实践4

叶增渝 519030910168

Exercise 1: Cache Visualization

场景1：

1.Cache命中率为0

2.因为每次访问步长为8word，并重复4次，故顺序访问了arr[0]、arr[8]、arr[16]、arr[24] 4次，由于采取直接映射，而一个block存放2个word，Cache中总共有4块，上述4个word对应的block它们的tag 都对应Cache中的第0块，所以会不断替换，没有一次能hit。

3.不会有改善。

4.由于每个block仅含2个word，所以在数组长度大于2word时，如果步长大于2且一次至少访问4个word，总是会产生替换，无法hit，所以当步长为1时能保证至少50%hit，在此基础上，如果array的block数小与等于4，那么一次访问将会将array中所有block存于Cache中，后面重复时不再会出现miss，所以重复次数越高，hit rate越高，而此时若option改为RW，那么hit总次数增加，hit rate也会上升。

综上，当步长为1，array大小在1-8word（4-32byte）时，取option1，重复次数越高，hit rate越高，可接近100%但无法达到。

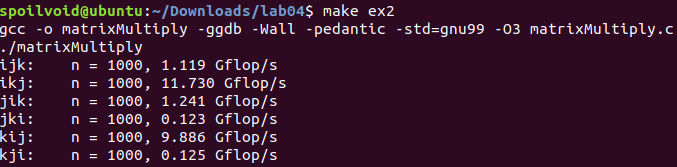
场景2：  
1.Cache命中率为75%。

2.Array的大小为256Bytes，所以共有64words，一个block含有4个word，数组分为16个block，Cache采取4路组相联映射，有4行，刚好能够在一次访问完后装满Cache，而访问步长为2，且采取RW策略，所以在读该block 的第一个data时miss，将整个block装入Cache，后面紧跟的write与2步长后的read和write都能成功hit，每个block均如此，所以Cache命中率为75%。

3.增加重复次数，命中率会不断上升，若无限次，则命中率无限接近100%，因为刚好能够在一次访问完后装满Cache，在接下来的重复中，所有数据都已经在Cache，不会存在miss的情况，所以命中率不断上升。

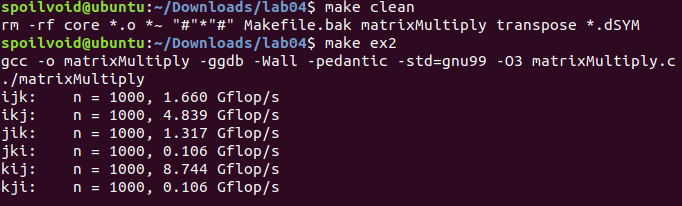
Exercise 2: Loop Ordering and Matrix Multiplication

1. 从浮点运算吞吐率 Gflops/s 来衡量不同实现方式的性能可以看出，kij&ikj嵌套方式的性能最好，jki&kji嵌套方式的性能最差。



2.与观察到的结果一致。如果在同行移动列的部分，如果步长过长，会导致载入一个block后后面数据的利用率下降，导致Cache命中率下降，从而使性能下降，但对同列按行移动则没有什么影响，因为一直无法hit。

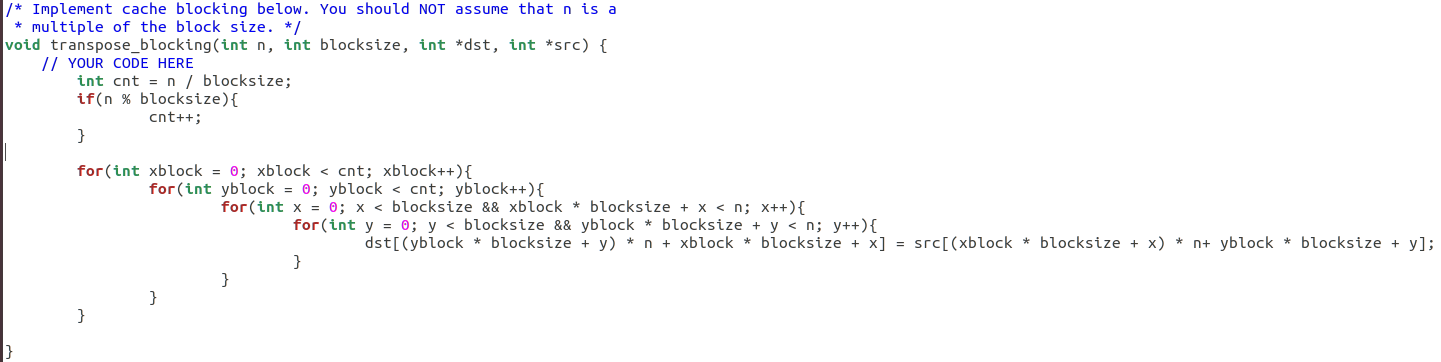
3.可以看到在修改代码后除ikj的浮点运算吞吐率 Gflops/s下降至第二外，性能并没有什么明显的改善。说明简单的固定一个矩阵中的数，对于最内层循环的帮助不大，可见在Cache中多次hit并不是导致程序性能下降的主要原因，这种改动没有办法很好地增加命中率，提高程序性能。



4.我认为可能有以下原因：（1）该处理器为流水线处理器，当在Cache中miss后，流水线会产生停顿等影响，使得tw阶段延长，影响流水线性能，从而导致连锁反应；（2）当在Cache中miss后，需要在多级缓存中一级机向上寻找，甚至到主存中寻找再一级级向下传递，在各级中寻找到数据对运行时间的影响是不一样的，从而导致了这个结果；（3）像最快的kij&ikj中按行取，处理器对空间局部性做出诸如prefetch等调整，增加了其性能，导致差距变大，而最坏算法均按列取，难以优化。

Exercise 3: Cache Blocking and Matrix Transposition

transpose.c中修改后的transpose\_blocking函数源代码：



void transpose\_blocking(int n, int blocksize, int \*dst, int \*src) {

// YOUR CODE HERE

int cnt = n / blocksize;

if(n % blocksize){

cnt++;

}

for(int xblock = 0; xblock < cnt; xblock++){

for(int yblock = 0; yblock < cnt; yblock++){

for(int x = 0; x < blocksize && xblock \* blocksize + x < n; x++){

for(int y = 0; y < blocksize && yblock \* blocksize + y < n; y++){

dst[(yblock \* blocksize + y) \* n + xblock \* blocksize + x] = src[(xblock \* blocksize + x) \* n+ yblock \* blocksize + y];

}

}

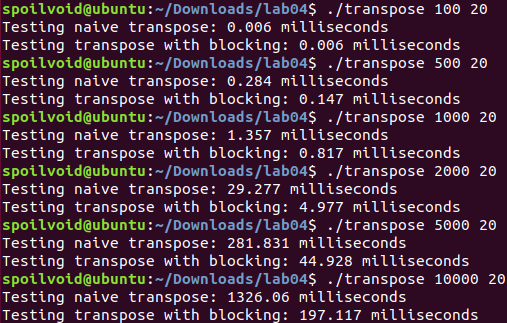
}

}

}

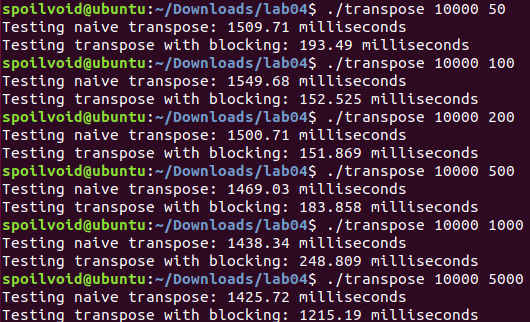


Part1：矩阵分块实现矩阵转置比不用矩阵分块的方法快。矩阵大小要达到一定程度， 矩阵分块算法才有效果，这是因为当矩阵规模很小时，虽然分块算法更优秀，但是整体所需时间与矩阵所含block数很少，本身缓存失效次数很少，所以几乎没有区别，而当数组规模增大时，分块利用的空间与时间局部性减少了大量缓存失效次数，所以产生了明显的效果，性能更好。



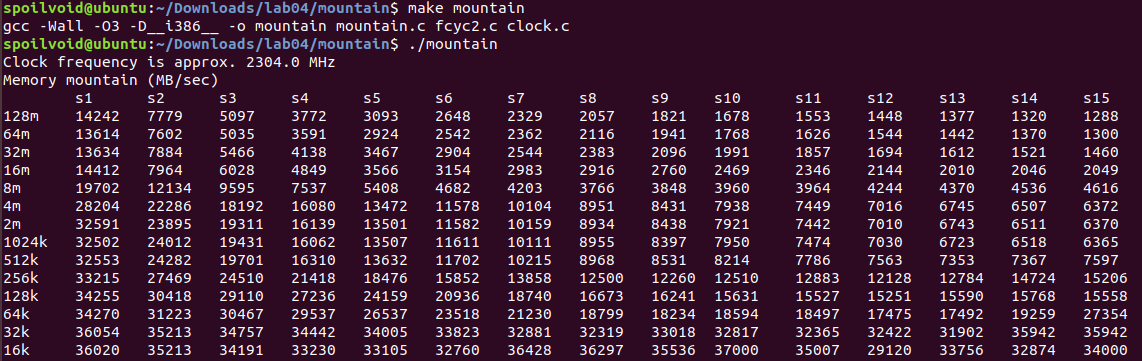
Part2：

当 blocksize 增加时性能呈现先提升后迅速下降的变化趋势。因为当block规模过小时，由于规模导致命中率受限，一次miss只能导致少量hit，所以效率不高，当block规模过大时，这种算法与普通的算法几乎没有任何区别，所以效率不高，当blocksize约为时效率最高。



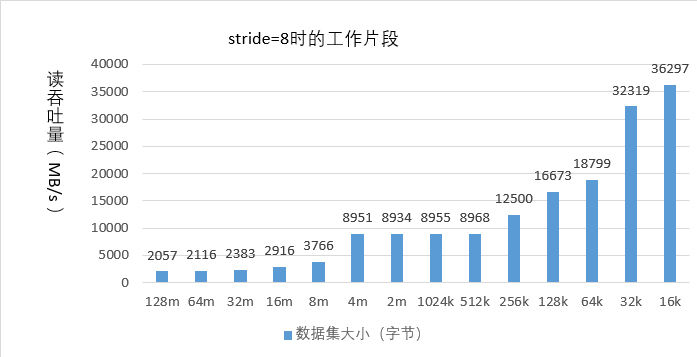
Exercise 4: Memory Mountain

1.运行结果：

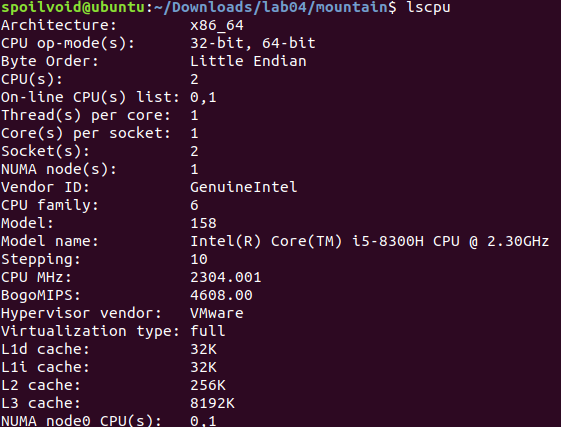
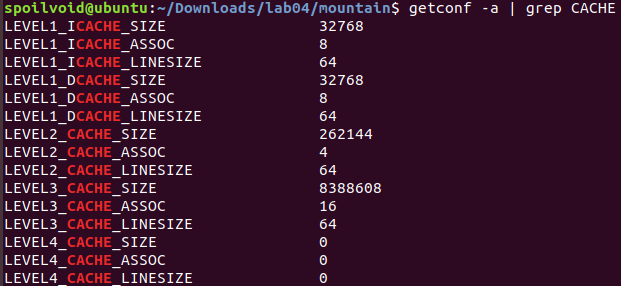


|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 | s10 | s11 | s12 | s13 | s14 | s15 |
| 128m | 14242 | 7779 | 5097 | 3772 | 3093 | 2648 | 2329 | 2057 | 1821 | 1678 | 1553 | 1448 | 1377 | 1320 | 1288 |
| 64m | 13614 | 7602 | 5035 | 3591 | 2924 | 2542 | 2362 | 2116 | 1941 | 1768 | 1626 | 1544 | 1442 | 1370 | 1300 |
| 32m | 13634 | 7884 | 5466 | 4138 | 3467 | 2904 | 2544 | 2383 | 2096 | 1991 | 1857 | 1694 | 1612 | 1521 | 1460 |
| 16m | 14412 | 7964 | 6028 | 4849 | 3566 | 3154 | 2983 | 2916 | 2760 | 2469 | 2346 | 2144 | 2010 | 2046 | 2049 |
| 8m | 19702 | 12134 | 9595 | 7537 | 5408 | 4682 | 4203 | 3766 | 3848 | 3960 | 3964 | 4244 | 4370 | 4536 | 4616 |
| 4m | 28204 | 22286 | 18192 | 16080 | 13472 | 11578 | 10104 | 8951 | 8431 | 7938 | 7449 | 7016 | 6745 | 6507 | 6372 |
| 2m | 32591 | 23895 | 19311 | 16139 | 13501 | 11582 | 10159 | 8934 | 8438 | 7921 | 7442 | 7010 | 6743 | 6511 | 6370 |
| 1024k | 32502 | 24012 | 19431 | 16062 | 13507 | 11611 | 10111 | 8955 | 8397 | 7950 | 7474 | 7030 | 6723 | 6518 | 6365 |
| 512k | 32553 | 24282 | 19701 | 16310 | 13632 | 11702 | 10215 | 8968 | 8531 | 8214 | 7786 | 7563 | 7353 | 7367 | 7597 |
| 256k | 33215 | 27469 | 24510 | 21418 | 18476 | 15852 | 13858 | 12500 | 12260 | 12510 | 12883 | 12128 | 12784 | 14724 | 15206 |
| 128k | 34255 | 30418 | 29110 | 27236 | 24159 | 20936 | 18740 | 16673 | 16241 | 15631 | 15527 | 15251 | 15590 | 15768 | 15558 |
| 64k | 34270 | 31223 | 30467 | 29537 | 26537 | 23518 | 21230 | 18799 | 18234 | 18594 | 18497 | 17475 | 17492 | 19259 | 27354 |
| 32k | 36054 | 35213 | 34757 | 34442 | 34005 | 33823 | 32881 | 32319 | 33018 | 32817 | 32365 | 32422 | 31902 | 35942 | 35942 |
| 16k | 36020 | 35213 | 34191 | 33230 | 33105 | 32760 | 36428 | 36297 | 35536 | 37000 | 35007 | 29120 | 33756 | 32874 | 34000 |

2.从stride=8的图表可以看出，可以按照数据集大小分为4个部分，16m-128m为主存部分，512k-8m为3级缓存部分，64k-256k为2级缓存部分，16k-32k为1级缓存部分。据此推测，程序运行所在的系统，一级高速缓存为32KB，二级高速缓存为256KB，存在三级高速缓存，容量为8MB。



3.判断与系统配置一致。截图如下：



4.long long int为标准的64位整数，占据8个字节。Test函数为4\*4展开，即以4stride移动，每次访问偏移为0，stride，2stride，3stride的数字，因此，而在s8后基本相平，说明此时每个高速缓存行一次访问，因此blocksize应为8x8(字节)=64Byte。

