实践4 实验报告

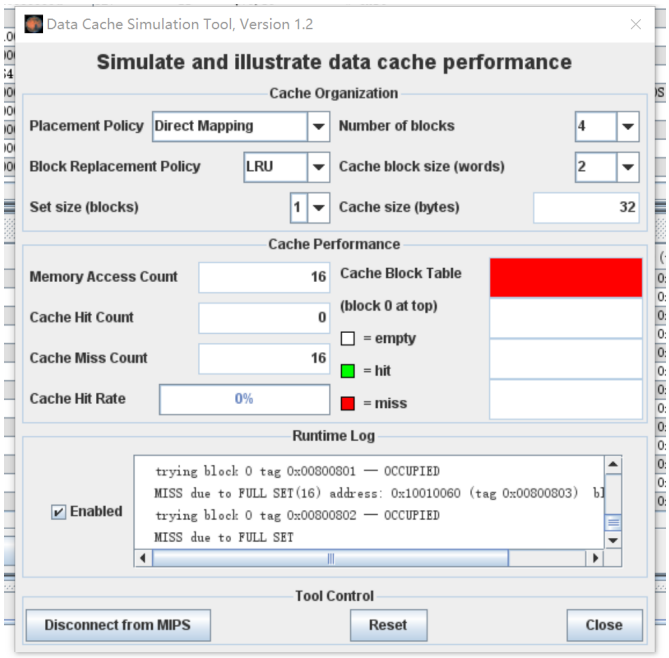
陈文迪 519021910071

Exercise 1: Cache Visualization

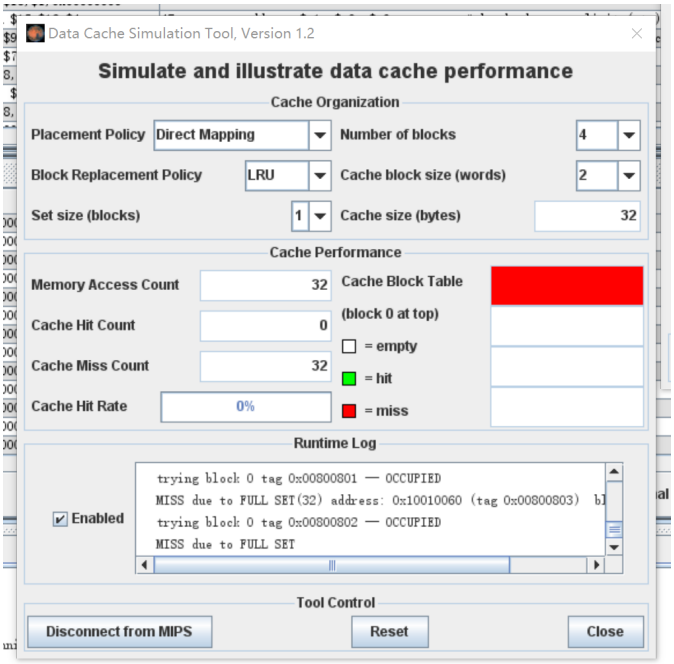
场景1：（使用cache.s）

问题回答：

1. cache命中率为0%。

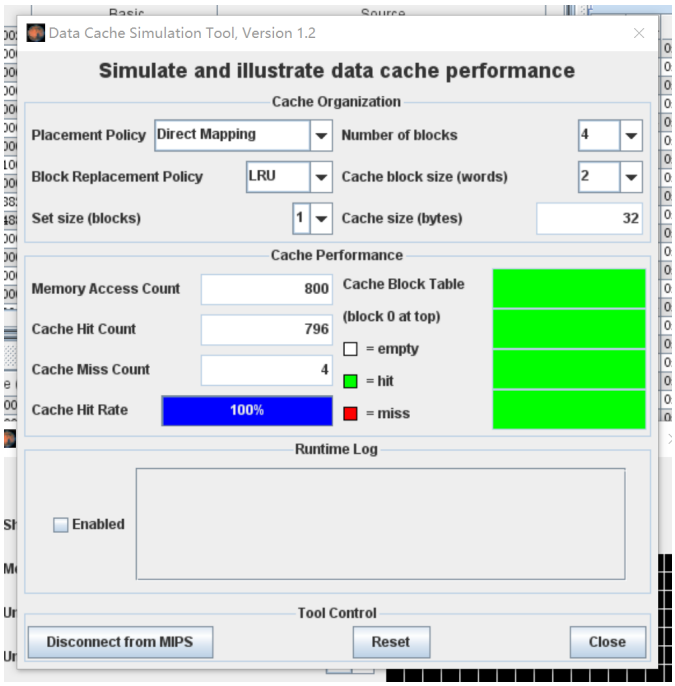


2. 在这种设置下，我们每次会访问一个字长（4个字节）的数据，而访问的跨度为8个字（32个字节）。因此，在整个过程中，我们会访问整个数组中4个位置的元素（大小均为一个字）。对于cache来说，由于每个缓存块的大小为2个字，而一共有四个缓存块，缓存总大小恰好为8个字，与访问跨度相等，这就导致每次访问的数据块其对应的缓存块其实是同一个。我们的程序每次循环仅仅会产生一次缓冲访问，而该缓存块已经被上一次访问的数据所占据，导致cache miss。当我们将其替换成新的数据块后，又会导致下一次的cache miss。这就是为什么cache命中率为0。
3. 不能。因为所有需要访问的数据对应的缓存块仍然是同一个，每次缓存块所保存的数据始终是上一次访问的数据块。我们从可以将rep count改为8，可以从结果中验证我们的结论。



4. 当我们这样设置program parameters时，可以让hit rate最大化。

```
main:   li  $a0, 32     # array size in BYTES (power of 2 < array size)
        li  $a1, 2      # step size  (power of 2 > 0)
        li  $a2, 100    # rep count  (int > 0)
        li  $a3, 1      # 0 - option 0, 1 - option 1
```

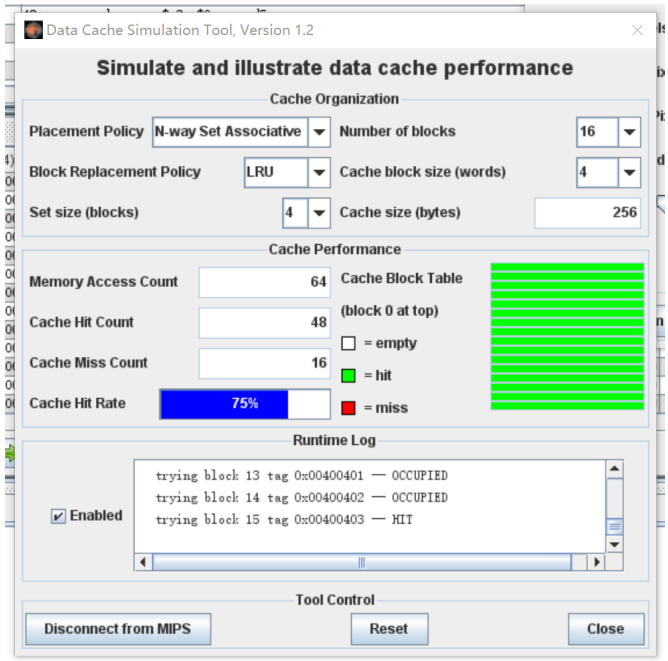


此时，我们两次相邻访问间的跨度为2个字，并且每次循环我们进行一次cache读一次cache写，写的时候必然cache hit。这种跨度保证了每次相邻访问的数据位于不同的缓存块。此外，我们将数组大小设置为32字节，与缓存的总容量一样。这样经过一次遍历之后，所有的数据都会被缓存到cache中，后续的全部访问都会cache hit。这样重复100次后，hit rate接近100%。

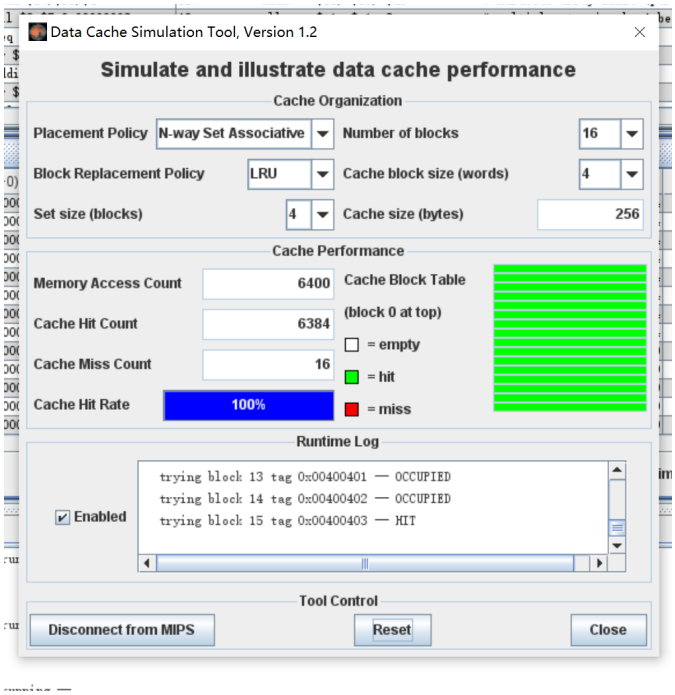
场景2：（使用cache.s）

问题回答：

1. 缓存命中率为75%。



2. 因为在这种条件下，两次循环的间隔为2个字，而一个缓存块的大小为4个字，因此两次相邻的循环只需要一次cache miss。由于每次循环会访问两次cache，因此$\text{cache hit rate} = \frac{4-1}{4} = 75\%$。
3. 由于我们采用的是4路组相联，cache总大小为256bytes，恰好覆盖了整个数组的空间，每个需要访问的数据块都可以被映射到一个cache块中而不需要替换。因此，经过一次rep之后，所有的数据都可以被保存在cache中，后续的所有访问都是cache hit。因此重复无限次后，命中率为100%。



Exercise 2: Loop Ordering and Matrix Multiplication

问题回答：

- ```
  PS C:\Users\12779\Desktop\lab04> .\matrixMultiply
  ijk:    n = 1000, 2.040 Gflop/s
  ikj:    n = 1000, 11.022 Gflop/s
  jik:    n = 1000, 2.044 Gflop/s
  jki:    n = 1000, 1.758 Gflop/s
  kij:    n = 1000, 10.255 Gflop/s
  kji:    n = 1000, 1.790 Gflop/s
  ```
  ```

从测试结果上可以看到 kij&ikj具有最好的性能，而jki&kji具有最差的性能。

- 和我的观察一致。当我们以步长0访问数组元素的时候，由于我们所访问的始终是同一个地址的数据，缓存命中率为100%；当我们以步长1访问数组元素的时候，我们的缓存命中率为$1-(\frac{数组元素大小}{缓存块大小})$；而当我们以步长为n访问数组元素时，由于通常由$n >> 缓存块大小$，此时的缓存命中率为0。由于我们总的循环次数是一样的，因此每次循环迭代的平均缓存命中率会极大地影响程序的总体性能。可以看到，kij&ikj之所以具有最好的性能，正是因为其具有最低的未命中总次数即最高的缓存命中率。
- ```
  PS C:\Users\12779\Desktop\lab04> .\matrixMultiply.exe
  ijk:    n = 1000, 2.079 Gflop/s
  ikj:    n = 1000, 10.948 Gflop/s
  jik:    n = 1000, 2.095 Gflop/s
  jki:    n = 1000, 1.766 Gflop/s
  kij:    n = 1000, 10.279 Gflop/s
  kji:    n = 1000, 1.831 Gflop/s
  ```

  可以看到性能几乎没有改善，这说明经过高级别编译器优化（例如使用寄存器变量加速变量读写）和硬件优化，程序的时间局部性、空间局部性已经得到了较好的利用，不需要在进行人为的优化。

  我们可以在"-O0"的情况下进行再次实验。先测试未优化版本。

  ```
  PS C:\Users\12779\Downloads\lab04> .\matrixMultiply.exe
  ijk:    n = 1000, 0.573 Gflop/s
  ikj:    n = 1000, 0.734 Gflop/s
  jik:    n = 1000, 0.586 Gflop/s
  jki:    n = 1000, 0.549 Gflop/s
  kij:    n = 1000, 0.737 Gflop/s
  kji:    n = 1000, 0.559 Gflop/s
  ```

  再测试优化版本。

  ```
  PS C:\Users\12779\Desktop\lab04> .\matrixMultiply.exe
  ijk:    n = 1000, 1.404 Gflop/s
  ikj:    n = 1000, 1.243 Gflop/s
  jik:    n = 1000, 1.536 Gflop/s
  jki:    n = 1000, 0.966 Gflop/s
  kij:    n = 1000, 1.245 Gflop/s
  kji:    n = 1000, 0.926 Gflop/s
  ```

  可以看到在低级别编译器优化的情况下，人为的优化对于效率还是有所帮助的，如使用寄存器变量来代替访问cache以加速。
- 第一点，当我们顺序访问数组时，硬件可以采用硬件预取的方式获得额外的优化；而当步长很大时，硬件预取失效。第二点，上文分析时，我们假设缓存块大小是数组元素大小的4倍，而实际元器件中缓存块大小会更大，可以存储更多的数组元素。这就导致对于 kij&ikj，其单次迭代的未命中总次数可以更小，而对于jki&kji，其单次迭代的未命中总次数仍然为2。这就拉大了两者的差距。

Exercise 3: Cache Blocking and Matrix Transposition

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 1000 33
Testing naive transpose: 1.107 milliseconds
Testing transpose with blocking: 1.001 milliseconds
PS C:\Users\12779\Desktop\lab04>
```

可见，代码正确。
```

Part 1：改变矩阵的大小

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 100 20
Testing naive transpose: 0 milliseconds
Testing transpose with blocking: 0 milliseconds
```

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 500 20
Testing naive transpose: 0 milliseconds
Testing transpose with blocking: 0 milliseconds
```

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 1000 20
Testing naive transpose: 1 milliseconds
Testing transpose with blocking: 1.06 milliseconds
```

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 2000 20
Testing naive transpose: 6.955 milliseconds
Testing transpose with blocking: 3.001 milliseconds
```

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 5000 20
Testing naive transpose: 122.759 milliseconds
Testing transpose with blocking: 22.955 milliseconds
```

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 20
Testing naive transpose: 649.785 milliseconds
Testing transpose with blocking: 175.899 milliseconds
```

问题回答：

在我们的测试中，当矩阵的规模较大的时候，使用矩阵分块实现的矩阵转置比原始方法更快。我们可以从缓存失效次数的角度进行分析。假设一个缓存块中可以存储8个数组元素。则原始方法的总缓存失效次数为$n^2 \times (\frac{1}{8}+1) = \frac{9n^2}{8}$，而采用分块的方法总缓存失效次数为$(\frac{n}{blocksize})^2 \times \frac{2blocksize^2}{8} = \frac{n^2}{4}$。但是由于矩阵分块需要更多的循环嵌套（4个），而循环语句会导致额外的开销。当矩阵规模较小时，这些开销占据主导，导致分块更慢；当矩阵规模较大时，cache miss 产生的开销占据主导。

Part 2：改变分块大小（Blocksize）

```
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 50
Testing naive transpose: 668.505 milliseconds
Testing transpose with blocking: 105.398 milliseconds
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 100
Testing naive transpose: 606.324 milliseconds
Testing transpose with blocking: 85.002 milliseconds
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 200
Testing naive transpose: 611.903 milliseconds
Testing transpose with blocking: 83.009 milliseconds
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 500
Testing naive transpose: 606.434 milliseconds
Testing transpose with blocking: 74.01 milliseconds
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 1000
Testing naive transpose: 608.877 milliseconds
Testing transpose with blocking: 92.255 milliseconds
PS C:\Users\12779\Desktop\lab04> .\transpose.exe 10000 5000
Testing naive transpose: 620.737 milliseconds
Testing transpose with blocking: 571.191 milliseconds
```

问题回答：

我们可以看到当blocksize增加时性能先上升再降低。开始性能上升是因为此时两个分块的大小小于缓存的容量$2blocksize^2 \leq C$，此时我们之前的推导是成立的。并且，通过硬件预取等技术，更大的分块可以提高缓存利用率，增加局部性和命中率；此外，内层循环的次数也减小，循环语句导致的开销也减小了。因此，总体性能上升。但是当分块大小更大的时候，此时两个分块的大小大于缓存容量$2blocksize^2 > C$，此时cache miss导致的开销占据主导，性能下降。事实上，当分块大小变为10000时，分块策略就退化成了原始策略。

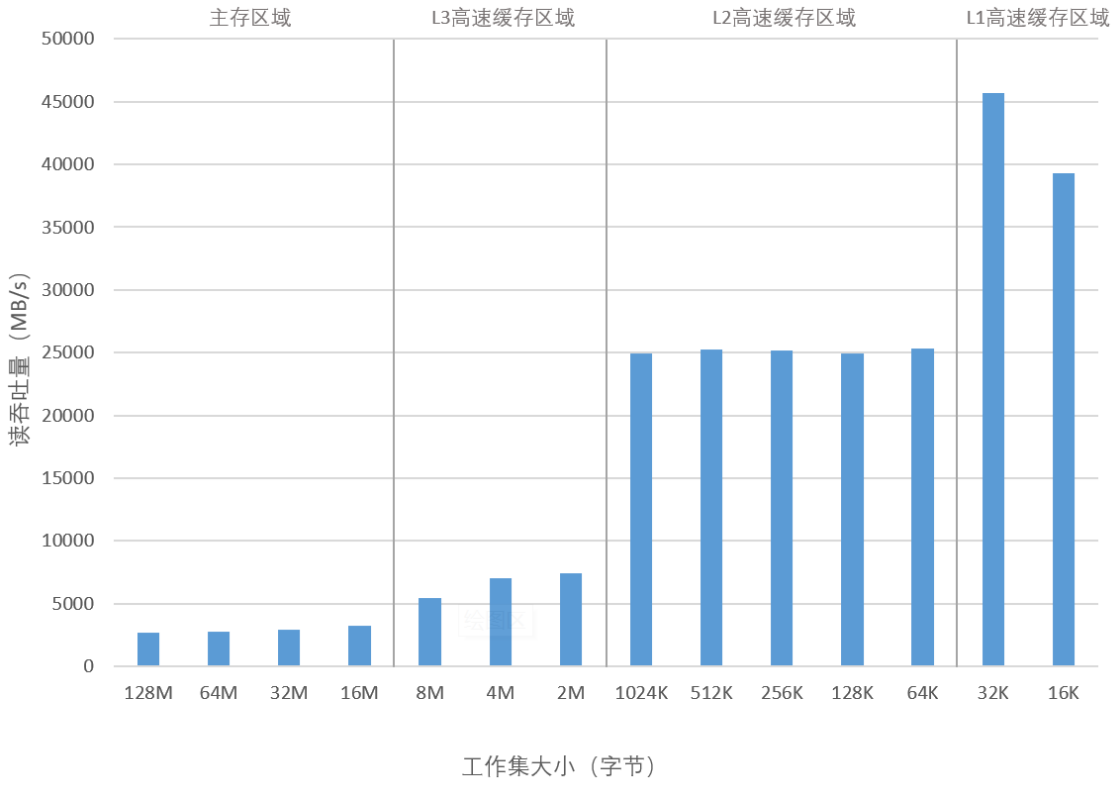
Exercise 4. Memory Mountain

问题回答：

- 运行结果如下：

```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab04/mountain$ ./mountain
Clock frequency is approx. 2419.0 MHz
Memory mountain (MB/sec)
        s1      s2      s3      s4      s5      s6      s7      s8      s9      s10     s11     s12     s13     s14     s15
128m    15170   9511    6727    5372    4352    3565    3044    2656    2471    2310    2171    2072    1971    1926    1867
64m     15007   9621    6922    5488    4457    3673    3145    2780    2543    2333    2223    2127    2027    1962    1871
32m     15411   9836    7154    5740    4681    3830    3285    2924    2708    2533    2370    2282    2143    2099    2022
16m     15772   10493   7959    6375    5267    4451    3762    3259    3094    2913    2790    2667    2598    2578    2580
8m      21768   14302   11593   9588    8150    7154    6210    5468    5248    5158    4996    4685    4770    4732    4756
4m      35715   24417   17944   13715   11143   9390    8098    7051    6611    6211    5933    5710    5493    5375    5311
2m      38121   26147   19291   14774   11905   10010   8583    7465    7068    6836    6939    7377    8378    10469   22408
1024k   42640   33156   33387   32887   31478   29315   27337   24920   24535   24863   24365   24344   24139   23774   23506
512k    42084   33134   33353   32911   31447   29559   27228   25224   24631   24746   24237   24335   23940   23683   23375
256k    42633   33488   33393   33138   32115   30187   28142   25164   24670   25005   24816   24985   24135   24003   23884
128k    42451   33481   33172   32754   31881   29922   28220   24958   24618   24655   24761   24901   23910   24067   23228
64k     42196   33166   32945   32673   31769   29686   27451   25341   24193   29355   42760   46028   46901   46407   45359
32k     50877   50424   48836   48215   47038   48037   46407   45660   43590   44272   43659   42606   42332   42882   41274
16k     50488   48098   47348   45660   44520   43734   42242   39318   38961   38469   37520   38846   35858   35826   35221
```

- 我们选取步长stride=8，不同工作集大小情况下的吞吐率如下：

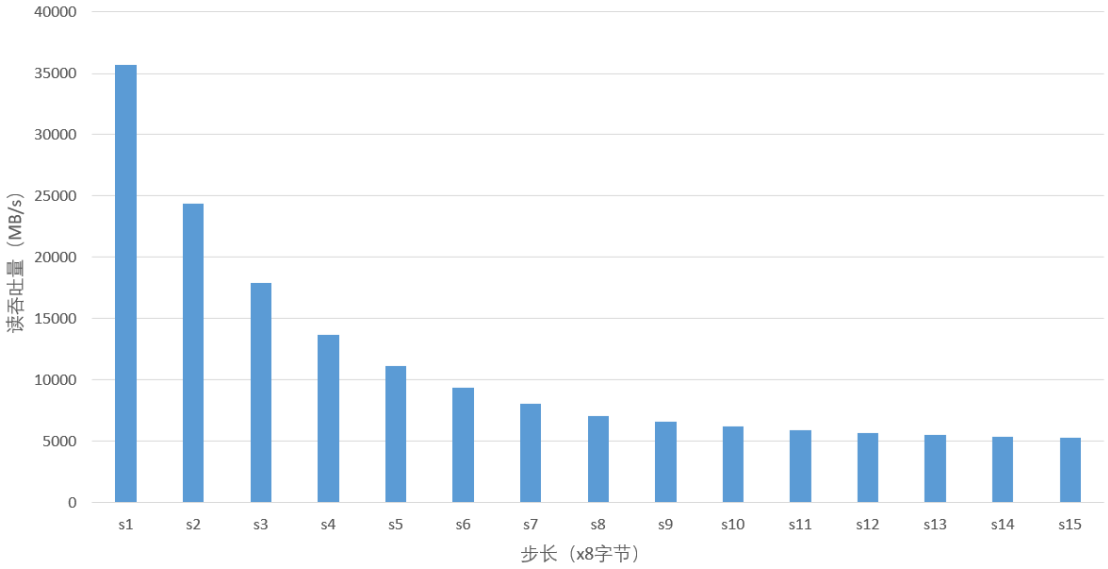


从上方图表中可以看出几条明显的分界线，因此该系统的一级高速缓存、二级高速缓存和三级高速缓存的大小分别为32KB、1024KB和8MB。

-

```
andycwd@DESKTOP-ES9E36H:/mnt/c/Users/12779/Desktop/lab04/mountain$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  1310720
LEVEL2_CACHE_ASSOC                 20
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 8
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

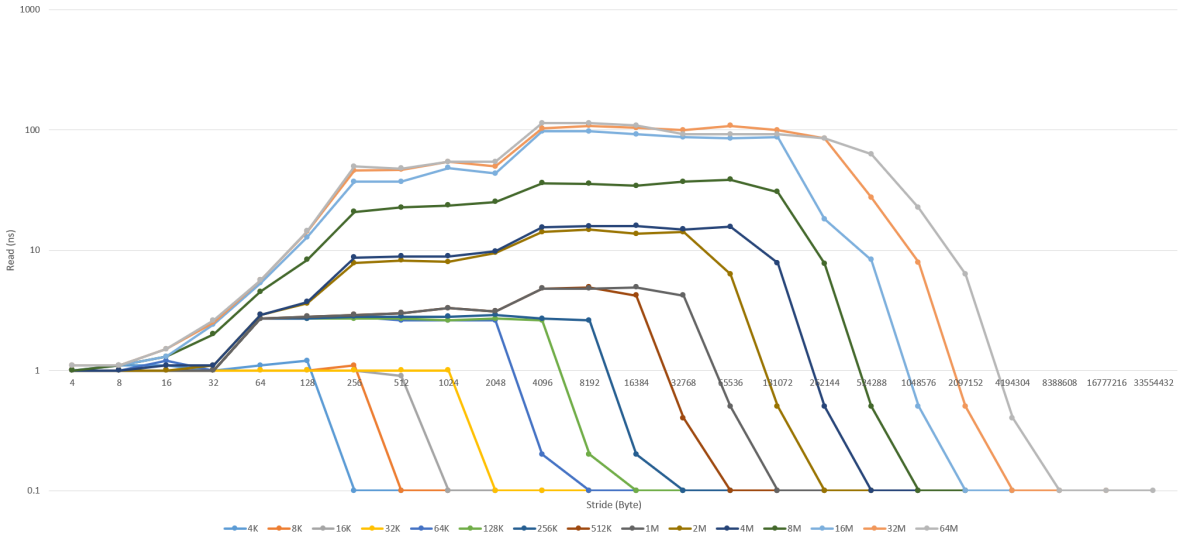
可以看出一级高速缓存、二级高速缓存和三级高速缓存的大小分别为48KB、1280KB和8MB。与我们的推测非常接近。

- 

可以看到从stride=8开始，读吞吐量基本保持不变，因此高速缓存的块大小应该为$8 \times 8B = 64B$。

Exercise 5: Memory Mountain（附加题，选做）

```
PS C:\Users\12779\Desktop\lab04> gcc -Wall -O3 -o test aca_ch2_cs2.c
PS C:\Users\12779\Desktop\lab04> .\test.exe
 ,4B,8B,16B,32B,64B,128B,256B,512B,1K,2K,4K,8K,16K,32K,64K,128K,256K,512K,1M,2M,4M,8M,16M,32M,
4K, 1.0, 1.1, 1.1, 1.0, 1.1, 1.2, 0.1, 0.1, 0.1, 0.1,
8K, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.1, 0.1, 0.1, 0.1, 0.1,
16K, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9, 0.1, 0.1, 0.1, 0.1,
32K, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.1, 0.1, 0.1, 0.1,
64K, 1.0, 1.0, 1.2, 1.0, 2.7, 2.7, 2.8, 2.6, 2.6, 2.6, 0.2, 0.1, 0.1, 0.1,
128K, 1.0, 1.0, 1.0, 1.0, 2.7, 2.7, 2.7, 2.7, 2.6, 2.7, 2.6, 0.2, 0.1, 0.1, 0.1,
256K, 1.0, 1.0, 1.0, 1.0, 2.7, 2.7, 2.8, 2.8, 2.8, 2.9, 2.7, 2.6, 0.2, 0.1, 0.1, 0.1,
512K, 1.0, 1.0, 1.0, 1.0, 2.7, 2.8, 2.9, 3.0, 3.3, 3.1, 4.8, 4.9, 4.2, 0.4, 0.1, 0.1, 0.1,
1M, 1.0, 1.0, 1.0, 1.0, 2.7, 2.8, 2.9, 3.0, 3.3, 3.1, 4.8, 4.8, 4.9, 4.2, 0.5, 0.1, 0.1, 0.1,
2M, 1.0, 1.0, 1.0, 1.1, 2.9, 3.6, 7.8, 8.2, 8.0, 9.5,14.1,14.8,13.7,14.1, 6.3, 0.5, 0.1, 0.1, 0.1,
4M, 1.0, 1.0, 1.1, 1.1, 2.9, 3.7, 8.7, 8.9, 8.9, 9.8,15.5,15.8,15.9,14.9,15.6, 7.8, 0.5, 0.1, 0.1, 0.1,
8M, 1.0, 1.1, 1.3, 2.0, 4.5, 8.3,20.8,22.7,23.5,25.2,36.1,35.7,34.3,37.0,38.5,30.5, 7.7, 0.5, 0.1, 0.1, 0.1,
16M, 1.1, 1.1, 1.3, 2.4, 5.3,12.8,37.0,37.0,48.2,43.3,97.3,97.3,91.7,86.7,85.1,86.7,18.1, 8.3, 0.5, 0.1, 0.1, 0.1,
32M, 1.1, 1.1, 1.5, 2.5, 5.6,14.2,45.8,46.7,54.2,49.7,103.0,108.4,103.7,99.3,108.4,99.3,85.1,27.4, 7.9, 0.5, 0.1, 0.1, 0.1,
64M, 1.1, 1.1, 1.5, 2.6, 5.6,14.4,49.7,47.7,54.2,54.2,113.8,113.8,108.4,91.7,91.7,91.7,85.1,62.7,22.5, 6.3, 0.4, 0.1, 0.1, 0.1,
```



问题回答：

- 从测试图像上来看，二级缓存的大小应该是1MB。我们可以固定某一大小的stride，例如64B，可以看到访存延迟有两个主要的分界线，其中第二条分界线出现在工作集为1MB处，因此二级缓存的大小为1MB。我们固定工作集大小为1MB，继续观察，可以发现从stride为64B开始，访存延迟基本不变，因此单个二级缓存块的大小为64B。
- 通过类似的分析，我们可以发现三级缓存的大小为8MB。观察图像，二级缓存命中时的访问延迟大约为2.6ns，三级缓存命中时的延迟大约为9ns，因此二级缓存的miss penalty大约为6.4ns。
- 我们可以观察到，对于工作集从64KB到1MB的曲线，当stride达到某个点时，其访问延迟会突然下降。该点的下降是因为此时对于缓存中的某一组的每一路都恰好别填满，不需要进行替换，因此缓存命中率大大提高。故我们的路数大约为 $\frac{64KB}{4096B} = 16$。经过查阅，本机的二级缓存为20路组相联，还是比较符合我们的推算的。
- 本机测试无法测得主存大小。从教材图中，我们发现512MB的曲线没有出现在图中，因此主存大小应该为512MB。
- 对本题的题意并不是很清楚。