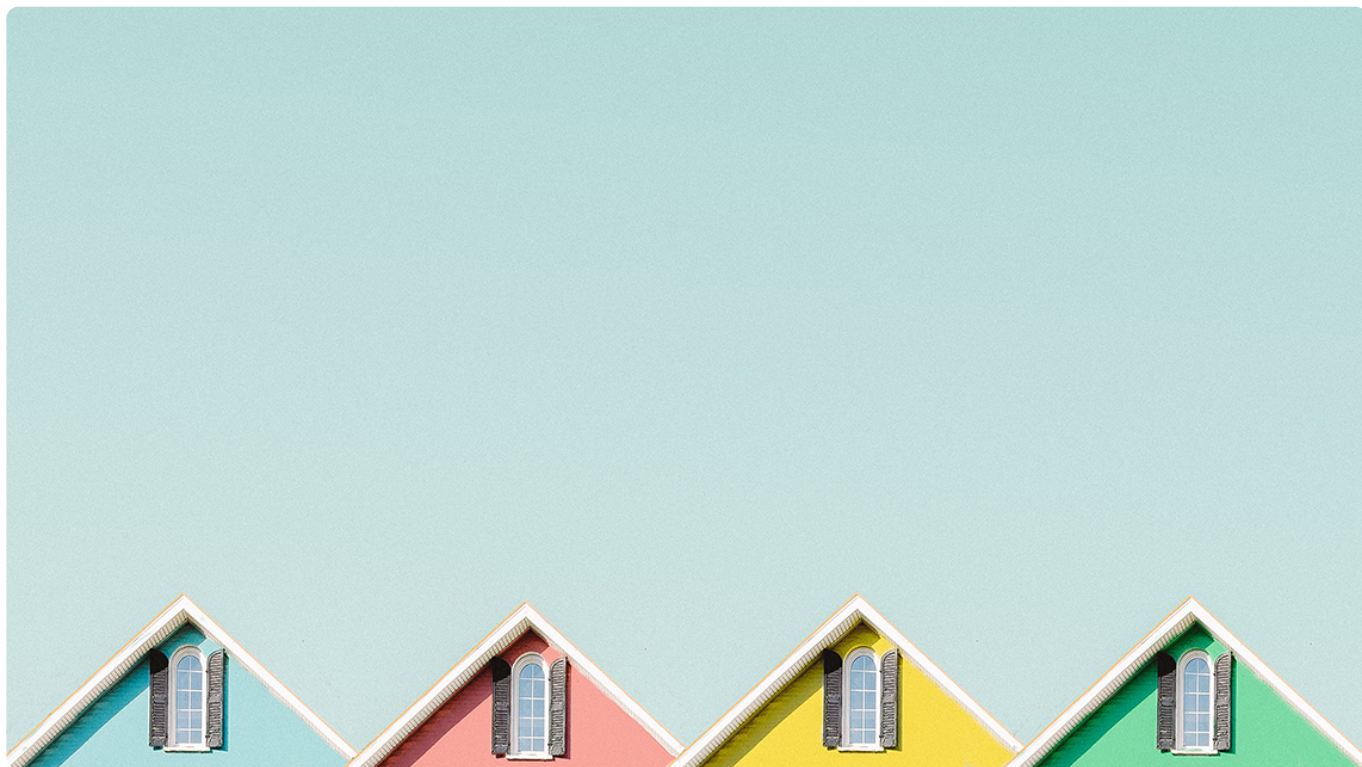


31 | 内存计算：对海量数据做计算，到底可以有多快？

2019-11-04 宫文学

编译原理之美

[进入课程 >](#)



讲述：宫文学

时长 17:46 大小 16.29M



内存计算是近十几年来，在数据库和大数据领域的一个热点。随着内存越来越便宜，CPU 的架构越来越先进，整个数据库都可以放在内存中，并通过 SIMD 和并行计算技术，来提升数据处理的性能。

我问你一个问题：做 1.6 亿条数据的汇总计算，需要花费多少时间呢？几秒？几十秒？还是几分钟？如果你经常使用数据库，肯定会知道，我们不会在数据库的一张表中保存上亿条的数据，因为处理速度会很慢。

但今天，我会带你采用内存计算技术，提高海量数据处理工作的性能。与此同时，我还会介绍 SIMD 指令、高速缓存和局部性、动态优化等知识点。这些知识点与编译器后端技术息息相关，掌握这些内容，会对你从事基础软件研发工作，有很大的帮助。

了解 SIMD

本节课所采用的 CPU，支持一类叫做 SIMD (Single Instruction Multiple Data) 的指令，**它的字面意思是**：单条指令能处理多个数据。相应的，你可以把每次只处理一个数据的指令，叫做 SISD (Single Instruction Single Data) 。

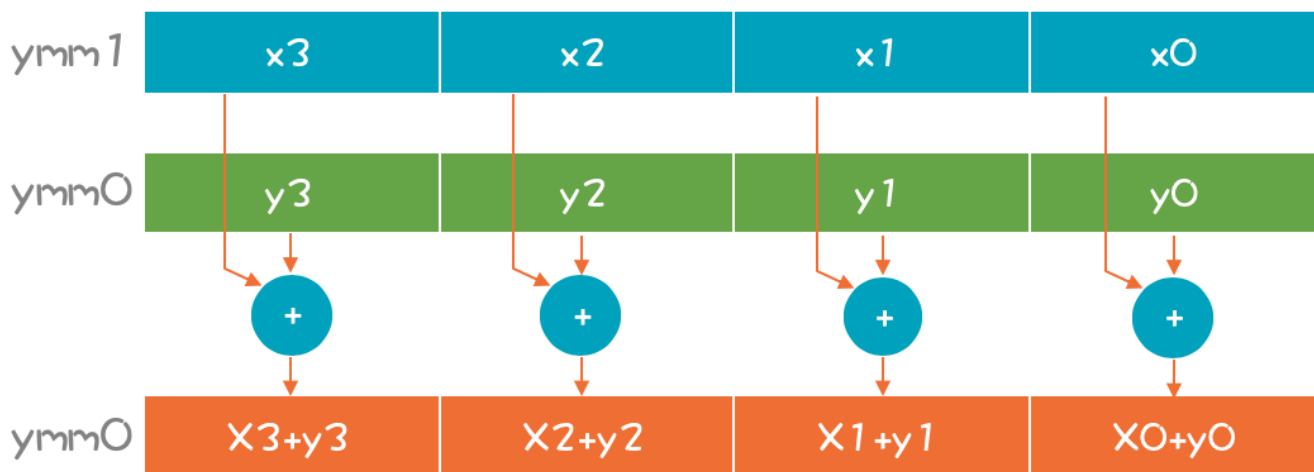
SISD 使用普通的寄存器进行操作，比如加法：

```
1 addl $10, %eax
```

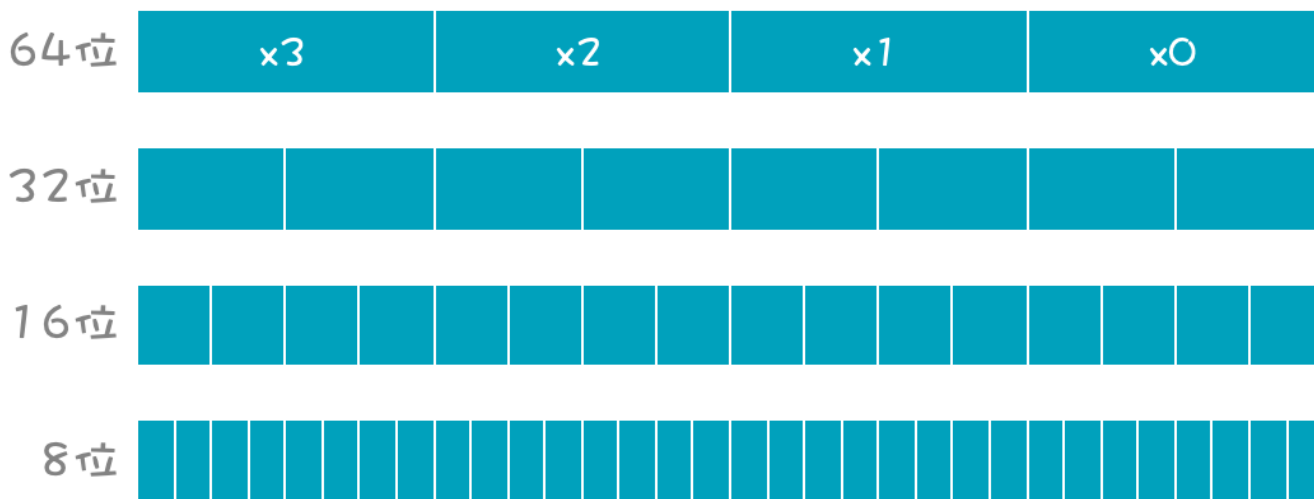
[复制代码](#)

这行代码是把一个 32 位的整型数字，加到 %eax 寄存器上（在 x86-64 架构下，这个寄存器一共有 64 位，但这个指令只用它的低 32 位，高 32 位是闲置的）。

这种一次只处理一个数据的计算，**叫做标量计算**；一次可以同时处理多个数据的计算，**叫做矢量计算**。它在一个寄存器里可以并排摆下 4 个、8 个甚至更多标量，构成一个矢量。图中 ymm 寄存器是 256 位的，可以支持同时做 4 个 64 位数的计算（xmm 寄存器是它的低 128 位）。



如果不做 64 位整数，而做 32 位整数计算，一次能计算 8 个，如果做单字节（8 位）数字的计算，一次可以算 32 个！



1997 年，Intel 公司推出了奔腾处理器，带有 MMX 指令集，意思是多媒体扩展。当时，让计算机能够播放多媒体（比如播放视频），是一个巨大的进步。但播放视频需要大量的浮点计算，依靠原来 CPU 的浮点运算功能并不够。

所以，Intel 公司就引入了 MMX 指令集，和容量更大的寄存器来支持一条指令，同时计算多个数据，这是在 PC 上最早的 SIMD 指令集。后来，SIMD 又继续发展，陆续产生了 SSE（流式 SIMD 扩展）、AVX（高级矢量扩展）指令集，处理能力越来越强大。

2017 年，Intel 公司发布了一款至强处理器，支持 AVX-512 指令（也就是它的一个寄存器有 512 位）。每次能处理 8 个 64 位整数，或 16 个 32 位整数，或者 32 个双精度数、64 个单精度数。你想想，一条指令顶 64 条指令，几十倍的性能提升，是不是很厉害！

那么你的电脑是否支持 SIMD 指令？又支持哪些指令集呢？在命令行终端，打下面的命令，你可以查看 CPU 所支持的指令集。

```
1 sysctl -a | grep features | grep cpu    //macOs 操作系统
2 cat /proc/cpuinfo                      //Linux 操作系统
```

复制代码

现在，想必你已经知道了 SIMD 指令的强大之处了。而它的实际作用主要有以下几点：

SIMD 有助于多媒体的处理，比如在电脑上流畅地播放视频，或者开视频会议；

在游戏领域，图形渲染主要靠 GPU，但如果你没有强大的 GPU，还是要靠 CPU 的 SIMD 指令来帮忙；

在商业领域，数据库系统会采用 SIMD 来快速处理海量的数据；

人工智能领域，机器学习需要消耗大量的计算量，SIMD 指令可以提升机器学习的速度。

你平常写的程序，编译器也会优化成，尽量使用 SIMD 指令来提高性能。

所以，我们所用到的程序，其实天天在都在执行 SIMD 指令。

接下来，我来演示一下如何使用 SIMD 指令，与传统的数据处理技术做性能上的对比，并探讨如何在编译器中生成 SIMD 指令，这样你可以针对自己的项目充分发挥 SIMD 指令的优势。

Intel 公司为 SIMD 指令提供了一个标准的库，可以生成 SIMD 的汇编指令。我们写一个简单的程序（参考 [simd1.c](#)）来对两组数据做加法运算，每组 8 个整数：

 复制代码

```
1 #include <stdio.h>
2 #include "immintrin.h"
3
4 void sum(){
5     // 初始化两个矢量，8 个 32 位整数
6     __m256i a=_mm256_set_epi32(20,30,40,60,342,34523,474,123);
7     __m256i b=_mm256_set_epi32(234,234,456,78,2345,213,76,88);
8
9     // 矢量加法
10    __m256i sum=_mm256_add_epi32(a, b);
11
12    // 打印每个值
13    int32_t* s = (int32_t*)&sum;
14    for (int i = 0; i < 8; i++){
15        printf("s[%d] : %d\n", i, s[i]);
16    }
17 }
```


把矢量加法运算翻译成汇编语言的话，采用的指令是 vpaddd（其中的 p 是 pack 的意思，对一组数据操作）。寄存器的名字是 ymm（y 开头意思是 256 位的）。

 复制代码

```
1 vpaddd %ymm0, %ymm1, %ymm0
```

在这个示例中，我们构建了两个矢量数据，这个计算很简单。**接下来，我们挑战一个有难度的题目：把 1.6 亿个 64 位的整数做加法！**

1.6 亿个 64 位整数要占据大约 1.2G 的内存，你要把这 1.2G 的数据全部汇总一遍！要实现这个功能，你首先要申请一块 1.2G 大小的内存，并且要是 32 位对齐的（因为后面加载数据到寄存器的指令需要内存对齐，这样加载速度更快）。

 复制代码

```
1 unsigned totalNums = 1600000000;
2 // 申请一块 32 位对齐的内存。
3 // 注意: aligned_alloc 函数 C11 标准才支持
4 int64_t * nums = aligned_alloc(32, totalNums * sizeof(int64_t));
5
6 // 初始化 sum 值
7 __m256i sum=_mm256_setzero_si256();
8
9 __m256i * vectorptr = (__m256i *) nums;
10 for (int i = 0; i < totalNums/4; i++) {
11     // 从内存加载 256 位进来
12     __m256i a = _mm256_load_si256(vectorptr+i);
13     // 矢量加法
14     sum=_mm256_add_epi64(sum,a);
15 }
```

完整的代码见 [🔗 simd2.c](#)。

最后，要用下面的命令，编译成可执行文件（-mavx2 参数是告诉编译器，要使用 CPU 的 AVX2 特性）：

 复制代码

```
1 gcc -mavx2 simd2.c -o simd2
2 或
3 clang -mavx2 simd2.c -o simd2
```

你可以运行一下，看看用了多少时间。

我的 MacBook Pro 大约用了 0.15 秒。**注意**，这还是只用了一个内核做计算的情况。我提供的 simd3.c 示例程序，是计算 1.6 亿个双精度浮点数，所用的时间也差不多，都是亚秒级。而计算速度之所以这么快，**主要有两个原因：**

采用了 SIMD;

高速缓存和数据局部性所带来的帮助。

我们先把 SIMD 讨论完，然后再讨论高速缓存和数据局部性。

矢量化功能可以一个指令当好几个用，但刚才编写的 SIMD 示例代码使用了特别的库，这些库函数本身就是用嵌入式的汇编指令写的，所以，相当于我们直接使用了 SIMD 的指令。

如果我们不调用这几个库，直接做加减乘除运算，能否获得 SIMD 的好处呢？也可以。不过要靠编译器的帮助，所以，接下来来看看 LLVM 是怎样帮我们使用 SIMD 指令的。

LLVM 的自动矢量化功能 (Auto-Vectorization)

各个编译器都在自动矢量化功能上下了功夫，以 LLVM 为例，它支持循环的矢量化 (Loop Vectorizer) 和 SLP 矢量化功能。

循环的矢量化很容易理解。如果我们处理一个很大的数组，肯定是顺序读取内存的，就如 [loop1\(\)](#) 函数的代码：

```
1 int loop1(int totalNums, int * nums){
2     int sum = 0;
3     for (int i = 0; i < totalNums; i++){
4         sum += nums[i];
5     }
6     return sum;
7 }
```

 复制代码

不过，如果你用不同的参数去生成汇编代码，**结果会不一样：**

```
clang -S loop.c -o loop-scalar.s
```

这是最常规的汇编代码，老老实实在地用 add 指令和 %eax 寄存器做加法。

```
clang -S -O2 loop.c -o loop-O2.s
```

它在使用 `paddb` 指令和 `xmm` 寄存器，这已经在使用 SIMD 指令了。

```
clang -S -O2 -fno-vectorize loop.c -o loop-O2-scalar.s
```

这次带上了 `-O2` 参数，要求编译器做优化，但又带上了 `-fno-vectorize` 参数，要求编译器不要通过矢量化做优化。那么生成的代码会是这个样子：

 复制代码

```
1  addl    (%rsi,%rdx,4), %eax
2  addl    4(%rsi,%rdx,4), %eax
3  addl    8(%rsi,%rdx,4), %eax
4  addl    12(%rsi,%rdx,4), %eax
5  addl    16(%rsi,%rdx,4), %eax
6  addl    20(%rsi,%rdx,4), %eax
7  addl    24(%rsi,%rdx,4), %eax
8  addl    28(%rsi,%rdx,4), %eax
```

也就是它一次循环就做了 8 次加法计算，减少了循环的次数，也更容易利用高速缓存，来提高数据读入的效率，所以会导致性能上的优化。

```
clang -S -O2 -mavx2 loop.c -o loop-avx2.s
```

这次带上 `-mavx2` 参数，编译器就会使用 AVX2 指令来做矢量化，你查看代码会看到对 `vpaddd` 指令和 `ymm` 寄存器的使用。

其实，在 `simd2.c` 中，我们有 [一段循环语句](#)，对标量数字进行加总。这段代码在缺省的情况下，也会被编译器矢量化（你可以看看汇编代码 [simd2-O2-avx2.s](#) 确认一下）。

在做自动矢量的时候，编译器要避免一些潜在的问题，看看 [loop2\(\)](#) 函数的代码：


 复制代码

```
1  void loop2(int totalNums, int * nums1, int * nums2){
2      for (int i = 0; i < totalNums; i++){
3          nums2[i] += nums1[i];
4      }
5  }
```


代码中的 `nums1` 和 `nums2` 是两个指针，指向内存中的两个整数数组的位置。但从代码里看不出 `nums1` 和 `nums2` 是否有重叠，一旦它们有重叠的话，矢量化计算结果会出错。

所以，编译程序会生成矢量和标量两个版本的目标代码，在运行时检测 `nums1` 和 `nums2` 是否重叠，从而判断是否跳转到矢量化计算代码。从这里你也可以看出：写编译器真的要有工匠精神，要把各种可能性都想到。

实际上，在编译器里有很多这样的实现。你可以将循环次数改为一个常量，看一下 [loop3\(\)](#) 函数，它所生成的汇编代码会根据常量的值做优化，甚至完全不做循环：

 复制代码

```
1 int loop3(int * nums){
2     int sum = 0;
3     for (int i = 0; i < 160; i++){
4         sum += nums[i];
5     }
6     return sum;
7 }
```

除了循环的矢量化器，LLVM 还有一个 SLP 矢量化器，它能在做全局优化时，寻找可被矢量化代码来做转换。比如下面的代码，对 `A[0]` 和 `A[1]` 的操作非常相似，可以考虑按照矢量的方式来计算：

 复制代码

```
1 void foo(int a1, int a2, int b1, int b2, int *A) {
2     A[0] = a1*(a1 + b1)/b1 + 50*b1/a1;
3     A[1] = a2*(a2 + b2)/b2 + 50*b2/a2;
4 }
```

所以，LLVM 确实在自动矢量化方面做了大量工作。**在你设计一个新的编译器的时候，可以充分利用这些已有的成果。**否则，在每个优化算法上，你都需要投入大量的精力，还不一定能做得足够稳定。

到目前为止，我们针对 SIMD 和矢量化谈得足够多了。2011 年左右，我第一次做内存计算方面的编程时，被如此快的处理速度吓了一跳。因为如果你经常操作数据库，肯定会知道从

数据库里做 1.6 亿个数据的汇总是什么概念。

一般来说，一张表有上亿条数据之前，我们就已经要做分拆了。大多数情况下，表中的数据要比 1.6 亿低一个数量级，就算是这样，你对一个有着一两千万行数据表做统计，仍然要花费不少的时间。

而毫不费力地进行海量数据的计算，就是内存计算的魅力。当然了，这里面有高速缓存和局部性的帮助。所以，我们继续讨论一下，跟内存计算有关的第二个问题：高速缓存和局部性。

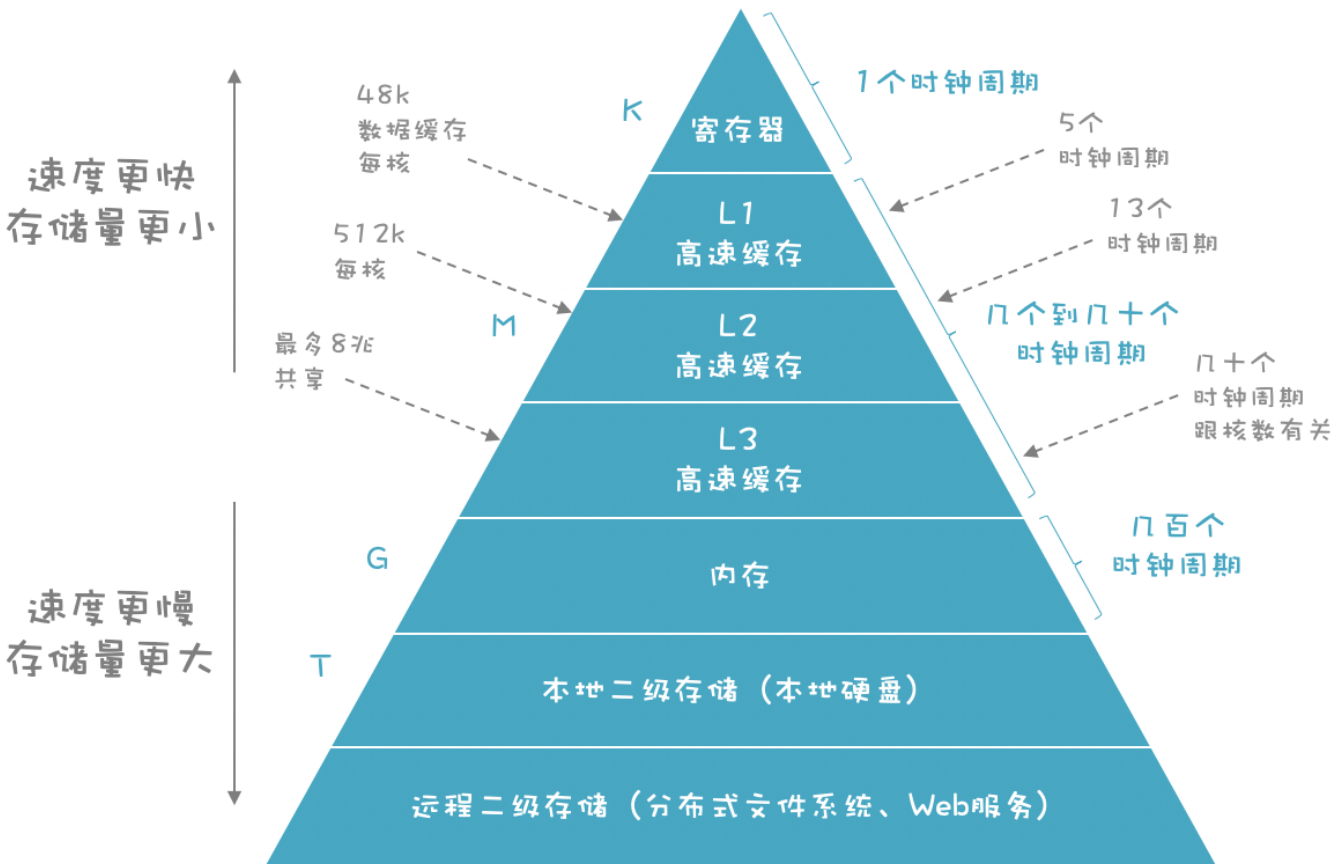
高速缓存和局部性

我们知道，计算机的存储是分成多个级别的：

速度最快的是寄存器，通常在寄存器之间复制数据只需要 1 个时钟周期。

其次是高速缓存，它根据速度和容量分为多个层级，读取所花费的时间从几个时钟周期到几十个时钟周期不等。

内存则要用上百到几百个时钟周期。



在图中的存储层次结构中，越往下，存取速度越慢，但是却可以有更大的容量，从寄存器的 K 级，到高速缓存的 M 级，到内存的 G 级，到磁盘的 T 级（灰色标的数据是 Intel 公司的 Ice Lake 架构的 CPU 的数据）。

一般的计算机指令 1 到几个时钟周期就可以执行完毕。所以，如果等待内存中读取，获得数据的话，CPU 的性能可能只能发挥出 1%。不过由于高速缓存的存在，读取数据的平均时间会缩短到几个时钟周期，这样 CPU 的能力可以充分发挥出来。所以，我在讲程序的运行时环境的时候，让你关注 CPU 上两个重要的部件：**一个是寄存器，另一个就是高速缓存。**

在代码里，我们会用到寄存器，并且还会用专门的寄存器分配的算法来优化寄存器。可是对于高速缓存，我们没有办法直接控制。

因为当你用 mov 指令从内存中，加载数据到寄存器时，或者用 add 指令把内存中的一个数据，加到寄存器中，一个已有的值上面时，CPU 会自动控制是从内存里取，还是在高速缓存中取，并控制高速缓存的刷新。

那我们有什么办法呢？答案是**提高程序的局部性 (locality)**，这个局部性又分为两个：

一是时间局部性。一个数据一旦被加载到高速缓存甚至寄存器，我们后序的代码都能集中访问这个数据，别等着这个数据失效了再访问，那就又需要从低级别的存储中加载一次。

第二个是空间局部性。当我们访问了一条数据之后，很可能马上访问跟这个数据挨着的其他数据。CPU 在一次读入数据的时候，会把相邻的数据都加载到高速缓存，这样会增加后面代码在高速缓存中命中的概率。

提高局部性这件事情，更多的是程序员的责任，编译器能做的事情不多。不过，有一种编译优化技术，叫做**循环互换优化 (loop interchange optimization)**可以让程序更好地利用高速缓存和寄存器。

下面的例子中有内循环和外循环，内循环次数较少，外循环次数很大。如果内循环里的临时变量比较多，需要占用寄存器和高速缓存，那么 i 就可能被挤掉，等下一次用到 i 的时候，需要重新从低一级的存储中获取，从而造成性能的降低：

```
1 for(i=0; i<10000000; i++)
2   for(j=0; j<10; j++){
3     a[i] *= b[i]
4     ...
5   }
```

编译器可以把内外层循环交换，这样就提高了局部性：

```
1 for(j=0; j<10; j++)
2   for(i= 0; i<10000000; i++){
3     a[i] *= b[i]
4     ...
5   }
```

不过，在大多数情况下，*i* 和 *j* 循环的次数不是一个常量，而是一个变量，在编译时不知道内层循环次数更多还是外层循环。这样的话，可能就需要生成两套代码，在运行时根据情况决定跳转到哪个代码块去执行，**这样会导致目标代码的膨胀。**

如果不想让代码膨胀，又能获得优化的目标代码，你可以尝试在运行时做动态的优化（也就是动态编译），这也是 LLVM 的设计目标之一。因为在静态编译期，我们确实没办法知道运行时的信息，从而也没有办法生成最优化的目标代码。

作为一名优秀的程序员，你有责任让程序保持更好的局部性。比如，假设你要设计一个内存数据库，并且经常做汇总计算，那么你会把每个字段的数据按行存储在一起，还是按列存储？当然是后者，因为这样才具备更好的数据局部性。

最后，除了 SIMD 和数据局部性，促成内存计算这个领域发展的还有两个因素：

多内核并行计算。现在的 CPU 内核越来越多，特别是用于服务器的 CPU。多路 CPU 几十上百个内核，能够让单机处理能力再次提升几十，甚至上百倍。

内存越来越便宜。在服务器上配置几十个 G 的内存已经是常规配置，配置上 T 的内存，也不罕见。这使得大量与数据处理有关的工作，可以基于内存，而不是磁盘。除了要更新数据，几乎可以不访问相对速度很低的磁盘。

在这些因素的共同作用下，内存计算的使用越来越普遍。在你的项目里，你可以考虑采用这个技术，来加速海量数据的处理。

课程小结

本节课，我带你了解了内存计算的特点，以及与编译技术的关系，我希望你能记住几点：

SIMD 是一种指令级并行技术，它能够矢量化地一次计算多条数据，从而提升计算性能，在计算密集型的需求中，比如多媒体处理、海量数据处理、人工智能、游戏等领域，你可以考虑充分利用 SIMD 技术。

充分保持程序的局部性，能够更好地利用计算机的高速缓存，从而提高程序的性能。

SIMD，加上数据局部性，和多个 CPU 内核的并行处理能力，再加上低价的海量的内存，推动了内存计算技术的普及，它能够同时满足计算密集，和海量数据的需求。

有时候，我们必须在运行期，根据一些数据来做优化，生成更优的目标代码，在编译期不可能做到尽善尽美。

我想强调的是，熟悉编译器的后端技术将会有利于你参与基础平台的研发。如果你想设计一款内存数据库产品，一款大数据产品，或者其他产品，将计算机的底层架构知识，和编译技术结合起来，会让你有机会发挥更大的作用！

一课一思

你是否在自己的领域里使用过内存计算技术？它能带来什么好处？欢迎分享你的观点。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

示例代码我放在文末，供你参考。

lab/31-simd (示例代码目录) [码云](#) [GitHub](#)

simd1.c (两个矢量常数相加) [码云](#) [GitHub](#)

simd2.c (1.6 亿个 32 位整数汇总) [码云](#) [GitHub](#)

simd3.c (1.6 亿个双精度浮点数汇总) [码云](#) [GitHub](#)

loop.c (测试对循环的自动矢量化) [码云](#) [GitHub](#)



编译原理之美

手把手教你实现一个编译器

宫文学

北京物演科技CEO



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 目标代码的生成和优化（二）：如何适应各种硬件架构？

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。