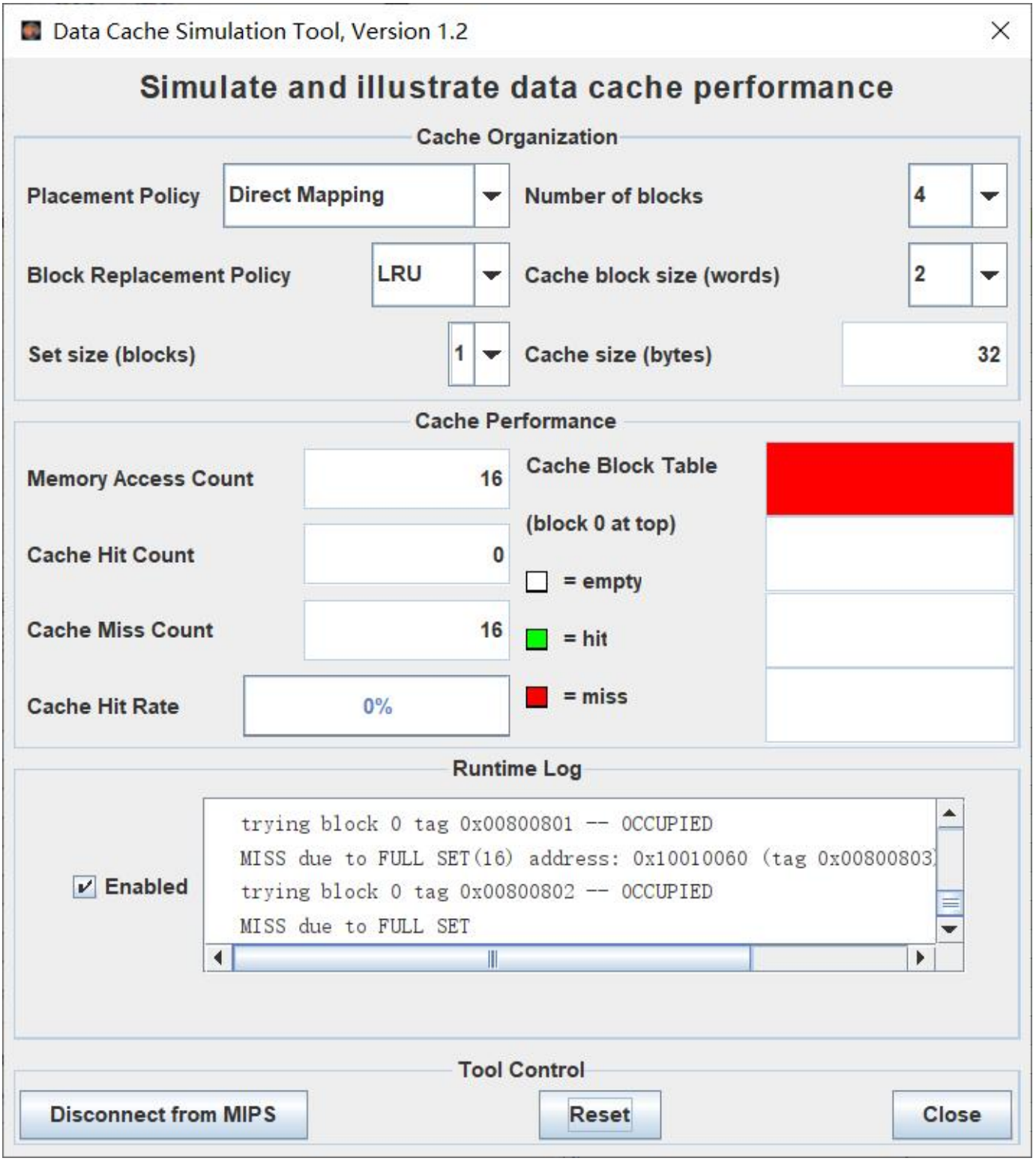
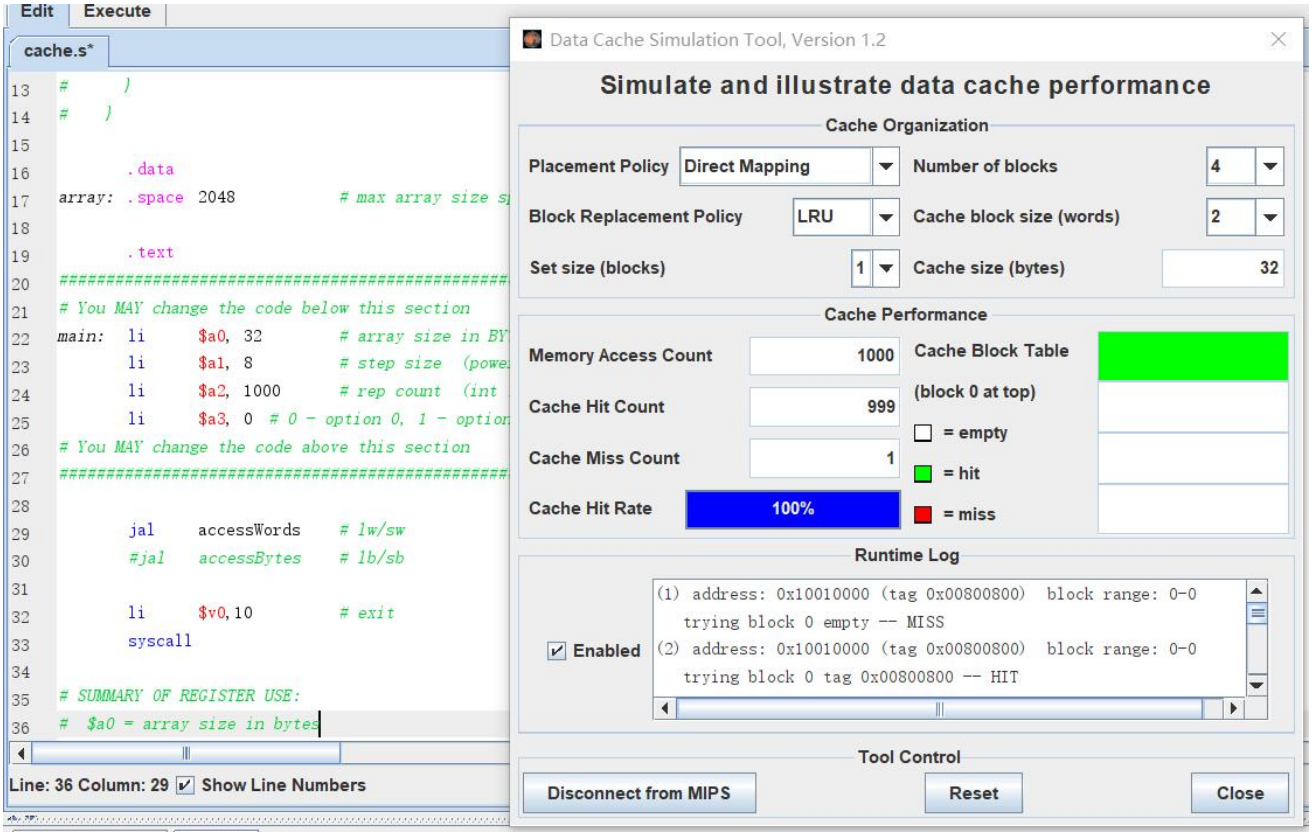
计算机系统结构 Lab04

Exercise 1: Cache Visualization

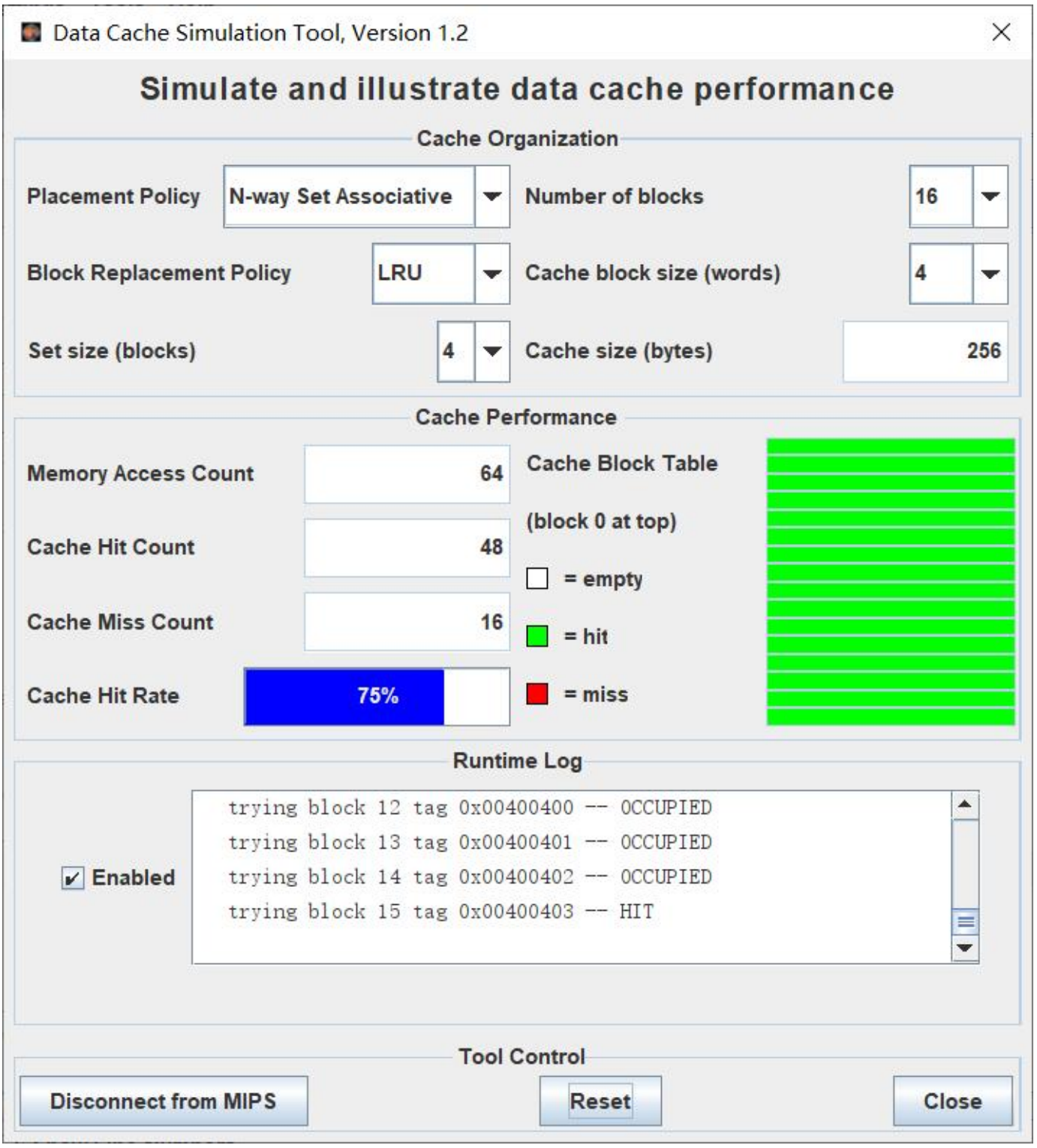
场景 1



- **Cache 命中率是多少?**
  命中率为0%。
- **为什么会出现这个 cache 命中率?**
  相邻两次访问的地址相差32bytes，不在同一个block内，同时由于下次访问会替换掉上一次访问的cache，所以每一次访问都会miss。
- **增加 Rep Count 参数的值，可以提高命中率吗? 为什么?**
  不可以。相邻两次访问的地址相差32bytes，刚好为cache大小，由于使用直接映射，每次访问都会映射到cache中同一个block，因此每一次访问都会替换掉上一次访问的cache，重复访问并不能提高命中率。
- **为了最大化 hit rate， 在不修改 cache 参数的情况下，如何修改程序中的参数（program parameters ）？ o maximize our hit rate?**
  将array size设置为32bytes，此情况下缓存中的块不会因为后续访问而被覆盖，因此仅第一次访问时缓存会失效，之后的访问均会命中，增大重复次数，当重复次数足够大时，命中率趋于100%。



场景 2



- **Cache 命中率是多少?**
  命中率为75%。
- **为什么会出现这个 cache 命中率?**
  Cache中每个block对应的内存空间被连续访问4次，除第1次访问miss外，剩下3次访问全部hit。
- **增加 Rep Count 参数的值，例如重复无限次，命中率是多少? 为什么?**
  命中率趋于100%。因为Cache的空间足够容纳整个数组，且访问过程中不会发生cache替换，因此第二次重复访问时，每一次访问都会hit。

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？**
  运行结果如下

```
ijk:    n = 1000, 1.751 Gflop/s
ikj:    n = 1000, 10.682 Gflop/s
jik:    n = 1000, 1.753 Gflop/s
jki:    n = 1000, 0.153 Gflop/s
kij:    n = 1000, 8.877 Gflop/s
kji:    n = 1000, 0.152 Gflop/s
```

  ikj嵌套的性能最好，jki和kji性能最差。

- **和你观察到的结果一致吗？最内层循环中数据访问的步长是怎么影响性能的？**
  和我观察到的结果一致。最内层数据访问步长越短，性能越高。缩短最内层数据访问步长可以提高程序的空间局部性，提高缓存命中率。

- **修改 matrixMultiply.c，再次观察程序的性能是否有改善，从中你得到哪些经验？**
  修改后运行结果如下

```
ijk:    n = 1000, 1.902 Gflop/s
ikj:    n = 1000, 11.709 Gflop/s
jik:    n = 1000, 1.860 Gflop/s
jki:    n = 1000, 0.151 Gflop/s
kij:    n = 1000, 7.771 Gflop/s
kji:    n = 1000, 0.151 Gflop/s
```

  对比发现程序性能有少量提升。
  将最常用的变量保存在寄存器中，可以在一定程度上提高程序性能。

- **为什么实际运算性能会差距如此大？**
  预取硬件可以识别出步长为1的访问模式，提前将内存中的内容缓存，进一步提高命中率。

Exercise 3: Cache Blocking and Matrix Transposition

`transpose_blocking` 函数实现如下：

```
1  void transpose_blocking(int n, int blocksize, int *dst, int *src){
2     int block_nr = (n-1)/blocksize + 1;
3     for(int bx=0;bx<block_nr;bx++)
4         for(int by=0;by<block_nr;by++)
5             for(int y = by*blocksize;y < (by+1)*blocksize && y < n;y++)
6                 for(int x = bx*blocksize;x < (bx+1)*blocksize && x < n;x++)
7                     dst[y + x * n] = src[x + y * n];
8  }
```

Part 1: 改变矩阵的大小

Matrix Size	100	500	1000	2000	5000	10000
Naive	0.006	0.241	1.05	23.374	236.755	1157.26
Blocking	0.005	0.223	1.173	5.472	35.768	197.643

- **矩阵分块实现矩阵转置是否比不用矩阵分块的方法快?**
  矩阵较小时，分块并没有明显更快，只有当矩阵大小达到一定大小时分块才能明显提高性能。
- **为什么矩阵大小要达到一定程度，矩阵分块算法才有效果?**
  矩阵较小时，原始矩阵可以完整或大部分放入缓存之中，因此分块的性能提升不大。

Part 2: 改变分块大小（Blocksize)

Block Size	50	100	200	500	1000	5000
Naive	1140.04	1123.95	1125.19	1163.29	1130.33	1118.88
Blocking	266.698	189.401	162.213	173.141	232.24	1079.58

- **当 blocksize 增加时性能呈现什么变化趋势? 为什么?**
  性能先提高后降低。blocksize过小时，缓存块中的数据不能被充分利用；当blocksize过大时，单个块无法被完整缓存，出现缓存互相覆盖的情况，因此性能接近于不分块的情况。

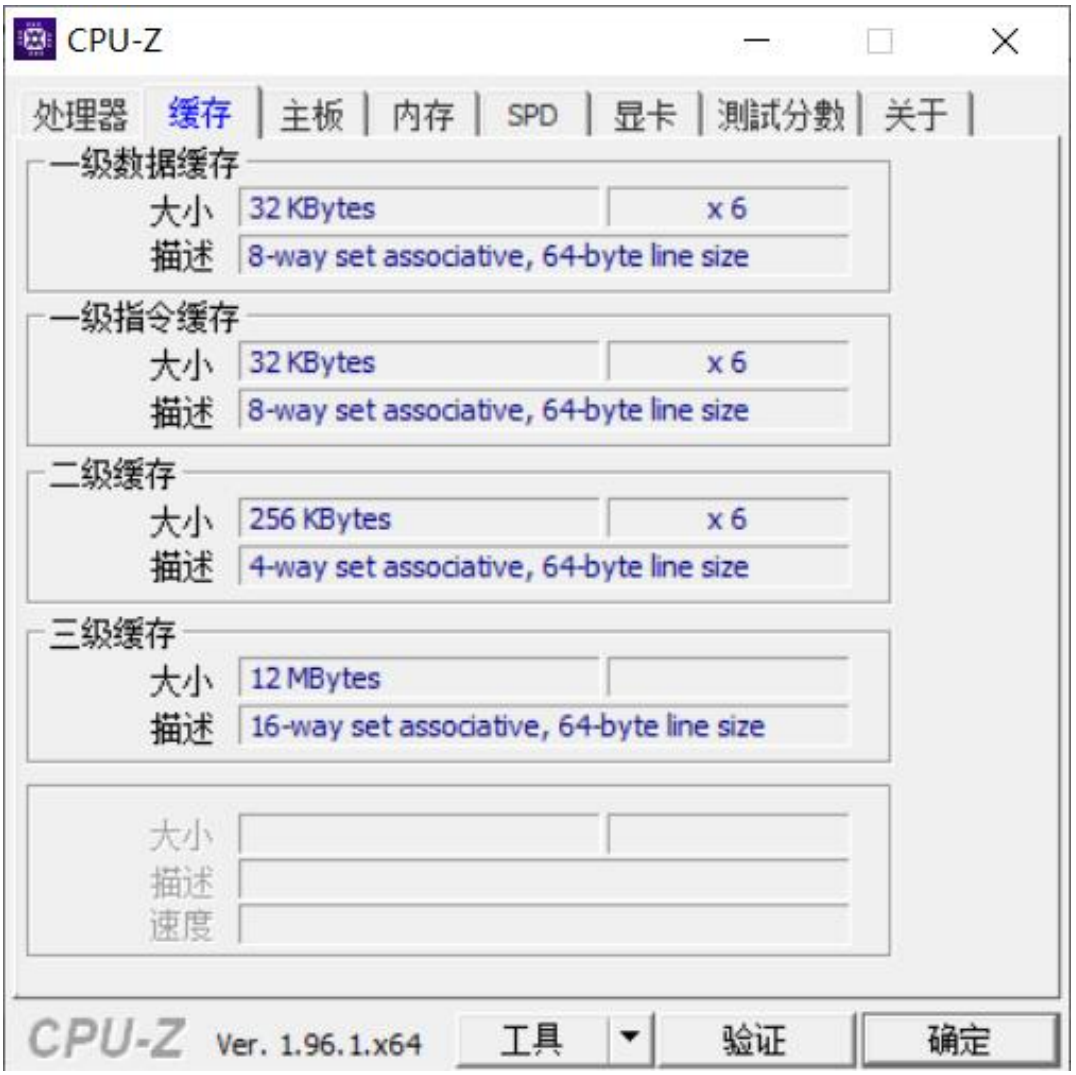
Exercise 4: Memory Mountain

- **请罗列出运行结果**

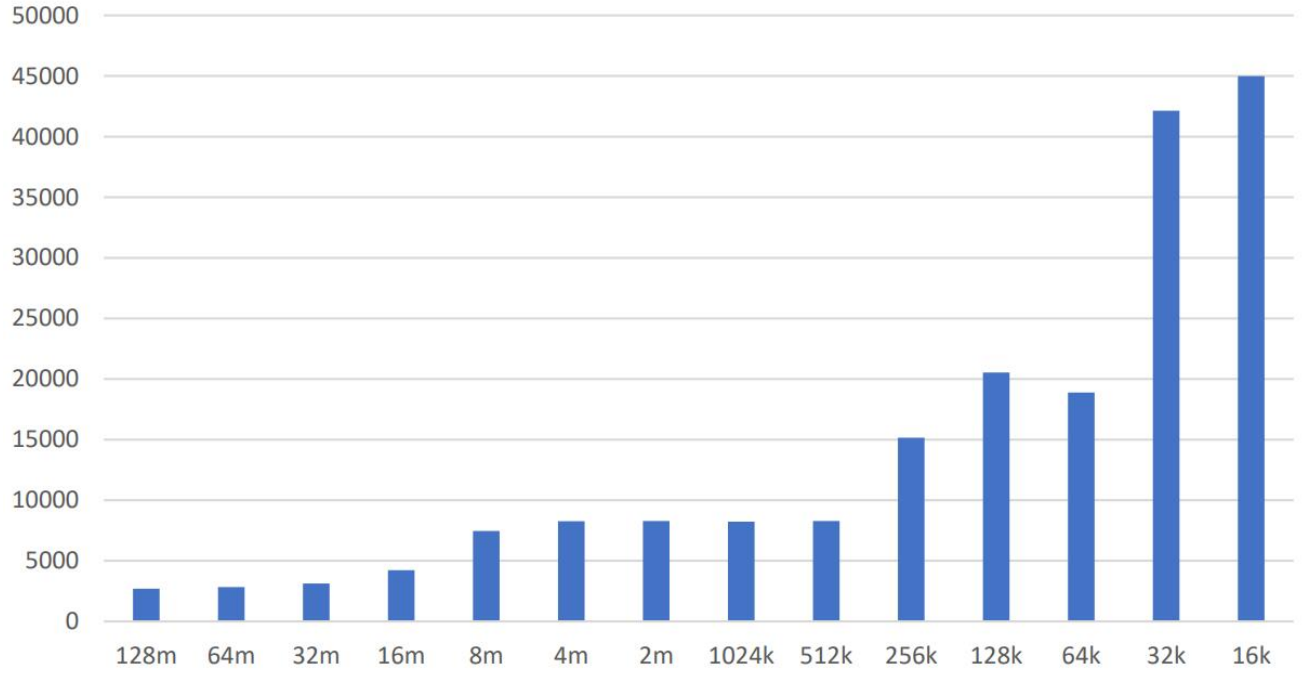
```
Clock frequency is approx. 2592.0 MHz
Memory mountain (MB/sec)
        s1      s2      s3      s4      s5      s6      s7      s8      s9      s10     s11     s12     s13     s14     s15
128m    17280   9632    6886    5274    4149    3554    3008    2707    2548    2481    2321    2161    2125    2044    2018
64m     16994   10007   7326    5475    4358    3722    3260    2831    2692    2400    2380    2213    2142    2053    2078
32m     16966   10477   7641    5711    4735    3996    3448    3132    2852    2788    2658    2540    2393    2367    2224
16m     18109   11724   9691    7375    5997    5089    4447    4227    4017    3894    4209    4163    3846    3862    3963
8m      27954   18080   14704   12379   10412   9161    8263    7465    7168    6858    6513    6202    5943    5846    5744
4m      32254   21518   17878   14934   12500   10668   9322    8279    7716    7171    6817    6413    6168    5987    5875
2m      32237   21244   17860   14941   12436   10686   9343    8292    7751    7263    6808    6407    6168    5989    5881
1024k   32219   21480   17913   14901   12455   10709   9339    8226    7721    7248    6793    6410    6187    5970    5869
512k    32095   21834   17984   15118   12602   10785   9436    8289    7883    7411    7119    6838    6942    6898    7147
256k    37334   28738   26239   24246   21715   19559   16970   15167   15093   15471   15702   16289   16022   16407   18369
128k    38748   31260   30656   29008   25147   22240   21268   20535   17096   17089   17648   19551   20873   17458   16484
64k     38872   31134   30640   28850   25165   22468   19989   18891   14721   14796   14762   14593   14814   16135   35169
32k     35687   46977   46107   45371   44233   43959   42424   42130   42120   43323   48244   45360   39828   43951   45653
16k     44609   46160   44512   43871   42457   41140   44598   44987   46249   45167   42883   34687   34744   32964   44226
```

- **程序运行所在的系统，一级高速缓存、二级高速缓存的大小分别为多大？有三级高速缓存吗？ 如果有，容量为多少?**
  每个物理核心一级数据缓存为32KB，一级指令缓存为32KB，二级高速缓存为256KB。所有核心共享12MB的三级高速缓存。

- **查看系统中高速缓存的配置，并截图。**



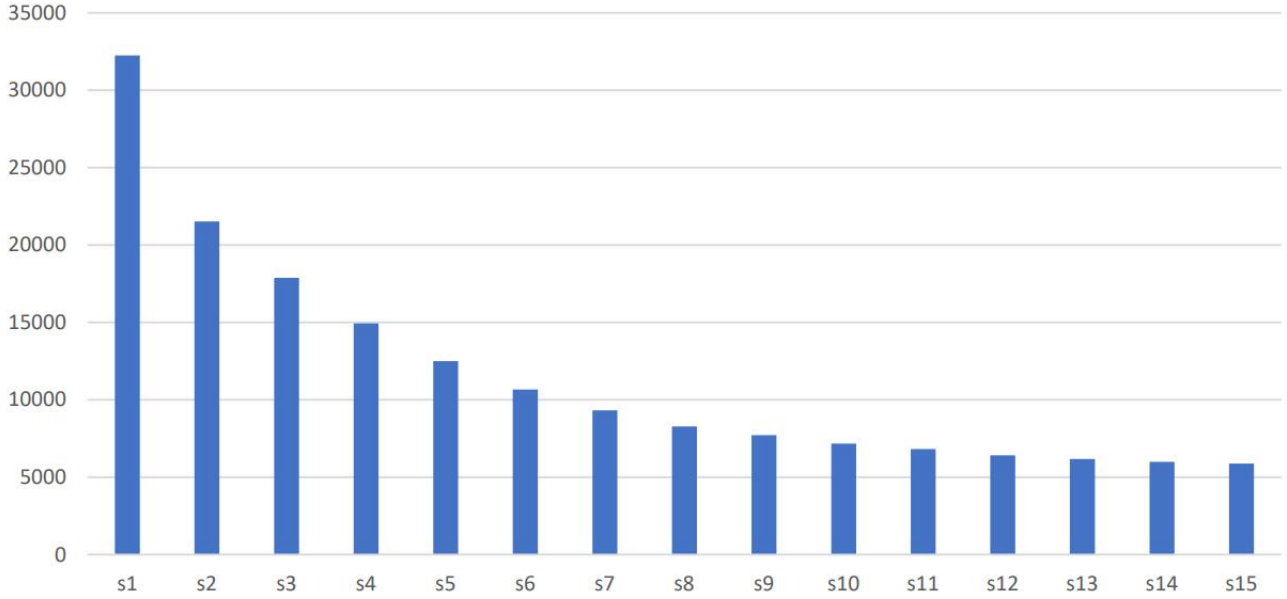
将步长8的测试数据作图如下：



可见断层分别出现在8~16M，256KB，32KB。

由测试数据得出的结论与系统配置相符。

- **高速缓存的块大小（block size）是多少？ 为什么?**



块大小为64bytes，从图中可以看出当步长达到8之后，继续增大步长吞吐率变化缓慢。

Exercise 5: Memory Mountain（附加题，选做）

运行结果如下：

	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	1.6	1.5	1.6	1.7	1.7	1.7	1.8	2.2	3	3.5														
8K	1.5	1.6	1.6	1.6	1.6	1.6	1.7	1.9	2.3	3.1	3.7													
16K	1.5	1.6	1.5	1.6	1.6	1.6	1.7	1.7	1.9	2.3	3.2	3.8												
32K	1.6	1.6	1.6	1.6	1.7	1.7	1.8	2	2.3	1.9	2.3	3.2	3.8											
64K	1.5	1.6	1.6	1.8	3.4	3.4	3.4	3.4	3.4	3.5	3.6	2.4	3.2	3.8										
128K	1.5	1.6	1.6	1.8	3.4	3.4	3.4	3.4	3.4	3.4	3.5	3.6	2.4	3.2	3.7									
256K	1.6	1.6	1.6	1.9	3.4	3.6	3.7	3.7	3.8	4	4.1	3.5	3.7	2.4	3.2	3.7								
512K	1.6	1.6	1.6	1.9	3.7	4.5	7	7.9	9.4	11.6	12.7	12.6	12.4	12.6	4.6	3.3	3.7							
1M	1.6	1.6	1.6	2	3.7	4.5	7.1	7.9	9.4	11.9	12.8	12.7	12.7	12.7	12.7	4.6	3.3	3.7						
2M	1.6	1.6	1.7	2.1	3.9	4.7	7.2	8.1	9.6	11.7	13	13	12.9	12.8	12.9	12.8	4.7	3.4	3.7					
4M	1.7	1.7	1.9	2.4	4.5	6.3	7.8	8.3	9.9	12	13	12.9	13	12.9	13.1	13.1	13.2	4.7	3.3	3.6				
8M	1.7	1.8	2	2.7	5.3	14.5	14.9	12.6	13.7	19	22.9	17	15.6	15.7	20.8	12.5	12.6	12.4	4.5	3.1	3.5			
16M	1.6	1.8	2.1	3.1	6.1	19.6	35.3	34.6	35.1	39.7	59.6	61.9	39.4	42.6	57.5	12.5	12.5	12.3	12.2	4.4	3.1	3.4		
32M	1.6	1.7	2	2.9	5.7	18.1	34.1	35.1	41.8	54.2	61.1	58.2	54.2	62.7	64.4	56.8	20.4	12.4	12.3	12.4	4.4	3.1	3.4	
64M	1.6	1.7	2	2.9	5.7	18.1	34.1	37.3	44.2	59.6	70.1	66.2	62.7	70.1	62.7	70.1	56.8	12.8	12.3	12.4	12.4	4.4	3	3.4