of Computing. ACM, 1981.

[2] Elango V , Rastello F , Pouchet L N , et al. On Characterizing the Data Access Complexity of Programs[J]. ACM SIGPLAN Notices, 2015, 50(1):567-580.

[3] Bondhugula U, Hartono A, Ramanujam J, et al. A practical automatic polyhedral parallelizer and locality optimizer[C]// PLDI 2008.

[4]Dror Ironya, Sivan Toledoa, Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication[J]//JPDC 2004

[5] Smith T M, Lowery B, Langou J, et al. A Tight I/O Lower Bound for Matrix Multiplication[J]. 2017