Cache 实验报告

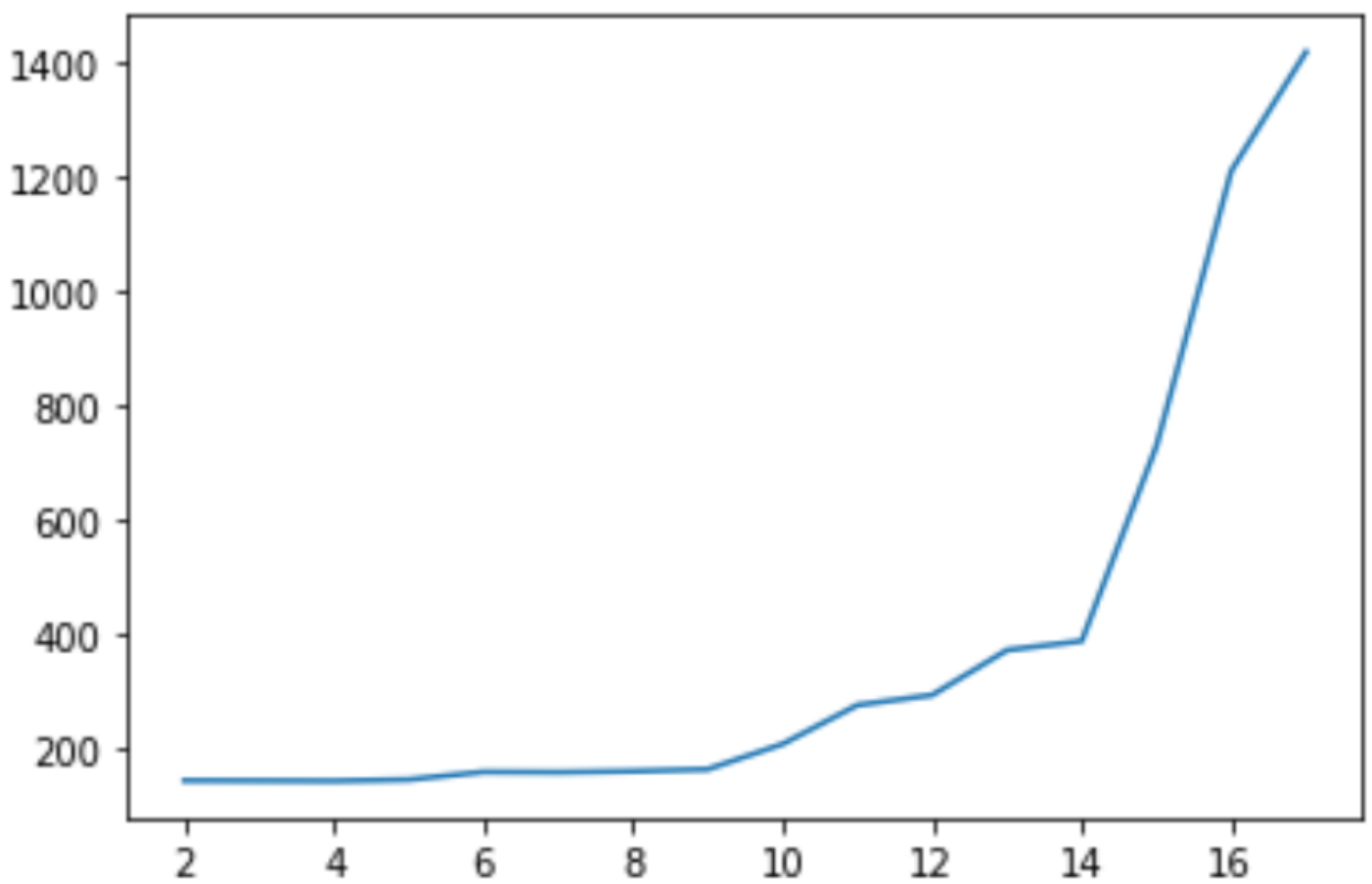
范逍宇 **2019013273**

0.实验环境

- 操作系统：linux ubuntu 20.04
- CPU 型号：Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz
- L1 d-cache：64 KiB
- L1 i-cache：64 KiB
- L2 cache：2 MiB
- L3 cache：33 MiB
- 使用实验文档中给出的进程绑定方法。

1.Cache Size

- 访存序列：对数组以 59 为步长进行访问，为了保持不同长度的数组的总访问次数相同，在访问下标超过数组长度时对 数组大小取模继续访问，直到达到总访问次数。
- 实验结果：
  - 数组所占空间大小从 4KB 开始增大，每次扩大为原来的 2 倍，最大为 64 MB。
  - 对所有数组均访问 2^{26} 次。
  - 用折线图展示访存时间随数组大小的变化如下：（横轴为 $\log_2 ArraySize$）



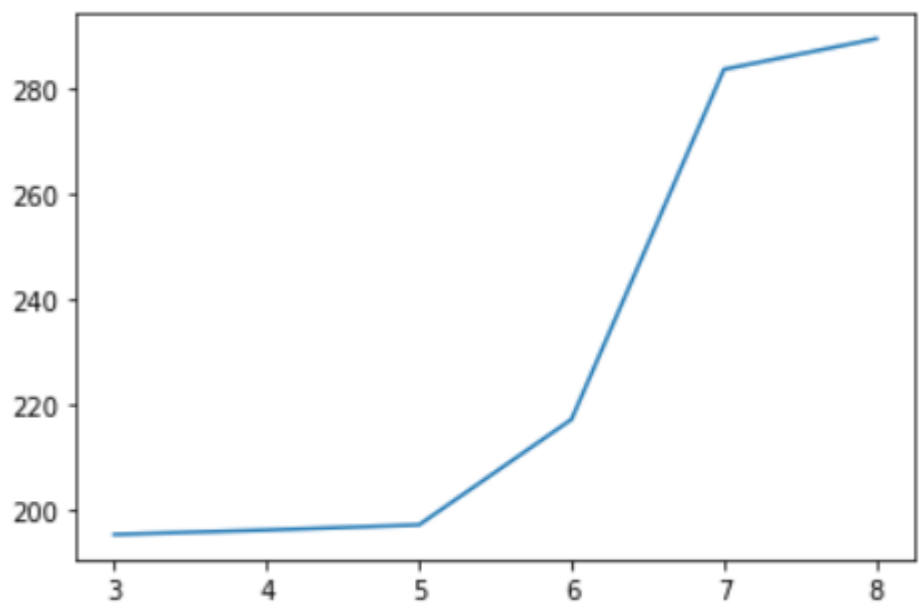
  - 不同大小数组具体的访存时间如下：

```
(base) root@iZ2ze3ps1pde59xbp2uiv6Z:/home/cache# ./test_cache
Test cache size ------------------
4 KB: 143.33 ms
8 KB: 143.06 ms
16 KB: 142.67 ms
32 KB: 145.01 ms
64 KB: 158.46 ms
128 KB: 157.95 ms
256 KB: 159.79 ms
512 KB: 163.18 ms
1024 KB: 207.34 ms
2048 KB: 275.56 ms
4096 KB: 293.06 ms
8192 KB: 371.88 ms
16384 KB: 387.71 ms
32768 KB: 729.82 ms
65536 KB: 1211.46 ms
131072 KB: 1419.71 ms
```

- 可以看到当数组大小为 $64\ KB, 2MB, 32MB$ 时访存时间明显增大，它们分别对应于 $L1\ d-cache, L2\ cache, L3 cache$ 的大小。
- 这里一开始的访存时间就比较大，这是因为在计时的部分除了进行访存的操作，还进行了乘法、取模、自增等操作，运算的基础时间就比较长。
- 这里访存时间在数组大小为 $64KB$ 时增大，而不是在数组大小为 $128KB$ 时增大，这可能与 cache 的替换算法有关，如果 cache 中只有数组数据，理论上说应该在数组大小为 $128KB$ 时增大，但是如果 cache 中还放了其他数据，那么就无法完整地放入整个数组，在数组大小小于 cache size 时 cache 就已经饱和，这样，在数组大小等于 cache 大小时 cache 中就会发生频繁的替换，从而使访存的时间变大，对于 L2, L3 cache 也有类似的现象。
- 总的来说，实验数据很好地验证了各级 cache 的大小。

2.Cache Line Size

- 访存序列：数组大小固定，对数组分别以 2^k 为步长进行访存（超出数组大小则对数组大小取模），各种步长的总访问次数均为 2^{26} 。
- 实验结果：
  - 用折线图展示数组访问时间与步长的关系：（横轴为 $\log_2$ 步长，纵轴为访问时间）



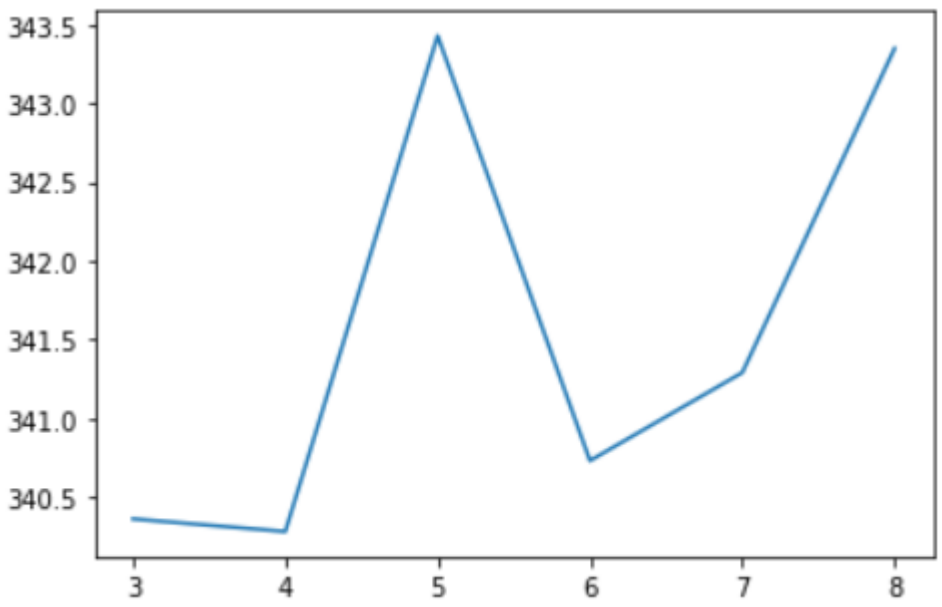
  - 不同步长具体的访问时间如下：

```
(base) root@iZ2ze3ps1pde59xbp2uiv6Z:/home/cache# ./test_cache
Test cacheline size ----------------
8: 195.12
16: 195.97
32: 196.96
64: 217.04
128: 283.70
256: 289.53
```

  - 可以看到步长为 64 时访问时间突然增大，说明 cache 的缺失率增大，说明 L1 d-cache 的 Cache Line Size=64B。

3.相联度

- 访存序列：根据说明文档中给出的算法对先对数组进行分块，然后依次访问数组中奇数块中的元素，循环访问整个数组，直到访存次数达到预先设定的总访问次数。
- 实验结果：
  - 不同分块个数对应的访存时间如下图：



  - 不同分块个数对应的具体的访存时间如下：

```
Test the associativity --------------
8: 340.36
16: 340.28
32: 343.43
64: 340.73
128: 341.29
256: 343.35
```

  - 可以看到，当分块个数为 32 时访存时间明显增大（计时的时间除了访存还进行了其他运算操作，所以基础时间比较长，但仍能看出明显的增大），根据我们的算法，这说明L1 d-cache 的相联度为 8 。
- 算法解释：
  - 假设相联度为 2（相联度为其他数值时可以用同样的分析方法），数组的大小为 2 倍的 L1 d-cache 的大小，将数组按顺序划分为同样大的 $a_1, a_2, \ldots, a_8$ 这 8 部分，这样在第一次访问时，数组被分为 $(a_1, a_2, a_3, a_4), (a_5, a_6, a_7, a_8)$ 两个大块，奇数块正好全部放入 cache 中，且 cache 的第一路为 a_1, a_2 ，第二路为 a_3, a_4，在第二次访问时，数组被划分为 $(a_1, a_2), (a_3, a_4), (a_5, a_6), (a_7, a_8)$ 四个大块，其中奇数块为 $(a_1, a_2), (a_5, a_6)$，则 a_3, a_4 被替换为 a_5, a_6 ，缺失率为 50%，（如果相联度大于2，接下来的缺失率将仍为 50%），第三次访问时，数组被划分为 8 个块，其中奇数块为 a_1, a_3, a_5, a_7 ，但不同的是这几个奇数块的 hash 对应的都是 cache 每一路的上半部分，相当于 cache 有用的部分变为原来的一半，所以这时缺失率会增加，不难发现这时的数组块数正好为cache相联度的四倍。

4.矩阵优化

按照老师讲授的方法，改变矩阵的运算顺序，原算法为：

```
for (i = 0; i < MATRIX_SIZE; i ++)
    for (j = 0; j < MATRIX_SIZE; j ++)
        for (k = 0; k < MATRIX_SIZE; k ++)
            c[i][j] += a_[i][k] * b_[k][j];
```

注意到如果这样访问，最后一层中 $a[i][k]$ 和 $b[k][j]$ 都是在不断变化的，而且 $b[k][j]$ 是按列访问的，考虑将其改为：

```
for (i = 0; i < MATRIX_SIZE; i ++)
        for (k = 0; k < MATRIX_SIZE; k ++){
            int r=a[i][k];
            for (j = 0; j < MATRIX_SIZE; j ++)
                d[i][j] += r * b[k][j];
        }
```

这样，调换了 k 和 j 的访问顺序，使得最后一层中 $a[i][k]$ 不变，而且 $b[k][j]$ 是按行访问的，这样就利用到了数据的局部性，发挥了 cache 的作用。

最终的优化效果如下：

```
(base) root@iZ2ze3ps1pde59xbp2uiv6Z:/home/cache# g++ matrix_mul.cpp -o matrix_mul -O0
(base) root@iZ2ze3ps1pde59xbp2uiv6Z:/home/cache# ./matrix_mul
time spent for original method : 7.47584 s
time spent for new method : 2.62853 s
time ratio of performance optimization : 2.84411
```

性能提升倍数为 2.844.