缓存测量实验报告

计04 何秉翔 2020010944

1. 实验机器 Cache 参数

使用 `getconf -a` 命令可以获得实验机器上的 Cache 参数如下：

```
...
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
...
```

我们再使用 `lscpu` 命令得到部分参数如下：

```
Model name:                      Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
...
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        8 MiB
```

所用实验的硬件环境为 `Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz`，各层级 Cache 的详细信息如下：

Cache	数目	单个大小	总大小	块大小	相联度
L1 I-Cache	4	$32KB$	$128KB$	$64KB$	8
L1 D-Cache	4	$32KB$	$128KB$	$64KB$	8
L2 Cache	4	$256KB$	$1MB$	$64KB$	4
L3 Cache	1	$8MB$	$8MB$	$64KB$	16

2. 必做部分

2.1 步骤 1: Cache Size

实验代码： `lab1.cpp`

编译指令： `make lab1`

实验结果： `lab1.txt`、`lab1.png`

实验结果绘图代码： `lab1_plot.py`

2.1.1 实验思路

我们设置不同的 `test_size`，具体而言，$test\_size = 2^i KB$, $i = 0, 1, \ldots, 11$，对应地我们设置一系列的 `int` 数组长度 $INT\_ARRAY\_SIZE = 256 \times 2^i$，对于每一个 i，我们做如下工作：

- 我们测试共 $ROUNDS = 2^{30}$ 轮访存，根据实验指导手册的建议，我们采取 `store` 的访存方式。
- 先将数组元素全部载入到缓存中。
- 以 32 为步长来循环访存数组。

2.1.2 访存序列

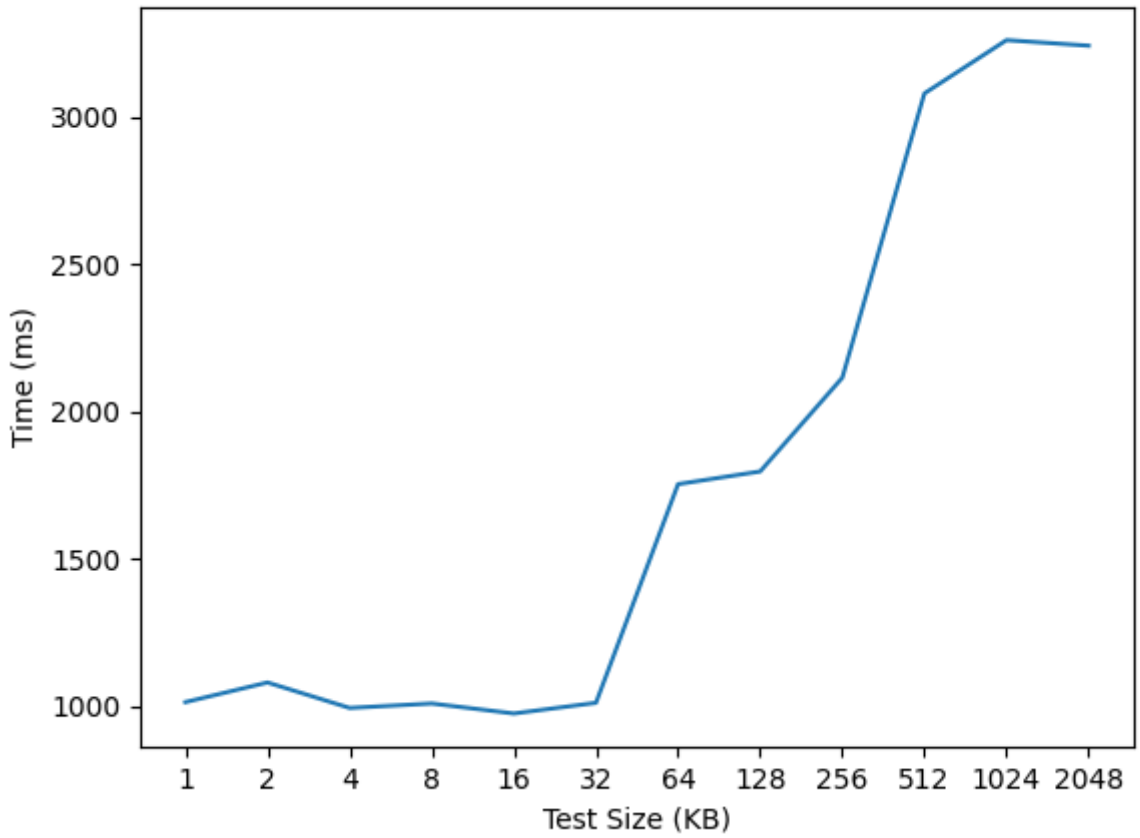
由前面可知访存序列以 32 为步长，因此访问的数组下标依次为：

```
0, 32, 64, ... , INT_ARRAY_SIZE - 32, 0, ... (循环访问)
```

2.1.3 执行结果

TEST_SIZE(KB)	TIME(ms)
1	1012.83
2	1079.55
4	993.045
8	1008.18
16	974.222
32	1010.49
64	1753.28
128	1796.38
256	2114.85
512	3080.36
1024	3261.47
2048	3242.52

测试结果统计如下图：



可以发现，在 $32KB$ 和 $256KB$ 附近，访存时间出现了最剧烈的跳变，当数组大小超过 L1 D-Cache 的大小之后，会出现 L1 D-Cache 的读缺失，平均读取速度会显著增加，而且对于 L2 Cache 同理。因此可以推知 L1 D-Cache 的大小为 $32KB$，L2 Cache 的大小估计为 $256KB$，结果基本匹配。

但是我们也发现，实际上在 $128KB$ 时也发生了稍微明显的跳变，这也就是实验指导手册里提到的**思考题**：

> **思考题：**
>
> 理论上 L2 Cache 的测量与 L1 D-Cache 没有显著区别。但为什么 L1 D-Cache 结果匹配但是 L2 Cache 不匹配呢？你的实验有出现这个现象吗？请给出一个合理的解释。
>
> 提示：DCache v.s. Cache

事实证明，在我们的实验中，确实出现了这种现象。因为 L1 D-Cache 所存的全部都是数据，因此在循环访存的过程中只涉及到这些数组地址和数组元素。而 L2 Cache 不区分指令 Cache 和数据 Cache，因此即便我们将数组全部载入缓存，仍然不可避免将部分指令地址和指令给缓存到 L2 Cache 中，因此实际能有效 cache hit 的数据访存小于 $256KB$，因此测得的 L2 Cache 似乎偏小。

2.2 步骤 2: Cache Line Size

实验代码： lab2.cpp

编译指令： make lab2

实验结果： lab2.txt、lab2.png

实验结果绘图代码： lab2_plot.py

2.2.1 实验思路

我们由 `coherency_line_size` 可知 `cache line` 大小为 $64B$，我们设置可变参数 `stride(B)`，即访存步长，分别取值 `4, 8, 16, 32, 64, 128, 256, 512`：

- 我们测试共 $ROUNDS = 2^{30}$ 轮访存，根据实验指导手册的建议，我们采取 `store` 的访存方式。
- 先将数组元素全部载入到缓存中。
- 以 `stride` 为步长来循环访存数组。

2.2.2 访存序列

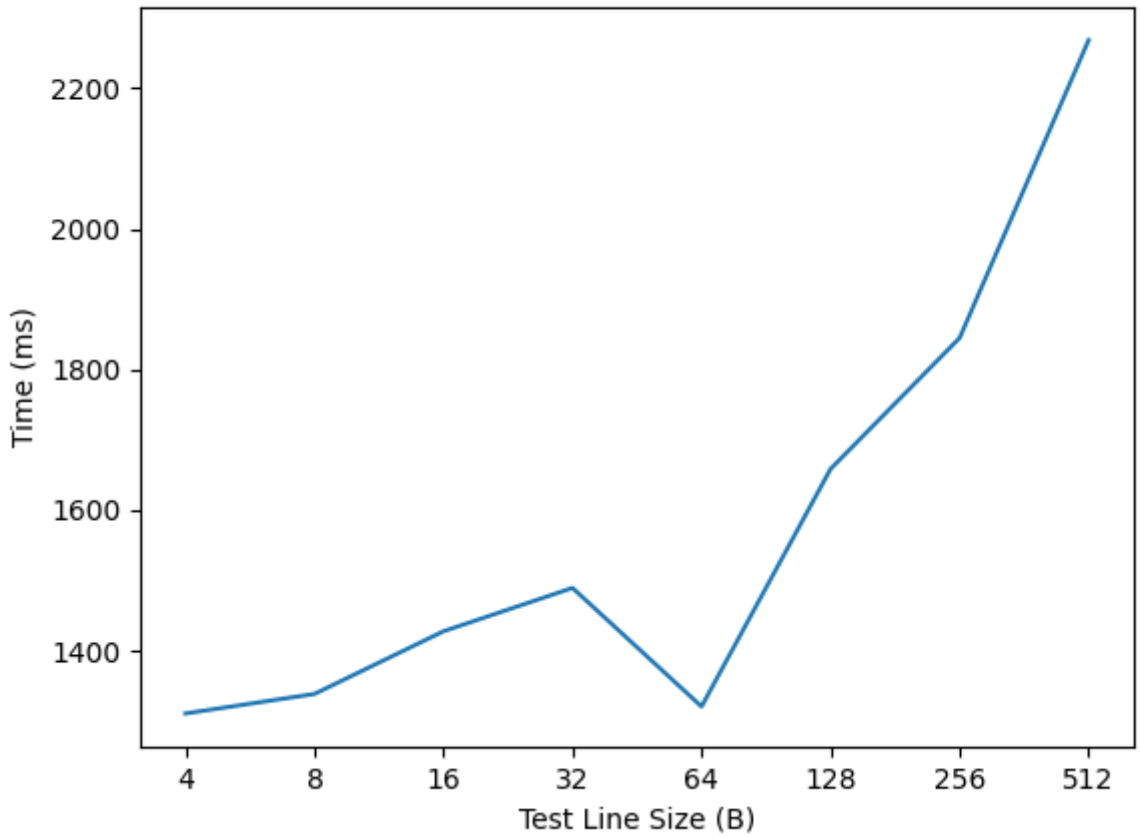
对于不同的 `stride` 有不同的访存序列，比如对于 `stride = 8B`，访问的数组下标依次为：

```
0, 2, 4, 8, ..., INT_ARRAY_SIZE - 2, 0, ... (循环访问)
```

2.2.3 执行结果

STRIDE(B)	TIME(ms)
4	1310.87
8	1338.64
16	1427.77
32	1489.42
64	1320.79
128	1658.68
256	1844.92
512	2268.51

测试结果统计如下图：



可以发现运行时间在 64B 步长之后开始较为显著的跳变，与预期相符，说明 L1 D-Cache Line 为 $64B$。

> **思考题：**
>
> Prefetch 对于该实验的结果有影响吗？给出你的分析和结论。

Cache 预取对实验的结果是有影响的，尤其是对于顺序访问这种有规律的访问模式，虽然以超出 Cache Line 的步长访问基本上会 miss，但如果 Cache 推测接下来大概率顺序访问，其有可能将后面的 Cache Line 一起载入到 Cache 中，可能带来 hit 的情况。比如我们看到实验结果在超过 $64KB$ 之后有变缓，这有可能是预取带来的影响。

2.3 步骤 3: Cache Associativity

实验代码： lab3.cpp

编译指令： make lab3

实验结果： lab3.txt、lab3.png

实验结果绘图代码： lab3_plot.py

2.3.1 相联度算法分析

我们采用实验思路 1：

> - 使用一个 2 倍 Cache Size 大小的数组
> - 将数组分为 2^n 块，只访问其中的奇数块
> - 逐渐增大 n 的取值，当某一次访问变慢的时候，2^{n-2} 就是相联度。

对于 L1 D-Cache，我们设置 $64KB$ 大小的数组，按照实际的组相联度 8，我们需要将数组分为 32 块，每一块 $2KB$，只访问奇数块时，共访问 16 次，间隔范围为 $2KB \times 2/64B = 64$ 个缓存行，因此每次访问都将访问到 Cache 的同一组内。对于组相联数 8，前 8 次访问填满这一组后，当超过 8 次访问时，每次访问都将进行 cache 驱逐，导致访问变慢。可以将此时的 cache 访问模式视作**在 cache line 阵列上画了两道水平线。**

- 若实际数组分组少于 32 块，如果前一次访问到 Cache 的第 0 组，则下一次访问到 Cache 的第 1 组、第 2 组等等，而不会正好是第 0 组。
- 若实际数组分组大于 32 块，如果前一次访问到 Cache 的第 0 组，则下一次访问到 Cache 的第 63 组，第 62 组等等（L1 D-Cache 总共 64 组），而不会正好是第 0 组。

因此这两种情况下，前 8 次访问后，每一组的 Cache 均未填满，此时后 8 次访问将不一定造成 cache 驱逐，以至于耗时较少。

2.3.2 实验思路

我们由 ways_of_associativity 可知相联度为 8，我们设置可变参数 test_ass，即测试相联度，分别取值 1, 2, 4, 8, 16, 32, 64, 128：

- 我们测试共 $ROUNDS = 2^{30}$ 轮访存，根据实验指导手册的建议，我们采取 load + store 的访存方式，比如 a[i] += j。
- 然后按照实验思路 1 完成实验。

2.3.3 访存序列

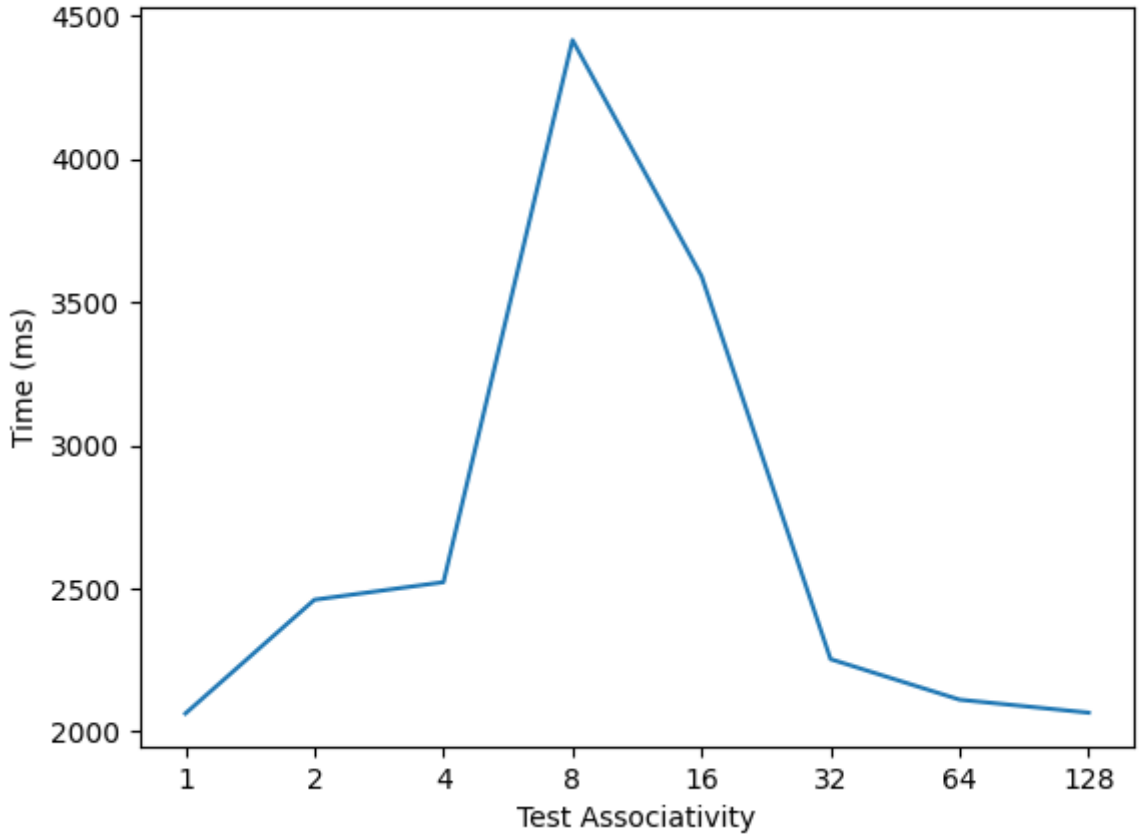
对于不同的 test_ass 有不同的访存序列，比如对于 test_ass = 8，此时数组将被分为 32 块，每块有 $2KB$，共 512 个 int 元素，由于访问奇数块首元素，因此访问的数组下标依次为：

```
512, 512 + 1024, 512 + 2 * 1024, ... , (循环访问)
```

2.3.4 执行结果

TEST_ASS	TIME(ms)
1	2063.29
2	2460.84
4	2522.18
8	4416.57
16	3591.72
32	2253.29
64	2111.59
128	2066.15

测试结果统计如下图：



发现当 `Test Associativity = 8` 时，访存时间最长，根据实验思路 1，说明 `8` 为 `L1 D-Cache` 的相联度，与预期相符。

2.4 步骤 4: MatMul Optimization

实验代码： `lab4.cpp`

编译指令： `make lab4`

实验结果： `lab4.txt`

实验脚本： `run_lab4.sh`

我们尝试的优化方法如下：

2.4.1 更改运算顺序

我们注意到原始的运算顺序为 `i, j, k`：

```
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];
```

对于具体的计算，发现最内层循环 `k` 时，对于 `b` 数组，将会间隔访存，使得 `cache miss` 的可能性很大，因此我们调整 `j, k` 的顺序，使得最内层循环的访存都是连续访存：

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

2.4.2 矩阵分块

为了更高效地利用 `Cache`，我们将矩阵进行分块来计算，设置分块大小 `BLOCK_SIZE`：

```
for (i = 0; i < MATRIX_SIZE; i += BLOCK_SIZE)
    for (k = 0; k < MATRIX_SIZE; k += BLOCK_SIZE)
        for (j = 0; j < MATRIX_SIZE; j += BLOCK_SIZE)
            for (i_b = i; i_b < i + BLOCK_SIZE; i_b ++)
                for (k_b = k; k_b < k + BLOCK_SIZE; k_b ++)
                    for (j_b = j; j_b < j + BLOCK_SIZE; j_b ++)
                        d[i_b][j_b] += a[i_b][k_b] * b[k_b][j_b];
```

经过测试，我们选择分块大小 `BLOCK_SIZE = 16`，使得一个矩阵分块内部的计算涉及的空间大小不会超过 $32KB$，实际上为 $16^2 \times 3 \times 4B$。

2.4.3 执行结果

由于性能波动较大，尤其是 `original method` 的波动较大，我们测试十次（`run_lab4.sh`），将十次结果报告如下：

实验编号	original method(s)	new method(s)	加速比
1	5.79324	2.40867	2.40516
2	7.10879	2.38955	2.97494
3	4.41774	2.35742	1.87397
4	8.68168	2.41231	3.5989
5	8.52914	2.5024	3.40838
6	8.27568	2.52541	3.27696
7	5.52437	2.48289	2.22498
8	8.84209	2.57648	3.43185
9	11.7632	2.63929	4.45695
10	6.31679	2.59928	2.43021

我们发现，优化后的算法时间稳定在 $2.5s$ 左右，有半数的测试实验达到了大于 3 的加速比，我尝试了一台 `i5` 的机器来跑 `lab4` 的二进制文件，发现加速比稳定在 4 以上，因此推知可能是不同机器带来的影响。

3. 意见 & 建议

1. 本地测试过程中，使用了 `register`，并且绑定了 `core`，似乎实验测试波动还是比较大，对于分析稍微有些困难。
2. 对于矩阵乘优化实验中，原始方法的性能波动较大，相对而言会对加速比的结果造成影响。
3. 对于选做实验，希望能提供一些参考的相关的资料。

最后实验框架以及实验文档写的非常详细！谢谢助教和老师的精心准备！