南開大學
Nankai University

计算机学院

并行程序设计实验报告

体系结构相关编程

姓名：张天歌

学号：2211123

专业：计算机科学与技术

2024 年 3 月 27 日

目录

摘要：本次实验将探究 Cache 优化、超标量优化两种对程序的优化方法以及带来的性能提升，从实验结果中探究并行程序设计的优越之处，并进行不同编译器的汇编代码分析，分析性能优化的原因。

关键字：并行程序设计；*Cache* 优化；超标量优化；循环展开；汇编代码

实验平台：x86 平台，Windows 11 64 位操作系统，CPU 型号 Intel Core i5-12500H，12 核 16 线程，主频 2500MHz，一级数据 4*48KB +8*32KB，L1 缓存 1.1MB，L2 缓存 9.0MB，L3 缓存 18.0MB，DDR5 16GB 内存，Visual Studio 2022 集成开发环境，godbolt 汇编分析浏览器。

1 基础要求

1.1 实验设计

1.1.1 计算矩阵与向量的内积

1) 逐列访问元素的平凡算法

按照矩阵的乘法运算，初始化一个大小为 n 的数组，用于存储每一列与给定向量的内积结果。对于一个 1*n 的向量 An 和 n*n 的矩阵 Bn 采用逐列访问元素的方法，依次将 Bn*n 的每一列和 An 相乘并累加，每次求出一个内积的结果并存储在之前初始化的数组中。

2)Cache 优化算法

由于按列访问方阵的元素容易造成更多的 Cache 缺失，使得运行效率降低，于是考虑从 Cache 的角度对算法进行优化。采用逐行访问矩阵元素的方法，依次将方阵 A 的第 i 行与 B 的第 i 个值相乘，每次得到最终向量结果的一个累加因子，存储在 sum[i] 中。

逐行访问元素矩阵元素时，一行内详列元素 A[i][j]、A[i][j+1] 地址相邻，且通常都在 cache 缓存中，当对 A[i][j] 执行操作后能快速方便的从 A[i][j+1] 执行下一步操作，省去了从内存读取数据的步骤，更加快速。

1.1.2 n 个数求和

1）平凡算法

即简单的顺序相加，遍历数组 A[N] 依次累加到 sum[i] 上，最终得到结果。

2）超标量优化算法

我采用的是两路链式算法，将循环展开，步长为 2，分别设置奇数组 sum1 和偶数组 sum2 接收循环中间变量的累加，最后将 sum1 和 sum2 的结果相加，算法执行了 N/2 次。同时执行两步运算，这样的算法节省了程序执行时间，体现了超标量的思想。

1.2 算法设计

1.2.1 计算矩阵与向量积的内积

本次实验数据规模是：5000*5000 的矩阵和 1*5000 的行向量，两者的向量内积是 1*5000 的向量 sum，具体定义和赋值详见 体系结构实验 1 代码。

(1) 平凡算法

```
1       for (int i = 0; i < 1000; i++) {
2        sum[i] = 0.0;
3        for (int j = 0; j < N; j++)  //平凡算法，逐列访问
```

```
        {
            sum[i] += b[j][i] * a[j];
        }
    }
```

(2)cache 优化算法

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)          //cache 优化，逐行访问
    {
        sum[j] += b[i][j] * a[i];
    }
}
```

1.2.2 n 个数求和

为了方便观察实验结果，本次实验的数据规模是 1e*6，并且循环 50 次求和操作。
(1) 平凡算法

```
    for (int i = 0; i < N; i++)   //平凡算法
    {
        sum0 += a[i];
    }
```

(2) 超标量优化算法

```
    for(int i=0;i<N;i+=2)
    {
        sum1+=a[i];           //优化算法
        sum2+=a[i+1];
    }
    sum0=sum1+sum2;
```

1.3 性能测试

1.3.1 计算矩阵与向量的内积

数据规模为 5000 时，计算程序的执行时间如下表所示：

平凡算法的平均执行时间为 159.1ms，cache 优化的平均执行时间为 48.1ms，加速比约为 3.3，cache 优化后的执行时间是平凡算法的 33%，性能提升近乎 2/3。

	1	2	3	4	5	6	7	8	9	10
平凡算法	147.6	161.7	162.7	154.7	155.1	162.8	154.9	162.1	166.1	163.6
cache 优化	48.5	46.7	45.5	47.5	51.6	46.8	47.9	49.6	49.2	47.6

表 1: 矩阵乘法性能测试结果 (单位:ms)

更改数据规模后得到以下结果:

问题规模	500	1000	2000
平凡算法	0.86	3.89	21.279
cache 优化	0.417	1.854	7.173
加速比	2.06	2.09	2.75

表 2: cache 优化性能测试结果 (单位:ms)

在问题规模分别为 500,1000,2000 时，加速比分别为 2.06,2.09,2.75，随着矩阵的规模不断成两倍扩大，加速比提升显著，柱状图如下所示:

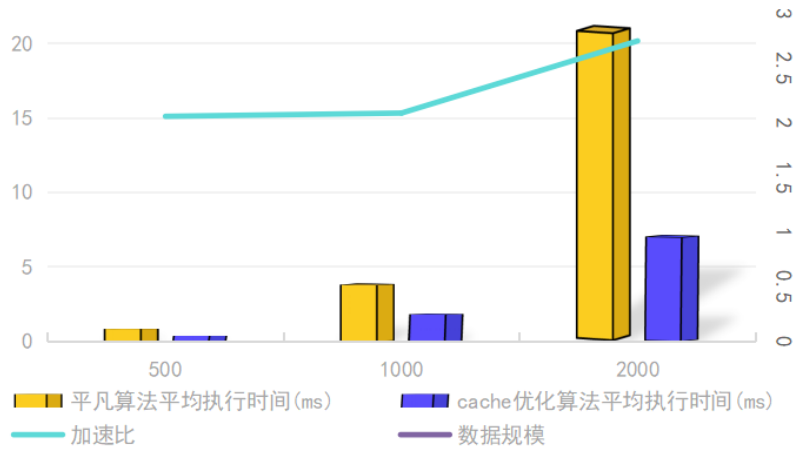

图 1.1: 平凡算法 VS cache 优化算法

综上所述，cache 优化算法后的程序性能优于平凡算法，且随着问题规模不断增大，这样的趋势诱发明显，在 2000 规模时，加速比达到 2.75，速度快了约 3 倍，这是因为 cache 优化算法的访存模式有更好的空间局部性，令 cache 的作用得以发挥，增加 cache hit 的频率，减少 cache miss 的频率，提高计算效率。

1.3.2 n 个数求和

用计算程序执行时间来衡量程序的性能，经过多次运行程序测试后得到程序的执行时间结果如下表:

	1	2	3	4	5	6	7	8	9	10
平凡算法	221.957	219.5	219.0	230.9	218.4	219.2	219.0	229.2	219.4	222.7
cache 优化	122.2	118.7	122.4	122.3	122.1	114.7	114.9	113.7	116.4	121.8

表 3: 矩阵乘法性能测试结果 (单位:ms)

每组经过 10 次重复计算后，平凡算法的平均执行时间为 221.9ms，超标量优化的平均执行时间为 118.9ms，加速比约为 1.866，超标量优化后的执行时间是平凡算法的 53.58%，性能提升近乎 1/2，而且可以预计随着数据规模的提升，这种性能提升会更加明显，更改数据规模后得到结果如下：

问题规模	103*50	104*50	105*50
平凡算法	0.23	2.66	22.3
两路链式算法	0.12	1.22	11.5
加速比	1.91	2.18	1.93

表 4: cache 优化性能测试结果 (单位:ms)

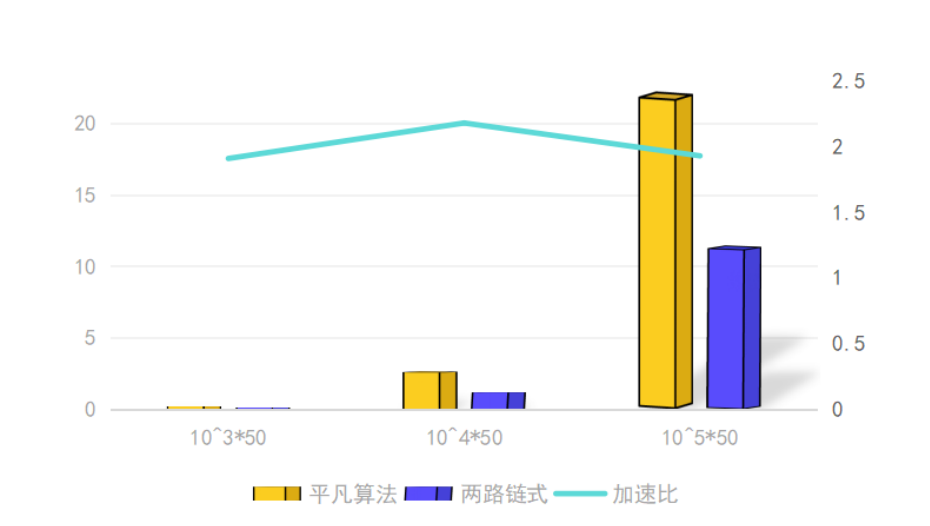

图 1.2: 平凡算法 VS 超标量优化算法

综上所述，两路链式优化算法后的程序性能优于平凡算法，且随着问题规模不断增大，加速比几乎维持在 2 不变，在 1e*5 规模时，加速比达到 1.93，速度快了约 2 倍, 这是因为两路链式优化算法一个循环同时存储两个奇、偶数组，运算效率快了一倍，循环次数少了 1/2。

2 进阶要求

2.1 循环展开操作

2.1.1 cache 优化

1. 设计思路

将内部循环展开两倍，每次迭代都会计算两个 j 值，并且对这两个值进行相应的乘法和赋值操作，这种操作可以减少循环控制的开销，每次迭代都会执行更多的计算工作。

2. 核心代码

(1) 按列相乘的循环展开操作

```
for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; j++)
    {   sum[i] += b[j][i] * a[j];
        if (j + 1 < N) {
            sum[i + 1] = b[j + 1][i + 1] * a[j + 1];
        }
    }
}
```

(2) 按行相乘的循环展开操作（cache 优化）

```
for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; j++) {
        sum[j] = b[i][j] * a[i];
        if (j + 1 < N) {
            sum[j + 1] = b[i + 1][j + 1] * a[i + 1];
        }
    }
}
```

3. 循环展开前后运行时间对比

		1	2	3	4	5	6	7	8	9	10	平均耗时	循环展开加速比
N=2000（循环展开后）	按列相乘ms	16.6	18.82	21.97	18.4	21.9	21.1	19.8	19.1	18.3	19	19.49895	3.36
	cache优化ms	5.4	6.02	5.84	5.38	6.02	6.1	5.81	6.76	5.83	5.4	5.856	
N=5000（循环展开后）	按列相乘ms	127.6	120.3	124.3	121.2	127.2	124.8	131.9	121.8	129.8	131.6	126.05	4.29
	cache优化ms	29.6	28.2	27.7	28.8	27.7	33.6	31.1	27.5	31.7	27.7	29.36	
N=2000（循环展开前）	按列相乘ms	21.1	22.9	22.5	21.8	21.3	20.3	19.7	19.5	23.2	20.2	21.25	2.75
	cache优化ms	6.4	7.5	6.5	7.6	7	7.7	7.4	7.1	7.1	7.3	7.16	
N=5000（循环展开前）	按列相乘ms	147.7	161.8	162.8	154.8	155.1	162.8	154.5	162.1	166.1	163.7	159.14	3.3
	cache优化ms	48.5	46.7	45.5	47.5	51.6	46.8	47.9	49.6	49.1	47.6	48.08	

图 2.3: 循环展开前后执行时间对比图

选取规模为 2000，5000 比较好观察的数据样本，每组记录 10 个用时情况，计算平均耗时和加速比，发现当同等规模时，循环展开处理后的矩阵乘法运算都加快了，就平凡算法而言，N=2000 循环展开后运行时间加快 9%，N=5000 循环展开后加快约 19%；cache 优化算法在循环展开后运行速度更快，N=2000 加速 20%，N=5000 时加速 40%。由此可见 cache 算法循环展开后程序的性能更优化了，其中的原因可能是 cache 缓存在指令中的高效性，高效率的 cache hit，较少的 cache miss。从计算机硬件基础层面来看，循环展开的性能优化原因有以下几点，以思维导图的效果呈现：

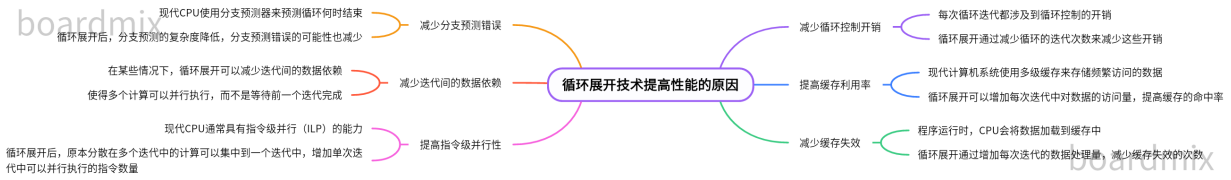

图 2.4: 循环展开的优点

2.2 cache 优化汇编代码分析

为了进一步探索性能表现的原因，这一部分使用了 godbolt 网页对同一个问题的平凡、优化算法进行汇编，且同一个算法使用编译器优化并进行分析。

2.2.1 平凡算法（gcc 13.2-O3 优化）

```
1       movsd   xmm0, QWORD PTR [rax]
2       unpcklpd        xmm0, xmm0
3       mulpd   xmm0, XMMWORD PTR [rdx-40000]
4       addpd   xmm1, xmm0
```

2.2.2 cache 优化算法 (gcc 13.2-O3 优化)

```
1       movapd  xmm0, XMMWORD PTR [rdx+rax]
2       movapd  xmm1, XMMWORD PTR [rbx+rax]
3       add     rax, 16
4       mulpd   xmm0, xmm2
5       mulpd   xmm1, xmm3
6       addpd   xmm1, XMMWORD PTR sum[rax-16]
7       addpd   xmm0, xmm1
8       movaps  XMMWORD PTR sum[rax-16], xmm0
```

平凡算法和 cache 优化的对比：

缓存优化算法与平凡算法在处理数组乘法任务时采取了不同的策略，这些策略在缓存利用、数据加载、计算效率和内存访问模式等方面展现出显著的差异。缓存优化算法通过精心设计的步长增加来跨越缓存行，这样可以减少缓存行冲突并提高缓存命中率。同时，它使用特定的指令来一次性加载更大的数据块，比如 128 位，而不是平凡算法中的 64 位，这样可以减少数据移动并提高数据加载的效率。

在计算过程中，缓存优化算法能够并行处理更多的数据，例如同时执行两个 128 位的乘法操作，这显著提高了计算效率。而平凡算法通常只处理单个数据点，每次迭代的计算量较小。此外，缓存优化算法在内存访问模式上也更为高效，它通过减少访问次数和提高访问的局部性来减少内存延迟，从而提升整体性能。

总的来说，缓存优化算法通过这些策略的综合运用，显著提升了程序的执行效率和缓存利用率，尤其是在处理大型数据集时，这种优势更为明显。而平凡算法虽然简单直接，但在性能上通常不如经过

缓存优化的算法。

2.3 两路链式算法的编译器优化分析

2.3.1 x86-64 gcc 13.2 编译器

```
        mov     eax, DWORD PTR sum1[rip]
        pxor    xmm1, xmm1
        cvtsi2sd        xmm1, eax
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        movsd   xmm0, QWORD PTR a[0+rax*8]
        addsd   xmm0, xmm1
        cvttsd2si       eax, xmm0
        mov     DWORD PTR sum1[rip], eax
        mov     eax, DWORD PTR sum2[rip]
        pxor    xmm1, xmm1
        cvtsi2sd        xmm1, eax
        mov     eax, DWORD PTR [rbp-4]
        add     eax, 1
        cdqe
        movsd   xmm0, QWORD PTR a[0+rax*8]
        addsd   xmm0, xmm1
        cvttsd2si       eax, xmm0
        mov     DWORD PTR sum2[rip], eax
        add     DWORD PTR [rbp-4], 2
```

2.3.2 x86-64 gcc 13.2 编译器-O3 优化

```
        cvtsi2sd        xmm0, edx
        addsd   xmm0, QWORD PTR [rax-16]
        cvttsd2si       edx, xmm0
        pxor    xmm0, xmm0
        cvtsi2sd        xmm0, ecx
        addsd   xmm0, QWORD PTR [rax-8]
        cvttsd2si       ecx, xmm0
```

编译器-O3 优化前后差异：

1. 减少指令数量：

优化后的代码通过减少指令数量来提高效率。例如，它移除了不必要的 mov 指令和 cdqe 指令，因为 rax 已经在之前的操作中被正确设置。

2. 避免重复操作：

优化后的代码避免了重复的初始化操作。在未优化的代码中，xmm1 被重复初始化和使用，而在优化后的代码中，xmm0 被连续用于两次操作，减少了初始化次数。

3. 更有效的内存访问：

优化后的代码通过更新 rax 寄存器的值来连续访问内存，而不是在每次操作后重新加载索引，这有助于提高缓存效率。

4. 寄存器使用：

优化后的代码更有效地使用了 xmm0 寄存器，避免了在两次操作之间重新加载浮点数。

5. 计算顺序：

优化后的代码调整了计算顺序，使得内存访问和转换操作更加紧凑，减少了潜在的流水线延迟。

6. 减少转换次数：

优化后的代码减少了浮点数到整数的转换次数，这是通过重用 xmm0 寄存器来实现的。

综上所述，优化后的代码通过减少指令数量、避免重复操作、提高内存访问效率、优化寄存器使用和调整计算顺序，提高了代码的执行性能。这些优化有助于减少 CPU 周期消耗，提高指令吞吐量，从而提升整体性能。

3 实验总结和思考

3.1 cache 优化算法

从程序算法的角度来讲，当逐行访问元素矩阵元素时，一行内详列元素 A[i][j]、A[i][j+1] 地址相邻，且通常都在 cache 缓存中，当对 A[i][j] 执行操作后能快速方便的从 A[i][j+1] 执行下一步操作，省去了从内存读取数据的步骤，更加快速。且在循环展开步骤中，cache 优化算法的高效性能得到了更好的展现。

从实验结果分析，缓存优化算法通过改进数据访问模式显著提高了程序性能。它利用连续内存访问来增加缓存命中率，减少了因缓存未命中而产生的额外数据加载时间。这种方法通过并行处理两个数据元素，有效利用了 CPU 的超标量架构，从而在处理大规模数据时保持了较高的计算效率和较低的时间开销。

3.2 超标量优化算法

从程序算法的角度来讲，平凡算法依次累加数组元素，累加 N 次得到结果，这样的穿行算法仅仅利用了 CPU 的一条工作线，性能较低，随着数据规模变大，执行时间更加缓慢；而优化后的两路链式算法一步执行两次累加操作，只用 N/2 就能得到结果，对比普通的链式算法，两路链式算法能更好地利用 CPU 超标量架构，两条求和的链可令两条流水线充分地并发运行指令。

从实验结果分析，两路链式优化算法后的程序性能优于平凡算法，随着数据规模的增长，两路链式算法展现出稳定的加速比，证明了其在处理大数据量时的性能优势。在 1e*5 规模时，加速比达到 1.93，速度快了约 2 倍。这种算法利用了超标量处理器的能力，通过并行执行多个计算步骤，减少了总体执行时间。