南開大學
Nankai University

计算机学院

并行程序设计实验报告

作业二：体系结构及性能相关测试

姓名：熊宇轩

学号：2010056

专业：计算机科学与技术

2022 年 3 月 13 日

目录

1 实验目标

1. 以矩阵每一列与向量的内积为例，通过编写代码实践 cache 优化算法。
2. 以求数组累加和为例，通过编写代码实践两路链式相加、循环展开和递归相加等超标量优化算法。
3. 利用 prof 和 uprof 等工具，通过运行计时和事件计数的方法，量化分析普通算法和优化算法之间的性能差异。

2 核心代码

代码仓库地址：Github

2.1 高精度计时函数

由于计时的代码比较复杂，写在 main 函数中显得太为冗杂。因此将计时代码封装为一个函数，将待测试的函数指针作为参数。实验在 X86 和 ARM 平台下都是通过 Linux 系统进行的，故使用了 Linux 系统的 clock_gettime 函数。

```
#include <time.h>
#include <sys/time.h>
#define REPT 200
//...
void test(int (*func)(int*, int), const char* msg, int* arr, int len)
{
    timespec start, end;
    double time_used = 0;
    cout << "result: " << func(arr,len) << "    ";
    clock_gettime(CLOCK_REALTIME, &start);
    int repeat = REPT*(int)pow(2,(20-(int)(log2(len))));
    for (int i = 0; i < repeat; i++)
        func(arr, len);
    clock_gettime(CLOCK_REALTIME, &end);
    time_used += end.tv_sec - start.tv_sec; // seconds used
    time_used += double(end.tv_nsec - start.tv_nsec) / 1000000000; // nanoseconds used
    cout << msg << ": " << time_used << endl;
}
```

2.2 矩阵列向量和向量点积函数

2.2.1 平凡算法

```
#define REPT 200
//...
```

```
int *common_algo(int mat[N][N], int vec[N], int n)
{

    int *sum = new int[n];
    for (int i = 0; i < n; i++)
    {
        sum[i] = 0;
        for (int j = 0; j < n; j++)
            sum[i] += mat[j][i] * vec[j];
    }
    return sum;
}
```

2.2.2 优化算法

```
int *optimized_algo(int mat[N][N], int vec[N], int n)
{
    int *sum = new int[n];
    for (int i = 0; i < n; i++)
        sum[i] = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            sum[j] += mat[i][j] * vec[i];
    return sum;
}
```

2.3 累加函数

2.3.1 平凡算法

```
int common_algo(int *arr, int len)
{
    int total = 0;
    for (int i = 0; i < len; i++)
        total += arr[i];
    return total;
}
```

2.3.2 N 路链式（循环展开）

一开始在本实验中，笔者试图使用模板的方式来进行循环展开，然而，GCC 编译器展开模板时，最高只能展开 900 层，而这样还不足以凸显出 Cache 大小和循环展开代码的性能之间的关系。而二路链式其实是最基本的循环展开方式，因此笔者使用 Python 编写了一个代码变成器，生成了最多 8192 路链式相加的代码。

```
int unroll_algo_1024(int *arr, int len)
{
    int total[1024] = {0};
    for (int i = 0; i < len; i += 1024)
    {
        total[0] += arr[i + 0];
        total[1] += arr[i + 1];
        total[2] += arr[i + 2];
        total[3] += arr[i + 3];
        total[4] += arr[i + 4];
        total[5] += arr[i + 5];
        total[6] += arr[i + 6];
        total[7] += arr[i + 7];
        //...
        //这里真的有 1024 行代码
        //...
    }
    return common_algo(total, 1024);
}
```

2.3.3 递归算法

```
int recursive_algo(int *arr, int len)
{
    for(int m = len; m>1; m/=2)
        for(int i=0; i< m/2; i++)
            arr[i] = arr[i * 2] +arr[i *2 +1];
    return arr[0];
}
```

3 实验环境

3.1 X86 平台

X86 平台使用的是一台笔记本电脑，运行 Linuxmint 20.3 una 系统，内存 8GB，内核版本为 Linux 5.4.0-100-generic。CPU 为 AMD 的锐龙移动端处理器 R5-3500U，具体参数如下：

```
suhipek@suhipek-BOHK-WAX9X:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   43 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       AuthenticAMD
CPU family:                      23
Model:                           24
Model name:                      AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
Stepping:                        1
Frequency boost:                 enabled
CPU MHz:                         1223.521
CPU max MHz:                     2100.0000
CPU min MHz:                     1400.0000
BogoMIPS:                        4191.85
Virtualization:                  AMD-V
L1d cache:                       128 KiB
L1i cache:                       256 KiB
L2 cache:                        2 MiB
L3 cache:                        4 MiB
NUMA node0 CPU(s):               0-7
```

其中显示的 L1 数据和指令缓存为四个 CPU 核心缓存数量的总和，每个 CPU 核心实际的数据缓存大小为 32KB。

3.2 ARM 平台

ARM 平台使用课程提供的服务器，运行 CantOS，内核版本 Linux 4.14.0-115.el7a.0.1.aarch64，CPU 为华为的鲲鹏 920，具体参数如下：

```
[s2010056@master screenFetch]$ lscpu
Architecture:          aarch64
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:   0-95
Thread(s) per core:    1
Core(s) per socket:    48
座:                    2
NUMA 节点:             4
型号:                  0
CPU max MHz:           2600.0000
CPU min MHz:           200.0000
BogoMIPS:              200.00
L1d 缓存:              64K
L1i 缓存:              64K
L2 缓存:               512K
L3 缓存:               49152K
NUMA 节点 0 CPU:       0-23
NUMA 节点 1 CPU:       24-47
NUMA 节点 2 CPU:       48-71
NUMA 节点 3 CPU:       72-95
```

4 实验过程

4.1 矩阵列向量和向量点积

首先，生成测试数据

```
g++ -O2 ./mar_vec_gen.cpp -o mar_vec_gen
./mar_vec_gen
```

然后，运行进行时间测量的 shell 脚本mat_test.sh，这个脚本会通过编译时宏定义（-D 选项）多次编译运行并输出结果

```
./mat_tesh.sh > res.csv
```

之后，通过 perf 进行事件计数的性能测量

```
g++ -g -DN=1024 ./matrix_product.cpp -o ./matrix_product # N 为矩阵行数
sudo perf record -e \
```

```
L1-dcache-load-misses,L1-dcache-loads,L1-dcache-prefetches,branches, \
branch-misses,cycles,instructions,idle-cycles-backend,idle-cycles-frontend, \
L1-icache-load-misses,L1-icache-loads,branch-load-misses,branch-loads \
-g -o perf_$(date +%m_%d_%H_%M).data ./matrix_product
```

对于 ARM 平台，则通过 qsub 命令提交这两个脚本：mat_timing.sh mat_perf.sh

4.2 累加

该实验的测试数据可使用上文中实验生成的数据。

首先生成代码，mar_vec_gen.cpp这个程序会根据unroll_sum.h的内容生成中unroll_sum.cpp中展开的函数：

```
python3 ./unroll_gen.py
```

进行时间测量，与上个实验不同，这个实验的时间测量是在程序内完成的，没有使用编译时宏定义

```
g++  -DN=65536 ./cumulative.cpp ./unroll_sum.cpp -o cumulative
./cumulative > sum_timing.csv
```

使用 perf 进行事件计数

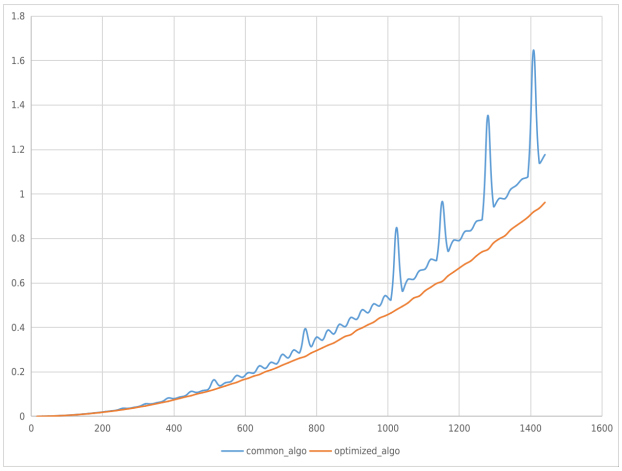
```
g++ -D USE_FIXED_N -DN=1048576 ./cumulative.cpp ./unroll_sum.cpp -o cumulative
sudo perf record -e \
L1-dcache-load-misses,L1-dcache-loads,L1-dcache-prefetches,branches, \
branch-misses,cycles,instructions,idle-cycles-backend,idle-cycles-frontend, \
L1-icache-load-misses,L1-icache-loads,branch-load-misses,branch-loads \
-g -o perf_$(date +%m_%d_%H_%M).data ./cumulative
```

对于 ARM 平台，则通过 qsub 命令提交这两个脚本：sum_timing.sh sum_perf.sh

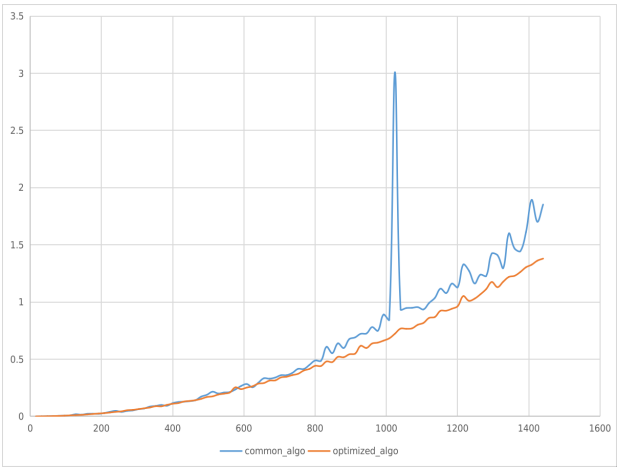
5 实验结果

5.1 运行计时

矩阵列向量点积实验运行计时的结果如下图所示，其中，矩阵行列数在实验中以 16 为步进。



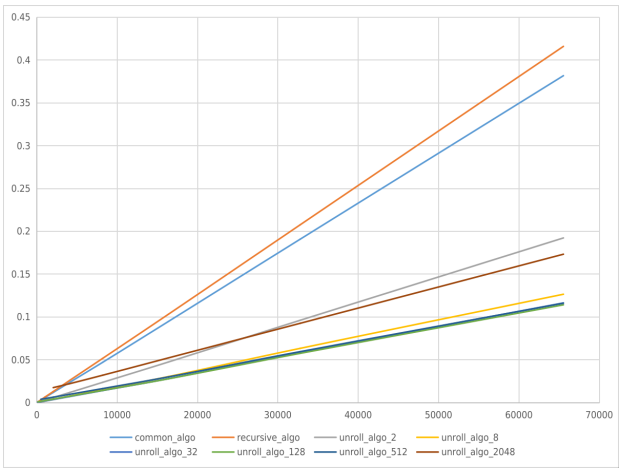
(a) ARM 平台



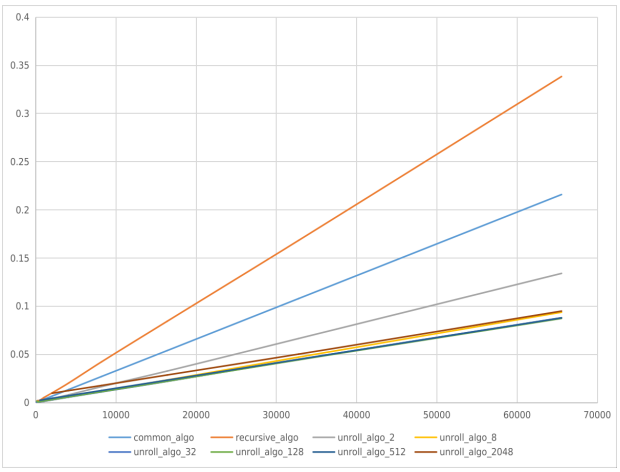
(b) X86 平台

图 5.1: 矩阵列向量点积 (矩阵行/列数-运行时间)(单位/秒)

累加实验运行计时的结果如下图所示



(a) ARM 平台



(b) X86 平台

图 5.2: 累加 (输入数组长度-运行时间)(单位/秒)

5.2 事件计数

函数	L1-dcache-load-misses	L1-dcache-loads	L1-dcache-store-misses	L1-dcache-stores
common_algo	86.84%	50.00%	86.83%	50.00%
optimized_algo	13.13%	49.99%	13.12%	49.99%
事件总数	30070924	3570518704	30070932	3570537681

表 1: ARM 平台下矩阵列向量与向量点积的 perf 事件计数结果

函数	L1-dcache-load-misses	L1-dcache-loads	L1-dcache-prefetches
common_algo	93.55%	50.61%	92.10%
optimized_algo	6.33%	49.23%	7.67%
事件总数	105630435	3161589638	80585651

表 2: X86 平台下矩阵列向量与向量点积的 perf 事件计数结果

优化方法或 循环展开的层数	cycles	instructions	CPI	L1-dcache-load-misses	L1-icache-load-misses
不优化	13.41%	9.19%	0.6296	8.24%	0.00%
递归	15.55%	18.67%	0.3594	15.01%	0.00%
2	6.73%	7.07%	0.4107	6.50%	0.00%
32	4.14%	5.21%	0.3429	5.06%	0.00%
512	4.15%	5.10%	0.3511	6.01%	0.00%
8192	9.89%	6.51%	0.6555	8.04%	45.59%
事件总数	122855688833	284706374706	0.4315	633713953	2540277364

表 3: ARM 平台下累加优化算法的 perf 事件计数结果

优化方法或 循环展开的层数	cycles	instructions	CPI	L1-dcache-load-misses	L1-icache-load-misses
不优化	11.81%	6.92%	0.6762	5.70%	2.10%
递归	22.60%	17.53%	0.5108	16.13%	4.13%
2	7.24%	6.70%	0.4281	5.51%	1.46%
32	4.84%	5.76%	0.3329	5.36%	1.30%
512	4.64%	5.59%	0.3288	5.38%	2.10%
8192	5.05%	5.69%	0.3516	10.48%	61.37%
事件总数	52560137817	132651641398	0.3962	889797839	60817536

表 4: X86 平台下累加优化算法的 perf 事件计数结果

6 数据分析

6.1 Cache 优化

从运行计时的结果图5.1可以看出，当输入数据规模超出某一个值后，优化的行主访问模式相较于列主访问，在运算时间上产生了较大的优势。根据事件计数的结果表1，ARM 平台上, 在 L1-dcache-

loads、L1-dcache-stores 事件数量相差无几的前提下，列主访问的平凡算法所产生的 L1-dcache-load-misses、L1-dcache-store-misses 的事件数量是行主访问的优化算法的 6 倍以上。这表明了，列主访问不能最大化发挥 cache 的潜能，这应该也是行主访问在运行计时上优于列主访问的原因。

6.2 超标量优化

从运行计时的结果图5.2可以看出，无论是两路链式，还是循环展开，其运行速度都优于平凡算法，根据表3可以看出，平凡算法的 CPI 为 0.6296，两路链式的 CPI 为 0.4107，而 32 路链式的 CPI 为 0.3429，这表明在一定范围内，通过增加单次循环中所执行的操作数来进行循环展开的优化，是可以充分利用 CPU 流水线的设计，得到一定的性能收益的

6.3 其它问题分析

6.3.1 列向量点积数据中的尖峰

不难注意到，在图5.1中，ARM 平台平凡算法的数据有四个明显的尖峰，除了这四个尖峰外，整个曲线表现出有规律的上下波动。经过几次重复实验后，发现曲线上相近的位置上依然存在着尖峰。并且查看尖峰对应的输入矩阵行数，发现它们恰巧对应着 1024、1152、1280、1408（delta=128）这四个数值。

在 ARM 平台上 perf 事件计数测量时发现，当矩阵行数为 1024 时，L1-dcache-load-misses 的事件数量比矩阵行数为 1025 时多了 60%。进一步分析汇编代码，发现这些事件基本都集中在提取矩阵 mat[j][i] 这个操作上。查阅资料后发现，这个问题可能和 cache 的结构有关，在鲲鹏 920 处理器上，Cache 划分成了 128 字节的小块，这种小块被称为 Cache Line，而同一个 Cache Line 是不可被同时访问的。由于出现尖峰的曲线对应的是列主访问的算法，行列数为整倍数的矩阵导致每次从 Cache 中提取元素都需要跨行访问，导致了额外的开销。也有可能由于超标量的特性，可能存在两条指令同时需要修改同一个 Cache Line 中的内容的情况，这就造成了伪共享（false sharing）现象，从而导致 Cache 命中率降低。

图 6.3: Cache 的结构 - Cache Line[1]

而在 X86 平台上，平凡算法的曲线上也存在一个尖峰。perf 分析发现，矩阵行数为 1024 时，branch-misses 事件数量比比矩阵行数为 1025 时多了 50%，而 L1-dcache-load-misses 的事件数量则是相近的。这可能是由于不同平台的 PMU（性能测量单元）或者 Cache 结构存在着差别，而 X86 平台上出现尖峰的原因与 ARM 平台是类似的。也可能是 4M 的 L3 Cache 被矩阵占满后，数据存储导致了额外的开销。

其实，在程序性能优化中，Cache Line 对齐是非常重要的一环，通过对数据结构的优化，程序可以获得可观的性能提升。

6.3.2 ARM 和 X86 平台的区别

不难注意到，在矩阵列向量点积的实验中，ARM 平台的总体 Cache 命中率要比 X86 平台高很多（99.158% vs 96.659%），这可能是由于 ARM 平台有着更加智能的 Cache 访问模式，也可能仅仅是因为 ARM 平台的 L1 缓存更大（64KB vs 32KB）。此外，不难发现 ARM 平台的累加的实验的运行时间要比 X86 平台长得多，这很有可能是因为指令集架构的不同导致的。

6.3.3 循环展开的层数

在图5.2中，可以发现并不是循环展开的路数越多，算法的性能就越好。比如在 ARM 平台上，2 路展开就和 8192 路展开运行的速度差不多。根据表3和表4的 perf 性能分析发现，8192 路展开产生了大量的 L1-dcache-load-misses 和 L1-icache-load-misses 事件，导致了性能的降低。

导致这现象一方面的原因是，展开循环后的每一行代码依然会产生一定的 misses 事件，积少成多便降低了 Cache 命中率。此外，输入数据达到一定规模后，L1 Cache 便不再能存下所有的数据了，因此后面的展开会造成大量的性能开销。

这个现象告诉程序设计者们，进行循环展开时的层数也不是越高越好的：展开的层数太高不仅会导致程序体积的变大，还不能给性能带来任何正面的提升。

6.3.4 递归算法的负优化

在图5.2中，显而易见的，递归算法是所有算法中效率最低的一个，这明显是不符合直觉的。分析表3和表4后发现，递归算法产生了大量的 L1-dcache-load-misses 事件，这很有可能是因为递归算法需要进行大量不连续内存的访问，跨越 Cache Line 或者 Cache Line 伪共享很可能是命中率降低的罪魁祸首。

此外，递归算法的代码本身就比平凡算法和多路链式算法要复杂，多出来的变量也很可能造成了不小的额外开销。

参考文献

[1] 华为云. cacheline 优化 ,Feb2022.