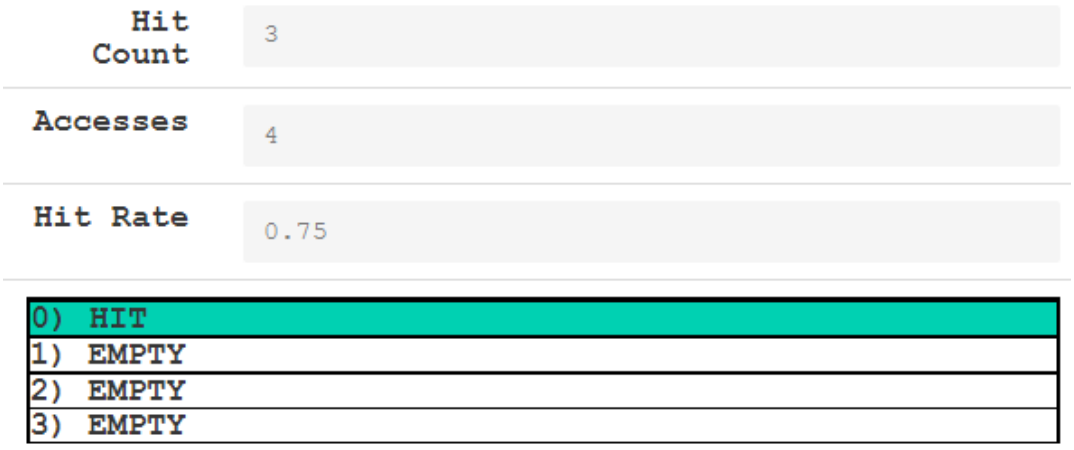
hw6

郑航 520021911347

Exercise 1: Cache Visualization

场景1

- cache hit rate：0
- 因为每个Rep周期里边，需要读取的数据为128/4/8=4个，而cache的总块数为4，但是由于采用direct mapped的方式，地址都相差4 的倍数，因此都会映射到cache中的同一个项中，即cache中永远只能保留两个相邻的数据（每个cache数据块为8bytes，可以放2个数据），因此若数据数大于1则会导致每次都为cache miss。本实验中数据数为4，因此cache hit rate为0
- 增加 Rep Count 参数的值，不可以提高命中率。理由如上面所阐述，每个周期中待访问的数据都会被后边的数据所替换，增加循环的周期数并不能提高命中率
- 最大化hit rate，就要令cache中不发生替换，有两个思路：
  - 每个周期中访问的数据量等于1，128/4/1=32，因此将Step Size 修改为大于等于32即可最大化hit rate，经测试hit rate为0.75（首个周期miss，其后的周期都会是hit，四个周期则3/4=0.75）



  - 修改Step Size，使得数据地址相差不为4的倍数，可以被映射到4个不同的cache块中，同时每周期的数据不超过4个，才不会发生替换，一个可能的取值是令Step Size=10，则每周期读4个数据且能放到4个cache块中，此时hit rate为0.75（首个周期miss，其后的周期都会是hit，四个周期则3/4=0.75），如下图：

Hit Count	12
Accesses	16
Hit Rate	0.75

```
0) HIT
1) HIT
2) HIT
3) HIT
```

场景2

- cache hit rate：0.75
- 一个周期里面读取的数据为256/4/2=32个，而cache的总块数为16，采用N路组相联映射，关联度为4，则有4个组，每个组中有4块，每块大小为16bytes，则可以放4个相邻的数据；step size为2，因此访问的数据可以每两个为1组放在一个cache块中（4/2=2），共32个数据，16块，因此恰好可以每个数据都得到缓存，而每个数据需要读和写各访问一次，每两个数据为一组中，只有第一个数据的第一次访问是cache miss的，其后第一个数据的第二次访问以及第二个数据的两次访问都是cache hit，因此总的cache hit rate=48/64=0.75
- 增加 Rep Count 参数的值，cache hit rate会增加，假如重复无限次，则cache hit rate≈1，理由：如上所言，一个周期里面，恰好可以对所有的数据进行缓存，因此随着周期数增加，后面周期内的所有数据访问都是cache hit的，因此hit rate会逐渐增加并趋近于1

场景3

- L1 cache的命中率为0.5，L2 cache的命中率为0
- 总共访问了 L1 cache 32次， L1 Miss 次数为16
- 总共访问了 L2 cache 16次
- 程序参数a2（重复周期数）可以增加 L2 hit rate, 并且保持 L1 hit rate 不变
- 如果将 L1 cache 中的块数增加， L1 hit rate 不变， L2 hit rate 也不变
- 如果将 L1 cache 中的块大小增加，L1 hit rate 增大， L2 hit rate 不变，依旧为0

Exercise 2: Loop Ordering and Matrix Multiplication

- kij和ikj嵌套顺序性能最好，jki和kji嵌套顺序性能最差
- 程序跑出的性能：kij/ikj > ijk/jik > jki/kji，而未命中总次数：kij/ikj < ijk/jik < jki/kji

  性能与未命中次数呈负相关，因此结论与结果一致

  最内层循环中数据访问的步长越长，则cache未命中总次数越多，继而导致性能越差
- 程序性能如下：

```
zh@ubuntu:~/arch-homework/hw06$ make ex2
gcc -o matrixMultiply_init -ggdb -Wall -pedantic -std=gnu99 -O3 matrixMultiply_init.c
./matrixMultiply_init
ijk:    n = 1000, 0.982 Gflop/s
ikj:    n = 1000, 7.213 Gflop/s
jik:    n = 1000, 1.231 Gflop/s
jki:    n = 1000, 0.092 Gflop/s
kij:    n = 1000, 6.157 Gflop/s
kji:    n = 1000, 0.101 Gflop/s
```

```
zh@ubuntu:~/arch-homework/hw06$ make ex2
gcc -o matrixMultiply -ggdb -Wall -pedantic -std=gnu99 -O3 matrixMultiply.c
./matrixMultiply
ijk:    n = 1000, 1.053 Gflop/s
ikj:    n = 1000, 2.221 Gflop/s
jik:    n = 1000, 1.132 Gflop/s
jki:    n = 1000, 0.078 Gflop/s
kij:    n = 1000, 1.837 Gflop/s
kji:    n = 1000, 0.092 Gflop/s
```

可见，程序性能没有提升，且ikj和kij两种方式的性能反而下降了

经验：不要在程序中增加无用的多余的变量，反而会增加访存的开销

- 为什么实际性能差距这么大：

  因为cache miss时需要到内存中访问数据，而内存访问的所需时间大约是cache的100倍左右，因此少量的cache miss增加就会导致访问时间的大量增加，运算性能与cache失效率并不是成正比的关系。

Exercise 3: Cache Blocking and Matrix Transposition

程序正确性验证：

```
zh@ubuntu:~/arch-homework/hw06$ make ex3
gcc -o transpose -ggdb -Wall -pedantic -std=gnu99 -O3 transpose.c
zh@ubuntu:~/arch-homework/hw06$ ./transpose 1000 33
Testing naive transpose: 1.817 milliseconds
Testing transpose with blocking: 1.884 milliseconds
```

Part 1: 改变矩阵的大小

```
zh@ubuntu:~/arch-homework/hw06$ ./transpose 100 20
Testing naive transpose: 0.011 milliseconds
Testing transpose with blocking: 0.017 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 500 20
Testing naive transpose: 0.574 milliseconds
Testing transpose with blocking: 0.608 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 1000 20
Testing naive transpose: 2.608 milliseconds
Testing transpose with blocking: 2.2 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 2000 20
Testing naive transpose: 47.039 milliseconds
Testing transpose with blocking: 7.665 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 5000 20
Testing naive transpose: 459.582 milliseconds
Testing transpose with blocking: 52.466 milliseconds
```

可见，当n较小的时候，矩阵分块实现矩阵转置比不用矩阵分块的方法甚至还要慢一些；当n较大的时候，矩阵分块实现矩阵转置比不用矩阵分块的方法显著要快

为什么矩阵大小要达到一定程度，矩阵分块算法才有效果 ：当矩阵大小太小的时候，无论是否采用分块，cache都足以容纳整个矩阵的数据，两者的cache失效率差距不大，且采用分块的方式，实现起来还要更为复杂一些，增加了一些变量的访存等过程，由于矩阵较小，这部分的开销就无法忽略了，因此分块的方式甚至速度还要慢一点。当矩阵大小足够大之后，cache无法容纳整个矩阵，此时两种方式的cache失效率就会有显著的区别，同时额外的变量访存时间此时也可以忽略，故分块的方式速度显著快于不分块的方式

Part 2: 改变分块大小

```
zh@ubuntu:~/arch-homework/hw06$ ./transpose 10000 50
Testing naive transpose: 3008.86 milliseconds
Testing transpose with blocking: 255.828 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 10000 100
Testing naive transpose: 2152.17 milliseconds
Testing transpose with blocking: 227.619 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 10000 200
Testing naive transpose: 2202.24 milliseconds
Testing transpose with blocking: 199.989 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 10000 500
Testing naive transpose: 2188.51 milliseconds
Testing transpose with blocking: 185.898 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 10000 1000
Testing naive transpose: 2363.96 milliseconds
Testing transpose with blocking: 331.386 milliseconds
zh@ubuntu:~/arch-homework/hw06$ ./transpose 10000 5000
Testing naive transpose: 2348.63 milliseconds
Testing transpose with blocking: 1938.55 milliseconds
```

可见，当blocksize增加时，性能先逐渐提升再逐渐下降，原因是当blocksize较小时（此处是小于500），块的大小较小，分块过程中cache可以容纳整个块，cache miss rate差距不大，且随着blocksize增加，减少了诸如循环变量访存以及for循环等的开销，因此性能逐渐提升；随着blocksize逐渐变大，cache无法容纳整个块的数据，cache miss rate会逐渐增大，分块的优势逐渐减少，因此性能也逐渐下降

Exercise 4: Memory Mountain

程序运行结果：

```
zh@ubuntu:~/arch-homework/hw06/mountain$ ./mountain
Clock frequency is approx. 2112.0 MHz
Memory mountain (MB/sec)
       s1      s2      s3      s4      s5      s6      s7      s8      s9      s10     s11     s12     s13     s14     s15
128m   9618    5095    3468    2536    2540    2215    1705    1660    1569    1242    1121    1193    1440    1373    876
64m    12558   7067    4353    3481    2683    2036    2092    1479    1450    1326    1216    1129    1084    1012    1056
32m    13027   5992    5049    3467    2025    2615    1638    2180    1427    1328    1197    1135    1135    1163    1052
16m    13375   7140    4277    3459    2694    2928    1964    2014    605     1339    1356    1308    1329    1322    1137
8m     10304   6726    5574    4597    2777    2578    1869    1762    2065    2261    1769    1460    1705    1354    1713
4m     9170    12423   9509    7913    6586    5665    4995    4423    4257    4060    3860    3677    3538    3488    3405
2m     17816   12864   10558   8918    7489    6424    5624    4945    4648    4332    4853    4579    4212    4070    3967
1024k  19384   14099   11556   10167   8510    6996    6118    6430    6412    6386    6207    6013    5839    5718    5643
512k   23610   19801   16568   13422   11185   9608    8411    7407    7193    7049    6852    6757    6651    7019    7020
256k   26746   22803   20675   18549   16034   14222   12455   11246   11001   10916   11106   11286   10982   11576   11556
128k   27866   25075   24541   22720   19844   17423   15375   13506   13223   16636   13189   12064   13275   12101   12046
64k    22061   23492   22795   21873   19944   18106   15618   14275   13098   13703   11824   11397   10953   11940   19887
32k    25939   25670   25128   25148   24366   24129   23539   23508   23875   23858   23468   23438   23140   23094   22391
16k    25594   25518   24750   24164   23375   23249   23094   22296   22348   21094   20689   20302   20160   19305   20592
```

固定步长

stride=8 时，不同工作集大小情况下的读吞吐率情况：

size	128M	64M	32M	16M	8M	4M	2M	1024K	512K	256K	128K	64K	32K	16K
吞吐率	1660	1479	2180	2014	1762	4423	4945	6430	7407	11246	13506	14275	23508	22296

根据数据，一级高速缓存大小应该为32K，二级高速缓存大小应该为256K，有三级高速缓存，三级高速缓存大小应该为4M

系统高速缓存配置：

```
zh@ubuntu:~/arch-homework/hw06/mountain$ getconf -a | grep CACHE
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                  6291456
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

可见，L1和L2的cache size与我们通过运行数据观察到的结论是一致的，而L3的实际大小为6M，由于测试中没有6M的工作集大小，因此与我们观察到的大小4M有所差距，但这并不影响测试方式和观察方式的正确性，假如增加一个6M大小的工作集，应该就可以看到更准确更明显的测试结果了

固定工作集大小

工作集大小=4MB时，不同步长情况下的读吞吐率情况：

stride	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
吞吐率	9170	12423	9509	7913	6586	5665	4995	4423	4257	4060	3860	3677	3538	3488	3405

高速缓存的块大小应为64bytes，因为long long int的大小为8bytes，当stride为8之后，吞吐率曲线逐渐变得平稳，说明在这之后，每次访问数据所需的时间相差不多，都会是cache miss的情况，故cache中一个块只能容纳8个数据，即blocksize为8*8bytes=64bytes