Homework 5

Exercise 1: Cache Visualization

1.1 场景1

> Cache命中率

0

> 为什么会是这个命中率

一个数据块可以装2个`int`，一共4个数据块，可以装8个；每个数据的内存地址的第3.4位用来编码Cache中的行，那么就是每8个`int`映射到Cache的同一个位置，而这里的访问步长就是8，每次都会映射到第一行，那么每次都会丢弃，命中率就是0

> 增加Rep Count的值

不行，因为这不能增加最内层循环的命中率

> 如何修改才能最大化hit rate

只能修改Step Size，并且因为一块内只能放2个`int`，将Step Size变为1后最大的hit rate才是0.5，取一次只能取两个，前一个miss后一个hit

1.2 场景2

> Cache命中率

0.75

> 为什么会是这个命中率

Cache能够装下整个array，way用第6.7位编码，因此不会有数据被置换

在读第一个元素时，将后续的3个`int`共4个`int`一起拿进来，因此连续的两次内层循环中，只有偶index元素的读会miss，偶index元素的写、奇index元素的读写都hit，因此是0.75

> 增加Rep Count的值

100%，实际上近第一次的上述操作会存在miss，如上所说整个数组都会被编到Cache，那么重复次数大起来之后就不在乎第一次的miss了，因此是100%

1.3 场景3

> L1 Cache和L2 Cache的命中率

L1：0.5

L2：0

> 总共访问了几次L1 Cache，miss次数是多少

访问了32次，miss了16次

> 总共访问了几次L2 Cache

16次

> 哪个参数可以增加L2 Cache Hit Rate，且保持L1 Cache Hit Rate不变

增大Rep Count

> 将L1 Cache中的块数增加，L1.2 Cache的Hit Rate的变化

L1的增大，L2的减小，前提是Rep Count > 1

> 将L1 Cache中的块大小增加，L1.2 Cache的Hit Rate的变化

L1的增大，L2的减小，前提是Rep Count > 1

Exercise 2: Loop Ordering and Matrix Multiplication

> 1000*1000的矩阵相乘，哪种嵌套顺序性能最好？哪种嵌套顺序性能最差？

kij的最好，jki的最差

> 修改代码，性能改善了吗

有一些原先比较慢点代码变快了，但是原先很快的代码反而变慢

经验就是对于一些难以优化的代码，就设置一个临时变量，在Cache的固定位置持续访问，用空间换时间

> 为什么Cache的命中率只差了4倍，运行效率却差了30倍

- 最优循环顺序的cache miss大多发生在访问连续地址时
- 最差循环顺序的cache miss往往在访问跨度较大的地址时
- 连续失效可以被硬件预取器预测和提前加载
- 随机失效则无法被预测,每次都要付出完整的访存延迟

因此

- 好的版本: 大部分miss被预取掩盖，实际延迟很小
- 差的版本: 每次miss都付出完整延迟，且会阻塞后续操作

Exercise 3: Cache Blocking and Matrix Transposition

`transpose_blocking()`实现如下

```
void transpose_blocking(int n, int blocksize, int *dst, int *src) {
    int row = n / blocksize;
    int col = n / blocksize;

    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            for (int k = 0; k < blocksize; k++)
                for (int m = 0; m < blocksize; m++)
                    dst[(j * blocksize + m) * n + (i * blocksize + k)] = src[(i
* blocksize + k) * n + (j * blocksize + m)];

    for (int i = 0; i < row * blocksize; i++) {
        for (int j = col * blocksize; j < n; j++) {
            dst[j * n + i] = src[i * n + j];
        }
    }

    for (int j = 0; j < col * blocksize; j++) {
        for (int i = row * blocksize; i < n; i++) {
            dst[j * n + i] = src[i * n + j];
        }
    }

    for (int i = row * blocksize; i < n; i++) {
        for (int j = col * blocksize; j < n; j++) {
            dst[j * n + i] = src[i * n + j];
        }
    }
}
```

3.1 改变矩阵的大小

`n=100`

> Testing naive transpose: 0.007 milliseconds
> Testing transpose with blocking: 0.005 milliseconds

`n=500`

> Testing naive transpose: 0.504 milliseconds
> Testing transpose with blocking: 0.181 milliseconds

`n=1000`

> Testing naive transpose: 2.226 milliseconds
> Testing transpose with blocking: 4.124 milliseconds

`n=2000`

> Testing naive transpose: 7.757 milliseconds
> Testing transpose with blocking: 17.68 milliseconds

`n=5000`

> Testing naive transpose: 166.506 milliseconds
> Testing transpose with blocking: 82.948 milliseconds

`n=10000`

> Testing naive transpose: 857.732 milliseconds
> Testing transpose with blocking: 302.016 milliseconds

> 为什么矩阵大小要到一定程度?

当矩阵很小时，整个矩阵都能放到Cache里，分块没有意义，反而加大了算法的复杂度

当矩阵慢慢变大，分块的优势才显示出来

3.2 改变分块大小

`b=50`

> Testing naive transpose: 790.462 milliseconds
> Testing transpose with blocking: 146.988 milliseconds

`b=100`

> Testing naive transpose: 814.614 milliseconds
> Testing transpose with blocking: 162.93 milliseconds

`b=200`

> Testing naive transpose: 815.227 milliseconds
> Testing transpose with blocking: 150.127 milliseconds

`b=500`

> Testing naive transpose: 798.285 milliseconds
> Testing transpose with blocking: 145.151 milliseconds

`b=1000`

> Testing naive transpose: 819.382 milliseconds
> Testing transpose with blocking: 186.604 milliseconds

`b=5000`

> Testing naive transpose: 798.76 milliseconds
> Testing transpose with blocking: 521.625 milliseconds

当`blocksize`变大时，程序有变慢的趋势，相当于Cache里也放不下一个block了

Exercise 4: Memory Mountain

4.1 运行结果

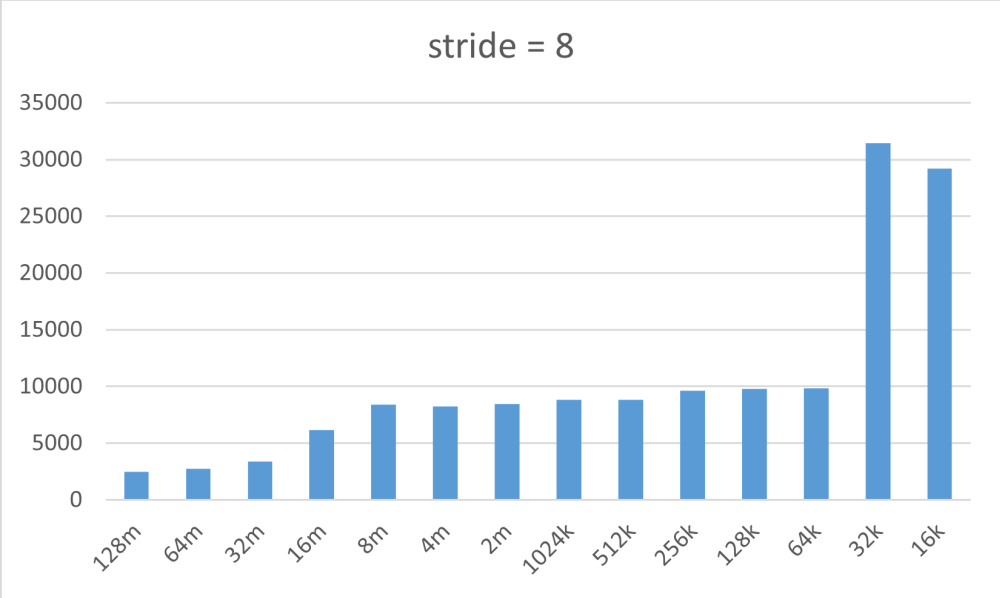
运行的结果如下

> Clock frequency is approx. 3293.7 MHz Memory mountain (MB/sec)

Block Size	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
128m	14643	8691	6599	5057	3945	3255	2613	2459	2151	1717	1620	1492	1492	1442	1386
64m	14818	8329	6803	5415	4417	3720	3132	2747	2296	1985	1607	1912	1920	1948	1990
32m	16874	9622	7669	6073	5175	4386	3846	3374	2928	2764	2640	2893	3076	3192	3616
16m	22716	15850	11218	9662	8794	6409	6741	6143	6034	6049	5207	6185	6238	6215	5884
8m	35197	26432	20317	15826	13082	11099	9380	8396	7878	8663	8664	8680	8652	8630	8645
4m	33156	25026	19569	15261	13422	10939	9339	8230	7852	7337	6940	6731	6500	6598	6637
2m	32074	24441	18878	15243	13223	10977	9445	8427	7916	7399	6909	6192	5955	5985	5948
1024k	35891	24660	16432	13241	10685	11453	9869	8798	8500	8787	7539	7404	7312	7597	7822
512k	32788	25652	20330	16171	13592	11491	9954	8804	8639	8399	8495	8792	8886	9050	9253
256k	33892	28317	24499	19070	15301	12826	10961	9619	9626	9690	9708	9777	9770	9683	9745
128k	34157	29202	25803	19584	15762	13056	11258	9792	9821	9836	9828	9821	9963	9941	9911
64k	33892	29202	25651	19239	15762	13294	11258	9851	9955	9910	10431	10482	12579	17304	27254
32k	37593	36340	35165	34068	34424	32062	31147	31448	33028	36331	27021	34060	35932	29194	31141
16k	36340	34068	34066	31448	32698	27248	29194	29202	30276	23355	24769	22707	31440	19463	27248

4.2 固定步长大小

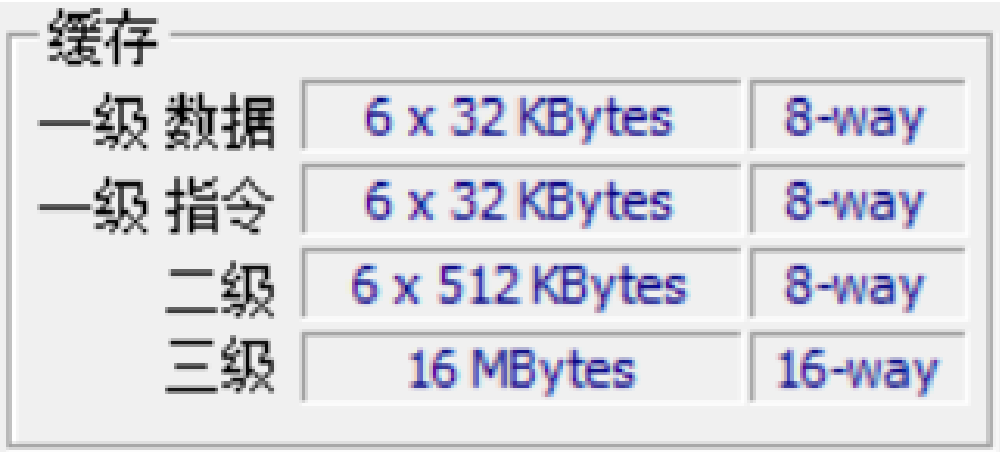
绘制成柱状图如下



`128M - 32M`是主存，`16M -512k`是三级缓存，`256k - 64k`是二级缓存，`32k - 16k`是一级缓存，三级缓存的容量是`16M`

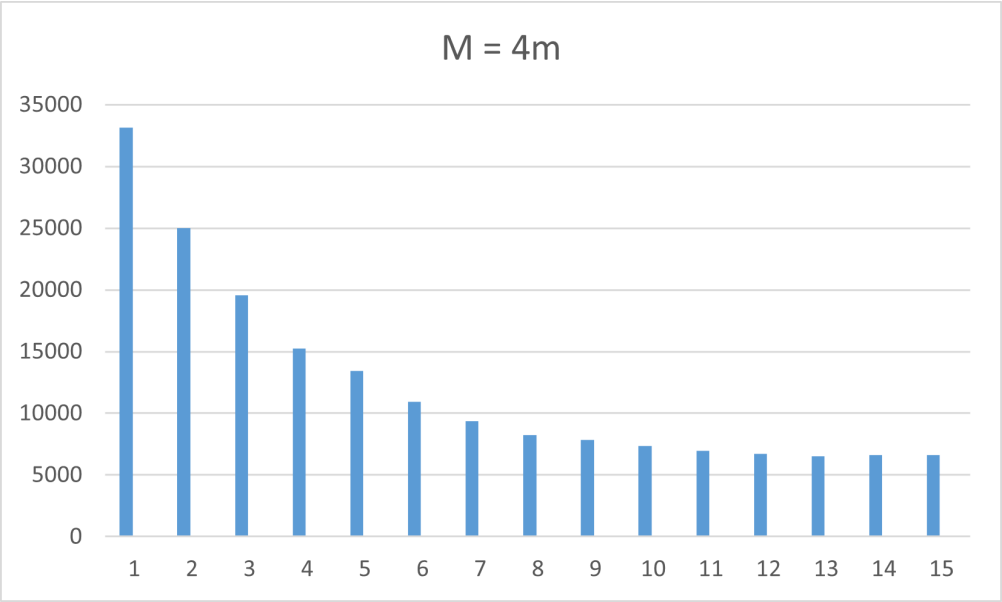
4.3 系统配置对比

系统配置如下



我的判断基本一致，但是`512k`应该是在二级缓存内

4.4 固定工作集大小



看起来一块的大小应该是`8*8=64B`，实际上使用`getconf -a | grep CACHE`也能得到

```
LEVEL1_ICACHE_SIZE         32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE     64
LEVEL1_DCACHE_SIZE         32768
LEVEL1_DCACHE_ASSOC        8
LEVEL1_DCACHE_LINESIZE     64
LEVEL2_CACHE_SIZE          524288
LEVEL2_CACHE_ASSOC         8
LEVEL2_CACHE_LINESIZE      64
LEVEL3_CACHE_SIZE          16777216
LEVEL3_CACHE_ASSOC         16
LEVEL3_CACHE_LINESIZE      64
```

可以发现数据块的大小是`64B`

看上面的图，从`stride = 8`开始，吞吐量变化几乎就没了，说明此时开始每次Cache都miss了，说明一块就是`8 * 8 = 64Byte`