Datawhale

Datawhale 开源社区

Datawhale open source community

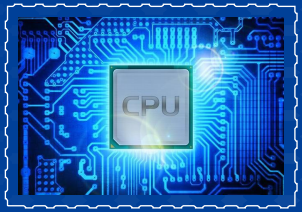
深入理解计算机系统（11）

Computer Systems A Programmer's Perspective CSAPP

李岳昆、易远哲

realyurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 12 月 18 日



第 I 部分

优化程序性能-I

优化编译器的能力和局限性

Datawhale

我们先看看下面的代码：

struct

```
1 void twiddle1(long *xp, long *yp) {
2     *xp += *yp;
3     *xp += *yp;
4 }
5
6 void twiddle2(long *xp, long *yp) {
7     *xp += 2 * *yp;
8 }
```

优化编译器的能力和局限性

Datawhale

在上一页的两段代码中，函数 twiddle2 的效率更高，因为它只要求 3 次内存引用（读 *xp，读 *yp，写 *xp），而 twiddle1 需要 6 次（2 次读 *xp，2 次读 *yp，2 次写 *xp）。

不过，如果 xp = yp，那么函数 twiddle1 实际的操作是将 xp 的值增加 4 倍。而函数 twiddle2 则是将 xp 的值增加了 3 倍。

由于编译器不知道 xp 与 yp 是否可能相等，因此 twiddle2 不能作为 twiddle1 的优化版本。

函数调用对优化的影响

Datawhale

考虑下面的代码：

struct

```
1 long f();
2
3 long func1() {
4     return f() + f() + f() + f();
5 }
6
7 long func2() {
8     return 4 * f();
9 }
```

先不考虑函数 f 的具体内容，可以看到，func2 只调用 f 一次，而 func1 调用 f 四次。但如果考虑函数 f 如下：

struct

```
long counter = 0;

long f() {
    return counter++;
}
```

显然，对于这样的 f，func1 会返回 6，而 func2 会返回 0。这种情况编译器也是无法判断的，因此编译器也无法做出这种优化。

程序性能的表示

Datawhale

对于一个程序，如果我们记录该程序的数据规模以及对应的运行所需的时钟周期，并通过最小二乘法来拟合这些点，我们将得到形如 $y = a + bx$ 的表达式，其中 y 是时钟周期，x 是数据规模。当数据规模较大时，运行时间就主要由线性因子 b 来决定。这时候，我们将 b 作为度量程序性能的标准，称为**每元素的周期数（Cycles Per Element, CPE）**。

代码移动

Datawhale

为了方便说明，先声明一个如下的结构：

typedef

```
typedef struct {
    long len;
    data_t *data;
} vec_rec, *vec_ptr
```

这个声明用 data_t 来表示基本元素的数据类型。

代码移动

Datawhale

先考虑如下的代码：

combine1

```
1  void combine1(vec_ptr v, data_t *dest) {
2      long i;
3
4      *dest = IDENT;
5      for (i = 0; i < vec_length(v); i++) {
6          data_t val;
7          get_vec_element(v, i, &val);
8          *dest = *dest OP val;
9      }
10 }
```

传统 DRAM

上一页的代码中，循环体每执行一次，都会调用一次函数 vec_length，但数组的长度是不变的，那么可以考虑将 vec_length 移出循环体来提升效率：

combine2

```
void combine2(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

减少过程调用

Datawhale

combine2

```
data_t *get_vec_start(vec_ptr v) {
    return v -> data;
}

void combine3(vec_ptr v, data_t *dest) {
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++)
        *dest = *dest OP data[i];
}
```

减少过程调用

Datawhale

在上一页的代码中，我们消除了循环体中的所有调用。但实际上，这样的改变不会带来性能的提升，在整数求和的情况下还会造成性能下降。这是因为内循环中还有其他的操作形成了瓶颈。

消除不必要的内存引用

Datawhale

先看看 combine3 的 x86-64 汇编代码:

assembly

```
.L17:
  vmovsd (%rbx), %xmm0
  vmulsd (%rdx), %xmm0, %xmm0
  vmovsd %xmm0, (%rbx)
  addq   $8, %rdx
  cmpq   %rax, %rdx
  jne    .L17
```

消除不必要的内存引用

Datawhale

通过上面的汇编代码可以看到，每次迭代时，累积变量的数值都要从内存中读出再写入到内存，这样的读写是很浪费的，而且是可以消除的：

combine4

```
1  void combine4(vec_ptr v, data_t *dest) {
2      long i;
3      long length = vec_length(v);
4      data_t *data = get_vec_start(v);
5      data_t acc = IDENT;
6
7      for (i = 0; i < length; i++)
8          acc = acc OP data[i];
9      *dest = acc;
10 }
```

现代处理器的优化

近期的 Intel 处理器是**超标量（superscalar）**的，意思是它可以在每个时钟周期执行多个操作。此外还是 **乱序的（out-of-order）**，意思是指令执行的顺序不一定与机器级程序中的顺序一致。

这样的设计使得处理器能够达到更高的并行度。例如，在执行分支结构的程序时，处理器会采用**分支预测（branch prediction）**技术，来预测是否需要选择分支，同时还预测分支的目标地址。

此外还有一种**投机执行（speculative execution）**技术，意思是处理器会在分支之前就执行分支之后的操作。如果预测错误，那么处理器就会将状态重置到分支点的状态。

循环展开

Datawhale

所谓循环展开，指的是通过增加每次迭代计算的元素数量来减少循环的迭代次数。考虑如下的程序：

psum1

```
1 void psum1(float a[], float p[], long n) {
2     long i;
3     p[0] = a[0];
4     for (i = 1; i < n; i++)
5         p[i] = p[i-1] + a[i];
6 }
```

下一页中将展示循环展开后的函数。

循环展开

通过对 psum1 函数进行循环展开，能够使迭代次数减半：

psum2

```
void psum2(float a[], float p[], long n) {
    long i;
    p[0] = a[0];
    for (i = 1; i < n - 1; i += 2) {
        float mid_val = p[i-1] + a[i];
        p[i] = mid_val;
        p[i+1] = mid_val + a[i+1];
    }
    if (i < n)
        p[i] = p[i-1] + a[i];
}
```

寄存器溢出

对于循环展开，很自然地考虑如下问题：是否展开的次数越多，性能提升越大？实际上，循环展开需要维护多个变量，一旦展开的次数过多，没有足够的寄存器保存变量，那么就需要将变量保存到内存中，这就会导致访存时间消耗增加。即便是在 x86-64 这样拥有足够多寄存器的架构中，循环也很可能在寄存器溢出之前就达到吞吐量限制，从而无法持续提升性能。