Lab - Data Locality and Cache Blocking and Memory mountain

1 Exercise 1: Cache Visualization

本实验利用Venus 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
```

1.1 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?
- 为什么会出现这个cache命中率?
- 增加Rep Count 参数的值，可以提高命中率吗? 为什么?
- 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）？

1.2 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?
- 为什么会出现这个cache命中率?
- 增加Rep Count 参数的值，例如重复无限次，命中率是多少? 为什么?

1.3 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?
- 总共访问了L1 cache几次? L1 Miss次数为多少?
- 总共访问了L2 cache几次?
- 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?
- 如果将L1 cache中的块数增加， L1 、 L2 hit rate 有什么变化 ?
- 如果将L1 cache中的块大小增加，L1 、 L2 hit rate 有什么变化 ?

2 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:

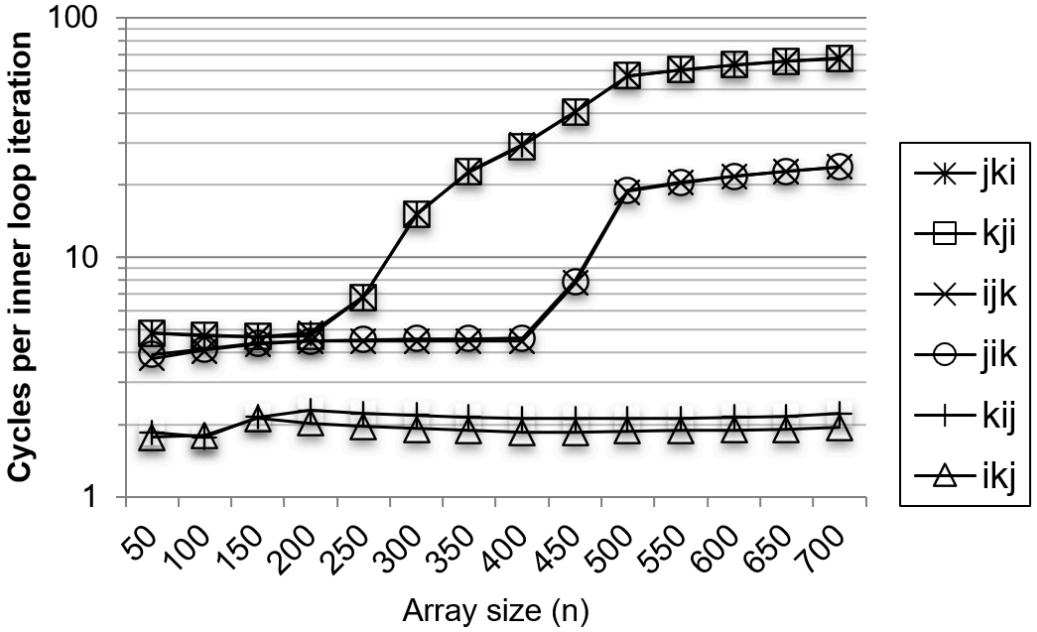
- 1000-1000的矩阵相乘，哪种嵌套顺序性能最好? 哪种嵌套顺序性能最差?
- 参考如下代码，修改matrixMultiply.c，再次观察程序的性能是否有改善（浮点运算吞吐率Gflops/s），从中你得到哪些经验?

- 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，最快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?

  以下是ijk、kij这两种嵌套的示例代码，你可以仿照他完成其他嵌套顺序的代码。

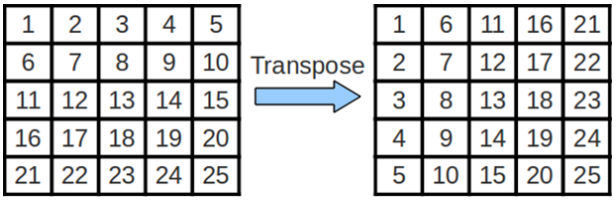
  - ```
    // ijk
    for (i = 0; i < n; i++)
        for (j=0; j< n; j++) {
            sum = 0.0;
            for (k = 0; k < n; k++)
                sum += A[i][k]*B[k][j];
            C[i][j] += sum;
        }
    ```

  - ```
    // kij
    for (k = 0; k < n; k++)
        for (i=0; i< n; i++) {
            r = A[i][k];
            for (j = 0; j < n; j++)
                C[i][j] += r * B[k][j];
        }
    ```

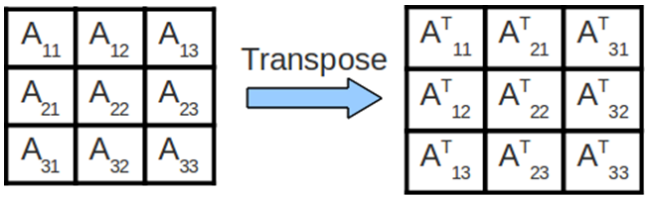
  - 

3 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。

3.1 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?

3.2 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?**

4 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
```

1. 请罗列出运行结果。
2. 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：

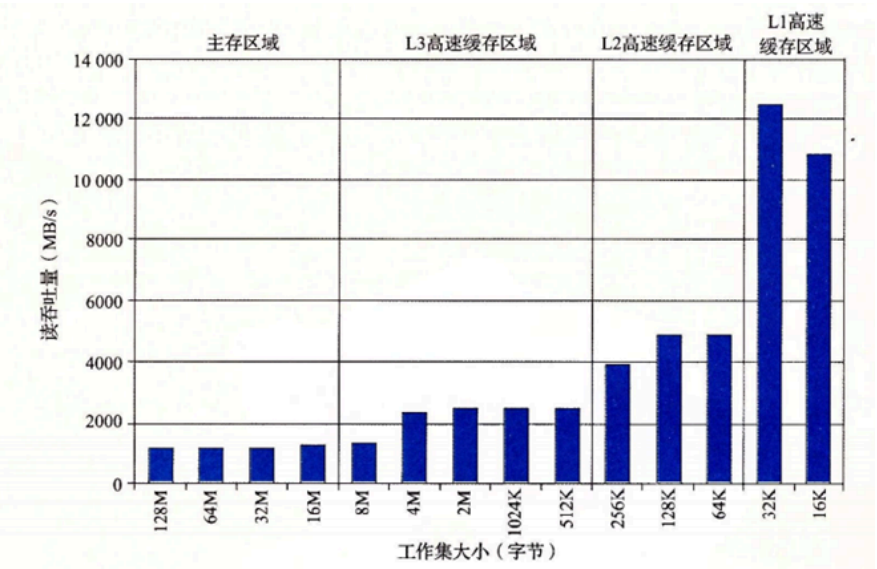


图 6-42　存储器山中时间局部性的山脊。这幅图展示了图 6-41 中 stride=8 时的一个片段

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？

3. 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。
4. 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？**

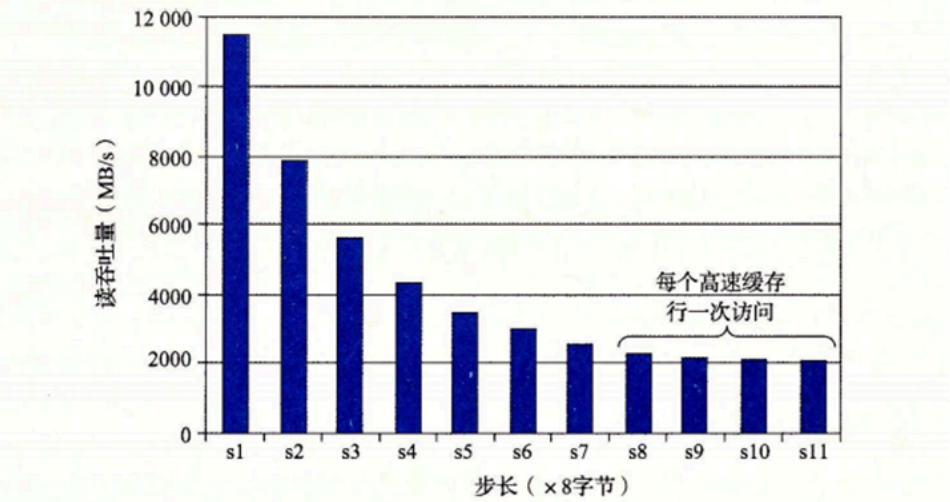


图 6-43　一个空间局部性的斜坡。这幅图展示了图 6-41 中大小=4MB 时的一个片段