Lab - Data Locality and Cache Blocking and Memory mountain

Exercise 1: Cache Visualization

本实验利用 Venus venus (cs61c.org) 中的可视化工具帮助你理解 cache 的性能及其相关术语。可视化工具使用步骤如下：

1. 将汇编语言文件（cache.s）复制到 Venus 的 Editor 中，在 simulator 中选择“Assemble and Simulate from Editor”
2. 在右边的观察窗口中，设置 cache 参数
3. 在 simulator 中运行汇编程序，如果直接运行代码，你可以在模拟器中看到数据 cache 的最终状态以及命中率。你也可以在每次访存时设置断点，以观察数据 cache 的命中和缺失.

```
# You MAY change the code below this section
main:   li    a0, 256   # array size in BYTES (power of 2 < array size)
        li    a1, 2     # step size   (power of 2 > 0)
        li    a2, 1     # rep count   (int > 0)
        li    a3, 1     # 0 - option 0, 1 - option 1
# You MAY change the code above this section
```

场景 1：(使用 cache.s)

Program Parameters:

- Array Size: 128 (bytes)
- Step Size(访问步长): 8
- Rep Count（重复次数）: 4
- Option: 0 （同一个单元一次访问：写）

Cache Parameters: (set these in the Cache tab)

- Cache Levels: 1
- 数据块大小（Block Size）: 8 bytes
- 总块数（Number of Blocks）: 4
- 关联度（Associativity）: 1 (不能修改)
- Enable?: Should be green
- 放置策略（Placement Policy）: Direct Mapped
- 替换策略（Block Replacement Policy）: LRU

回答问题：

- Cache 命中率是多少？
- **命中率为0.**
- 为什么会出现这个 cache 命中率？
- **因为访问的数组元素的内存块数对应的缓存块数总为同一个。**
- 增加 Rep Count 参数的值，可以提高命中率吗？为什么？
- **不能。因为这仍然不能改变上述问题。**
- 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 ）？
- **可以修改访问步长为1，这样会使得命中率为50%左右。**

场景 2：(使用 cache.s)

Program Parameters:

- Array Size: 256 (bytes)
- Step Size: 2
- Rep Count: 1
- Option: 1 （同一个单元两次访问：读和写）

Cache Parameters:

- Cache Levels: 1
- 数据块大小（ block size）： 16 bytes
- 总块数（Number of Blocks）： 16
- 关联度 (Associativity、组内的 block 数)： 4
- Enable?: Should be green
- 放置策略（Placement Policy）： N-way Set Associative
- 替换策略（Block Replacement Policy）： LRU

回答问题：

- Cache 命中率是多少？
- **命中率为0.75**
- 为什么会出现这个 cache 命中率？
- **由于cache块大小为16字节（4个int），每miss一个之后三次访问操作都会命中，故命中率为0.75**
- 增加 Rep Count 参数的值，例如重复无限次，命中率是多少？为什么？
- **命中率逐渐逼近1.因为第一次重复已经使得缓存中加入了整个数组，之后的所有访问均会命中。**

场景 3：(使用 cache.s)

Program Parameters:

- Array Size (a0): 128 (bytes)
- Step Size (a1): 1
- Rep Count (a2): 1
- Option (a3): 0

Cache Parameters: (set these in the Cache tab)

- Cache Levels: 2
- L1 cache
  - Block Size: 8 bytes
  - Number of Blocks: 8
  - Enable?: Should be green
  - Placement Policy: Direct Mapped
  - Associativity: 1
  - Block Replacement Policy: LRU
- L2 cache
  - Block Size: 8 bytes
  - Number of Blocks: 16
  - Enable?: Should be green
  - Placement Policy: Direct Mapped

- Associativity: 1
- Block Replacement Policy: LRU

回答问题：

- L1 cache 和 L2 cache 的命中率分别为多少？
- 命中率分别为0.5和0
- 总共访问了 L1 cache 几次？ L1 Miss 次数为多少？32次和16次
- 总共访问了 L2 cache 几次？16次
- 哪一个程序参数（寄存器 a0 ~ a3）可以增加 L2 hit rate，并且保持 L1 hit rate 不变？寄存器a2
- 如果将 L1 cache 中的块数增加， L1 、 L2 hit rate 有什么变化 ？
- L1和L2 hit rate未发生改变
- 如果将 L1 cache 中的块大小增加，L1 、 L2 hit rate 有什么变化 ？
- L1命中率增加，而L2命中率不变

Exercise 2: Loop Ordering and Matrix Multiplication

矩阵相乘是很多问题的核心算法。将两个矩阵相乘，我们可以简单采用 3 层嵌套循环，对应 c 语言程序如下：

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i*n+j] += A[i*n+k] * B[k*n+j];
```

上述代码将矩阵 A 乘以 B 得到结果矩阵 C，嵌套循环的顺序是 i, j, k. 在最内层循环 (k), 以步长 1（stride=1）访问 A 的元素，以步长 n（stride=n）访问 B 的元素，以步长 0（stride=0）访问 C 的元素.

虽然循环的嵌套顺序并不会影响矩阵相乘结果的正确性，但由于时间局部性和空间局部性，循环嵌套顺序会影响高速缓存的命中率。

观察程序 matrixMultiply.c ， 程序用不同的循环嵌套顺序来实现矩阵相乘，并使用浮点运算吞吐率 Gflops/s 来衡量不同实现方式的性能。 使用 make 编译并执行 matrixMultiply（Makefile 中已经使用了 '-O3' 最高级别编译优化）.

```
$ make ex2
```

回答问题：

- 1000-1000 的矩阵相乘，哪种嵌套顺序性能最好？（ikj嵌套方式）哪种嵌套顺序性能最差？（jki嵌套方式）
- 教材《深入理解计算机系统》（CSAPP 3e 中文版 P449） 分析了 6 个版本的矩阵乘法最内层循环中的 cache miss 次数， 如下图所示。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。
  当最内层循环中数据访问的步长长的时候，缓存块大小不能跟上，这就使得每一次访问缓存块中均不能命中，使得性能很低。

矩阵乘法版本（类）	每次迭代					
	加载次数	存储次数	*A*未命中次数	*B*未命中次数	*C*未命中次数	未命中总次数
ijk & *jik* (*AB*)	2	0	0.25	1.00	0.00	1.25
jki & *kji* (*AC*)	2	1	1.00	0.00	1.00	2.00
kij & *ikj* (*BC*)	2	1	0.00	0.25	0.25	0.50

图 6-45 矩阵乘法内循环的分析。6 个版本分为 3 个等价类，用内循环中访问的数组对来表示

- 参考如下代码（CSAPP 3e 中文版 P448），修改 matrixMultiply.c，再次观察程序的性能是否有改善（浮点运算吞吐率 Gflops/s），从中你得到哪些经验？我主要对ijk版本进行了该修改，发现性能有部分改善。这说明通过局部变量代替内存访问能够减少cache miss的可能性，从而改善程序性能。

code/mem/matmult/mm.c

```
1  for (i = 0; i < n; i++)
2      for (j = 0; j < n; j++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

a) *ijk*版本

```
1  for (j = 0; j < n; j++)
2      for (i = 0; i < n; i++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

b) *jik*版本

```
1  for (j = 0; j < n; j++)
2      for (k = 0; k < n; k++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }
```

c) *jki*版本

```
1  for (k = 0; k < n; k++)
2      for (j = 0; j < n; j++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }
```

d) *kji*版本

```
1  for (k = 0; k < n; k++)
2      for (i = 0; i < n; i++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }
```

```
1  for (i = 0; i < n; i++)
2      for (k = 0; k < n; k++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += A[i][k]*r;
6      }
```

- 教材《深入理解计算机系统》（CSAPP 3e 中文版 P449）在 Intel core i7 处理器上分析了 6 个版本的矩阵乘法的性能，可以发现：当矩阵大小为 700*700 时，最块的版本比最慢的版本快超过 30 倍，在图 6-45 中的分析可以看出：这两种算法的 cache 失效率相差的倍数仅为 4 倍，为什么实际运算性能会差距如此大？

  当cache失效时，需要cache的重新加载以及cpu的重新读取，这可能使得cache失效时所耗费的时间远高于cache命中时指令花费的时间，这就使得当cache失效次数有较小增加时可能总运算时间增加很多。

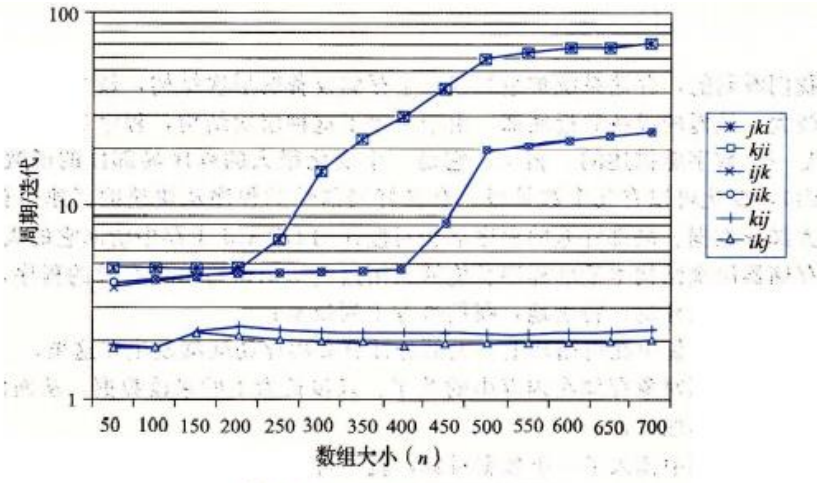
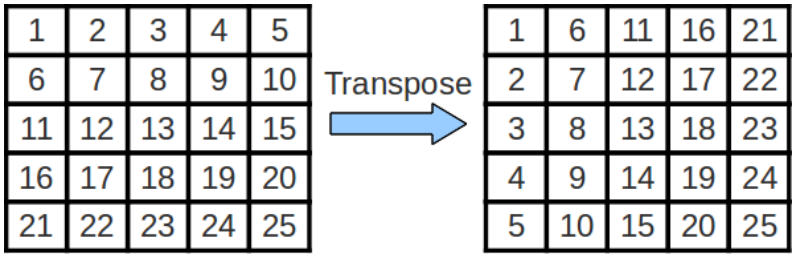


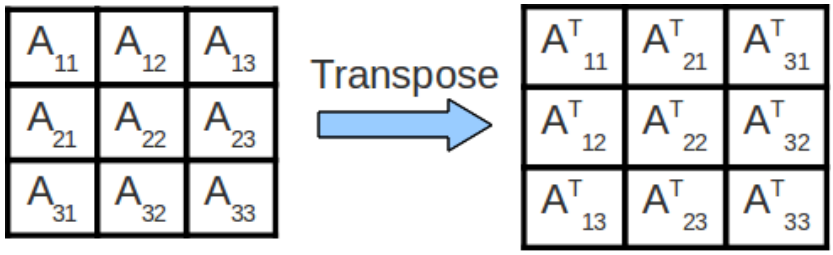
图 6-46 Core i7 矩阵乘法性能

Exercise 3: Cache Blocking and Matrix Transposition

矩阵转置的示意图如下：



为了提高从内存中访问的数据的时间和空间局部性、减少高速缓存失效次数，通常会采用高速缓存分块（cache blocking） 技术。例如：矩阵转置可以考虑采用一次转置一个 block 的方法，如下图所示，每次转置一个 block Aij 到结果矩阵的最终位置。这个方法减少了 cache 工作集的大小，进而提升性能。



实现 transpose.c 中的 transpose_blocking() 函数. 矩阵的宽度(n) 可以是任意值，不一定是 blocksize 的倍数. 然后运行代码：

```
$ make ex3
$ ./transpose <n> <blocksize>
```

你可以设置 n=1000 ， blocksize=33 验证你的代码是否正确。

Part 1: 改变矩阵的大小

将 blocksize 固定为 20， n 分别设置为 100, 500，1000, 2000, 5000, 和 10000. 矩阵分块实现矩阵转置是否比不用矩阵分块的方法快? 为什么矩阵大小要达到一定程度，矩阵分块算法才有效果？

根据结果能够发现分块后计算时间更少，说明分块的算法的确更快。
当矩阵大小较大时，普通算法计算时缓存已经不能容纳整个矩阵（行），其导致的矩阵命中率将会明显更低；这是矩阵分块算法才能派上用场。

Part 2: 改变分块大小（Blocksize）

将 n 的值固定为 10000，将 blocksize 设置为 50，100，200，500，1000，5000 分别多次运行 transpose 程序. 当 blocksize 增加时性能呈现什么变化趋势？为什么？

性能最终逐渐变差。当blocksize过大后，缓存对于分块后的矩阵仍然不能容纳，这导致miss率增加，导致性能变差。

Exercise 4: Memory Mountain

code/mem/mountain/mountain.c

```
1  long data[MAXELEMS];      /* The global array we'll be traversing */
2
3  /* test - Iterate over first "elems" elements of array "data" with
4   *        stride of "stride", using 4 x 4 loop unrolling.
5   */
6  int test(int elems, int stride)
7  {
8      long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
9      long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
10     long length = elems;
11     long limit = length - sx4;
12
13     /* Combine 4 elements at a time */
14     for (i = 0; i < limit; i += sx4) {
15         acc0 = acc0 + data[i];
16         acc1 = acc1 + data[i+stride];
17         acc2 = acc2 + data[i+sx2];
18         acc3 = acc3 + data[i+sx3];
19     }
20
21     /* Finish any remaining elements */
22     for (; i < length; i+=stride) {
23         acc0 = acc0 + data[i];
24     }
25     return ((acc0 + acc1) + (acc2 + acc3));
26 }
27
28 /* run - Run test(elems, stride) and return read throughput (MB/s).
29  *       "size" is in bytes, "stride" is in array elements, and Mhz is
30  *       CPU clock frequency in Mhz.
31  */
32 double run(int size, int stride, double Mhz)
33 {
34     double cycles;
35     int elems = size / sizeof(double);
36
37     test(elems, stride);                    /* Warm up the cache */
38     cycles = fcyc2(test, elems, stride, 0);  /* Call test(elems,stride) */
39     return (size / stride) / (cycles / Mhz); /* Convert cycles to MB/s */
40 }
```

code/mem/mountain/mountain.c

本练习通过测量程序的读吞吐率（ MB/s） 讨论存储系统的性能（带宽,Bandwidth）。运行的程序来自于教材《深入理解计算机系统》（CSAPP 3e 中文版 P444）。

Test 函数以步长 stride 扫描数组的头 elems 个元素，为提高可用的指令并行性，使用 4*4 展开。Run 函数调用 test 函数，并返回测量出的读吞吐率。Run 函数的参数 size 和 stride 用来控制读序列的时间和空间局部性。Size 值越小，工作集越小，时间局部性越好。Stride 值越小，空间局部性越好。我们给出的程序中，size 从 16KB 变化到 128MB， stride 从 1 变到 15 个元素。每个元素是一个 long long int。

步骤如下：

```
$ cd mountain    // 进入 lab3 下的子目录 mountain
$ make mountain  //编译
$ ./mountain     // 运行程序
```

- 请罗列出运行结果。

```
Clock frequency is approx. 2994.3 MHz
Memory mountain (MB/sec)
        s1      s2      s3      s4      s5      s6      s7      s8      s9      s10     s11     s12     s13     s14     s15
128m    9993    6070    4232    3286    2707    2341    1879    1689    1673    1592    1474    1529    1393    1444    1412
64m     10487   6132    4308    3498    2691    2330    1908    1705    1655    1580    1493    1557    1497    1430    1354
32m     10683   6702    4547    3476    2818    2385    2013    1980    1620    1772    1512    1624    1595    1518    1506
16m     9449    6091    4401    3437    2697    2754    1986    2059    1957    1899    1821    1810    1756    1710    1705
8m      13424   5725    5422    4276    3290    2818    2415    2143    2022    1920    1240    1865    1789    1841    1903
4m      24629   14594   10560   8211    6754    5725    5027    4493    4426    4365    4439    4629    4911    5487    5742
2m      36663   26149   19526   14810   12193   10434   8997    7920    7741    7598    7249    6963    6694    6513    6270
1024k   32264   22029   15604   11808   9939    8244    7133    6212    5944    5699    5325    5022    4745    4652    4453
512k    25086   16182   11203   8456    7037    5826    5061    4369    4221    3884    3638    3468    3252    3147    3013
256k    18079   10832   7598    5650    4640    3866    3285    2900    2648    2450    2270    2121    1977    1854    1765
128k    11398   6363    4458    3344    2675    2252    1930    1672    1517    1380    1268    1174    1084    1018    960
64k     6620    3524    2401    1800    1456    1227    1040    910     818     736     677     621     579     532     496
32k     3523    1841    1255    931     753     628     538     471     418     381     350     321     296     275     257
16k     1883    975     642     482     385     321     279     241     214     195     177     162     150     139     130
```

从运行结果中，模仿下图，固定一个步长（例如 stride=8），罗列出不同工作集大小情况下的读吞吐率. 并总结：

- 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？
- （由于我得到的数据和示例明显不同）我的判断是一级高速缓存：2M,二级
- 高级缓存：2M，三级高速缓存：8M
- 在 windows 下你可安装（cpu-z）,在 linux 系统下你可以使用命令：
  $ getconf -a | grep CACHE
- 查看系统中高速缓存的配置，并截图。对比一下你的判断是否和系统配置一致。
- 截图在右侧。我仅仅对三级缓存大小判断正确。

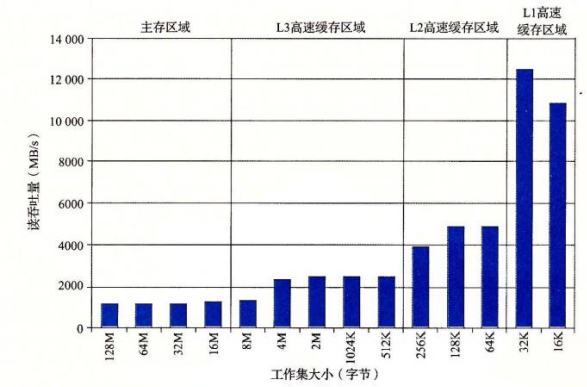


图 6-42 存储器山中时间局部性的山脊。这幅图展示了图 6-41 中 stride=8 时的一个片段

```
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                  524288
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 0
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

继续观察程序运行结果，固定工作集大小，模仿下图，例如数组长度为 4MB，观察步长从 1 变化到 15 的情况下读数据的吞吐率。回答问题：高速缓存的块大小（block size）是多少？ 为什么？

高速缓存块大小是64字节。从图中我们发现当步长大于8之后，吞吐量几乎未发生变化，这说明访问时已经不存在空间局部性，也即cache更新的内容完全未包含下一次的访问内容。

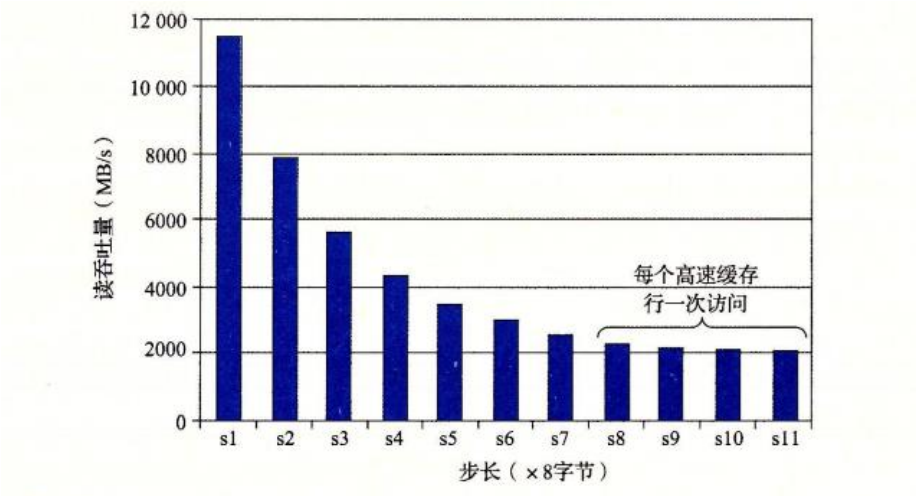


图 6-43 一个空间局部性的斜坡。这幅图展示了图 6-41 中大小=4MB 时的一个片段

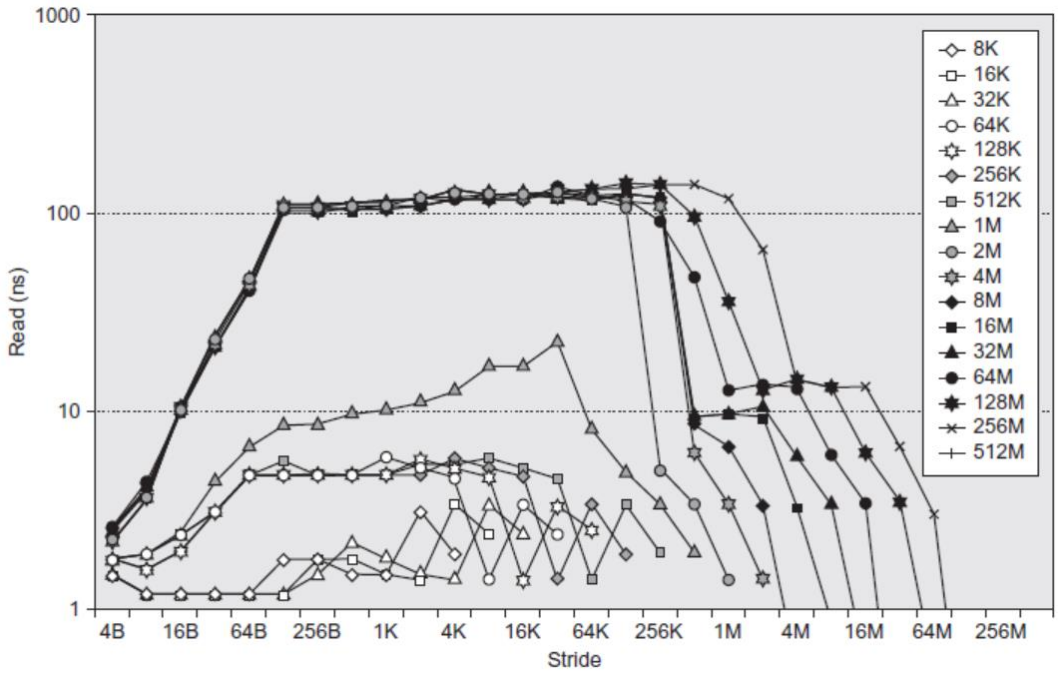
Exercise 5: Memory Mountain（附加题，选做）

本练习通过测量程序的平均访存延迟（Memory access latency: 单位 ns） 讨论存储系统的性能。

运行的程序来自于参考书：
John L. Hennessy，David A. Patterson. Computer Architecture: A Quantitative Approach, Fifth Edition，计算机体系结构：量化研究方法(第5版）电子书 P133
Case Study 2: Putting It All Together: Highly Parallel Memory Systems。Crosscutting Issues: The Design of Memory Hierarchies 问题 2.4

你也可以去这个网站下载该程序：www.hpl.hp.com/research/cacti/aca_ch2_cs2.c

程序按不同的步长（stride）访问不同大小的工作集中的数组元素，重复多次，计算平均访问延迟（单位 ：ns）。Stride 最小为 4B，最大为 32MB，工作集最小为 4KB，最大为 64MB。
下图是教材作者的实验结果。



对应 Microsoft Visual C++程序可下载自:

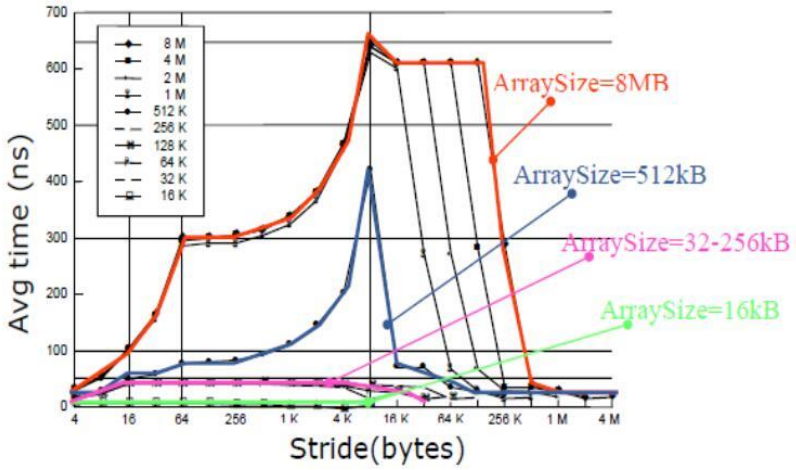
www.hpl.hp.com/research/cacti/aca_ch2_cs2.c

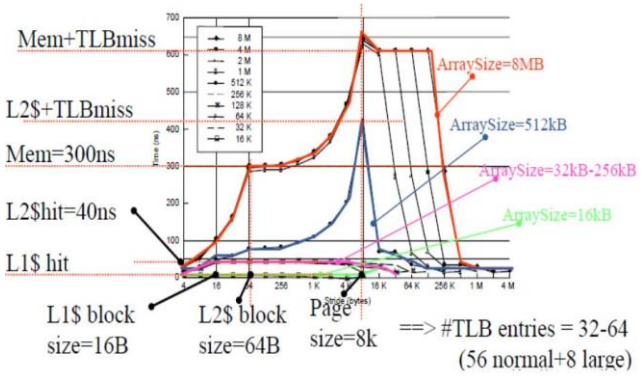
该程序简单修改头文件，就可以在 linux 和 windows 上运行。运行时间较长，请耐心等待结果。结果存为 csv 文件可以直接用 excel 打开并绘制折线图。

从实验结果推断以下信息：

- What are the overall size and block size of the second-level cache?
- What is the miss penalty of the second-level cache?
- What is the associativity of the second-level cache?
- What is the size of the main memory?
- What is the paging time if the page size is 4 KB?

教材中给出的答案有错误，我们就不提供教材中的答案给大家了。但会给大家一下提示。下面两个图是程序在一个只有两级高速缓存的系统上的运行结果以及分析，希望能给你一些帮助。





另外，你还可以考虑以下问题：

- If necessary, modify the code to measure the following system characteristics. Plot the experimental results with elapsed time on the y-axis and the memory stride on the x-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

a. What is the system page size?

b. How many entries are there in the translation lookaside buffer (TLB)?

c. What is the miss penalty for the TLB?

d. What is the associativity of the TLB?

Hint: This is visible in the graph above as a slight increase in L2 miss service time for large data sets, and is 4KB for the graph above. b. Hint: Take independent strides by the page size and look for increases in latency not attributable to cache sizes. This may be hard to discern if the amount of memory mapped by the TLB is almost the same as the size as a cache level. c. Hint: This is visible in the graph above as a slight increase in L2 miss service time for large data sets, and is 15ns in the graph above. d. Hint: Take independent strides that are multiples of the page size to see if the TLB if fully-associative or set-associative. This may be hard to discern if the amount of memory mapped by the TLB is almost the same as the size as a cache level

- In multiprocessor memory systems, lower levels of the memory hierarchy may not be able to be saturated by a single processor but should be able to be saturated by multiple processors working together. Modify the code and run multiple copies at the same time. Can you determine:

  a. How many actual processors are in your computer system and how many system processors are just additional multithreaded contexts?

  b. How many memory controllers does your system have?

  a. Hint: Look at the speed of programs that easily fit in the top-level cache as a function of the number of threads. b. Hint: Compare the performance of independent references as a function of their placement in memory.