

计 22

滕爽

2012011270

# LAB1 Report

计算机系统结构

2015/04/05

## 目录

|                                  |    |
|----------------------------------|----|
| 一、测量 cache 的大小.....              | 2  |
| 1、[实验原理].....                    | 2  |
| 2、[实验遇到问题].....                  | 2  |
| 3、[实验设计思路].....                  | 2  |
| 4、[实验数据分析].....                  | 3  |
| 5、[实验结果展示].....                  | 3  |
| 二、测量 Block 大小 .....              | 4  |
| 1、[实验原理].....                    | 4  |
| 2、[实验设计思路].....                  | 5  |
| 3、[实验数据分析].....                  | 5  |
| 4、[实验结果展示].....                  | 6  |
| 三、测量 cache 相连度.....              | 7  |
| 1、[实验原理].....                    | 7  |
| 2、[实验设计思路].....                  | 7  |
| 3、[实验数据分析].....                  | 7  |
| 4、[实验遇到问题].....                  | 8  |
| 5、[实验结果展示].....                  | 8  |
| 四、对所给程序 matrix_mul.cpp 进行优化..... | 9  |
| 1、[实验原理].....                    | 9  |
| 2、[实验结果展示].....                  | 10 |
| 五、测量数据缓存的写策略 .....               | 11 |
| 1、[实验原理].....                    | 11 |
| 2、[实验数据分析].....                  | 11 |
| 3、[实验结果展示].....                  | 12 |
| 六、测量数据缓存的替换策略 .....              | 12 |
| 1、[实验原理].....                    | 12 |
| 2、[实验设计思路].....                  | 12 |
| 3、[实验数据分析].....                  | 13 |
| 4、[实验结果展示].....                  | 13 |
| 七、实验代码说明 .....                   | 14 |

---

|                |    |
|----------------|----|
| 1、文件说明 .....   | 14 |
| 2、作图原始数据 ..... | 15 |
| 3、代码运行说明 ..... | 18 |

## 一、测量 cache 的大小

### 1、[实验原理]

测量 cache 的基本思想是对一段大小的数组反复读取，并且逐次增加数组的大小。当数组的大小超过缓存的大小时，频繁读取就会导致频繁替换缓存，使得吞吐量下降。所以只要测量吞吐量，观察发生突变的点，即可得到缓存的大小。

### 2、[实验遇到问题]

最新的 Intel 处理器自带硬件预读取功能，即会根据程序执行时读取内存的步长预测下一次访问，并且提前读取到缓存里。如果按照顺序访问数组的方法，则会发现吞吐量一直不会下降，或没有很明显的突变点，正是因为硬件预读取提前替换了缓存，没有影响到读取的效率。

### 3、[实验设计思路]

为了不让处理器预测出访问的步长，可以每次产生一个伪随机数作为下标访问。但是产生随机数本身就会影响程序计时，并且调用外部函数时会导致内存访问，使得之前的缓存失效。

仿照链表的实现方式，把下一次访问的地址放在这次访问的地址

所对应的变量中。每次读取内存的时候，把读到的值作为下一次访问的地址，防止硬件预读取工作。并且测量缓存大小时要适当增大步长，加大缓存替换的频率，使得结果更加明显。

## 4、[实验数据分析]

运行程序，对 1KB 到 2048KB 之间大小的数组进行测试，并且得出吞吐量。吞吐量的单位是 MB/s，但是因为使用 `clock()` 函数计时，结果并不是很准确。不过最重要的是吞吐量的相对大小，所以并不影响测量缓存的大小。

程序源代码为 `cacheSize.cpp`，程序会给出相应的吞吐量。

程序输出如图 fig2 所示，将结果画成折线图如图 fig1 所示。注意到横轴对应的是  $2^x$ KB。

从图中可以看出在 32KB 和 256KB 处有明显的吞吐量的突降，于是判断 L1、L2、L3 数据缓存的大小分别为 32KB、256KB、2048KB。

## 5、[实验结果展示]

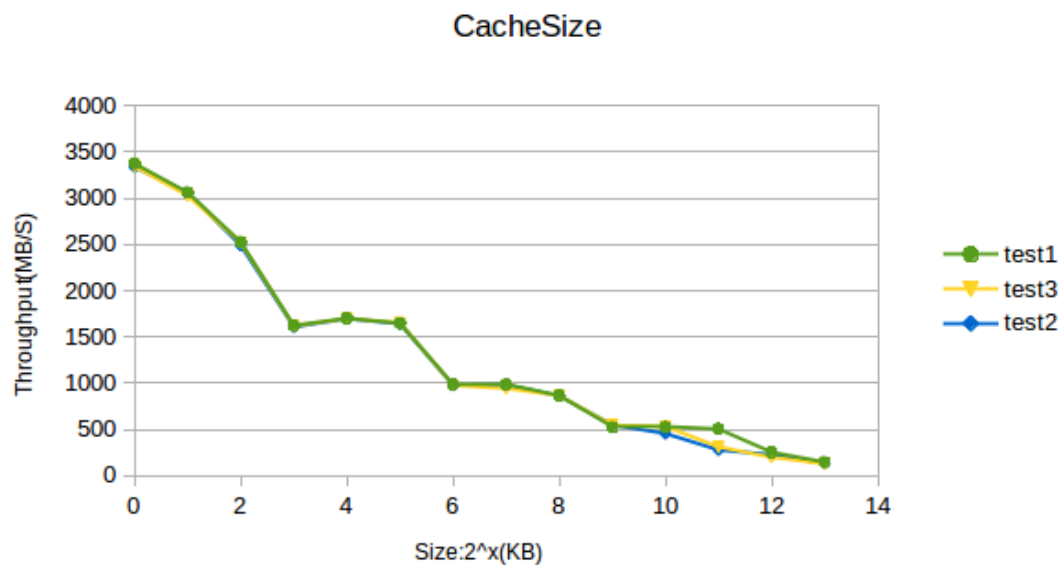


Figure 1

```
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./cacheSize
executing
1KB is being calculated!
2KB is being calculated!
4KB is being calculated!
8KB is being calculated!
16KB is being calculated!
32KB is being calculated!
64KB is being calculated!
128KB is being calculated!
256KB is being calculated!
512KB is being calculated!
1024KB is being calculated!
2048KB is being calculated!
4096KB is being calculated!
8192KB is being calculated!
```

Figure 2

## 二、测量 Block 大小

### 1、[实验原理]

对内存进行顺序访问，但是只要读到块里的某一个字节，整个块都会被缓存进来。所以如果按顺序每字节均访问，那么仅仅会在访问该块的第一个字节的时候访问更低级存储。接下来在该块内的访问会直接命中。如果不是每个字节都依次访问，加大访问的步长，可以预

见当步长等于块大小的时候，每一次读取就会要访问更低级存储，将整个块都加载进来，这样的吞吐量是最低的。

采取每次加大步长的方法，观察吞吐量将会出现先降后升的现象，并且最低点对应的步长即正好是块大小。

## 2、[实验设计思路]

注意到硬件预读取带来的影响，同样使用与第一题类似链表的数据结构进行访问。

## 3、[实验数据分析]

程序源代码为 `blockSize.cpp` 运行程序，对步长从 1 到 64 进行测试。这里的步长是指 `uint64` 的长度，即 8B。

得到程序输出如图 `fig3`，将结果画成折线图如图 `fig4`。程序前面在波动中下降，并当步长等于 8，即 64B 的时候达到吞吐量的最低值。

所以可以判断 L1 和 L2 数据缓存的块大小是 64B。

## 4、[实验结果展示]

```
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./blockSize
The 1th stride is calculated
The 2th stride is calculated
The 3th stride is calculated
The 4th stride is calculated
The 5th stride is calculated
The 6th stride is calculated
The 7th stride is calculated
The 8th stride is calculated
The 9th stride is calculated
The 10th stride is calculated
The 11th stride is calculated
The 12th stride is calculated
The 13th stride is calculated
The 14th stride is calculated
The 15th stride is calculated
The 16th stride is calculated
The 17th stride is calculated
The 18th stride is calculated
The 19th stride is calculated
The 20th stride is calculated
The 21th stride is calculated
The 22th stride is calculated
The 23th stride is calculated
The 24th stride is calculated
The 25th stride is calculated
The 26th stride is calculated
The 27th stride is calculated
The 28th stride is calculated
The 29th stride is calculated
The 30th stride is calculated
The 31th stride is calculated
The 32th stride is calculated
```

Figure 3

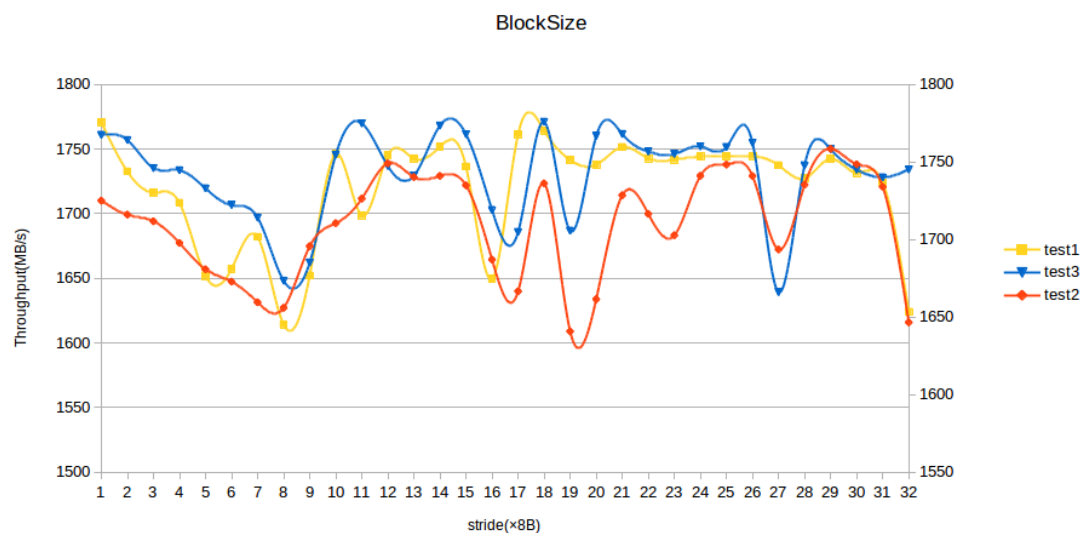


Figure 4

## 三、测量 cache 相连度

### 1、[实验原理]

已知块大小是 64B，第 1 级缓存为 32KB，2 级缓存为 256KB。下面只要测出来一共有多少个组，就能知道每个组的大小和相连度了。

### 2、[实验设计思路]

已知块大小是 64B，占了地址最低的 6 位。这里可以取一掩码，用来分割地址前面的标签和后面的索引和偏移量部分。实际上在程序实现时可以取这个掩码加 1 之后的值，每次往地址上累加。如果掩码没有盖住索引和偏移量部分，那么往地址上加的时候会改变索引，会从一个组跳到另一个组。

如果掩码正好盖住或者超过了索引和偏移量，那么往地址上累加的时候就只会改变标签，而仍然还在同一个组内。

通过枚举掩码长度，当掩码正好盖住索引和偏移量的时候，所有读取的内容都属于同一个缓存组，缓存冲突频率最大，吞吐量最低。

通过观察

吞吐量下降到极值的这个点，即可算出相连度。

### 3、[实验数据分析]

程序源代码为 `associativity.cpp` 依次左移掩码，得到程序输出如图 fig5。这里的 mask 实际上是掩码加 1 之后的值，即往地址上累加的



值。

数据如图 fig6 可以看出当掩码为 12 位的时候吞吐量最低，即可判断地址的低 12 位是索引和偏移量。已知偏移量为 6 位，所以索引为 6 位，共有 64 个不同的组。一级缓存共有 32KB，而块大小之前已经算出来是 64B，计算得到一组里有 8 个块，即是 8 相连的。

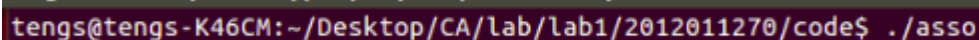
考虑到一级缓存和二级缓存有一定的一致性，猜测二级缓存也是 8 相连的。之前已经得到二级缓存共 256KB，算得一共应有 512 组。对应的掩码有 15 位，可以看到在图 fig6 中当掩码为 15 位时，吞吐率的增长趋势得到抑制，验证猜测是正确的，即二级缓存也是 8 相连的。

#### 4、[实验遇到问题]

图 fig6 后面吞吐量开始上升，猜测是因为程序采用固定读取量，测量时间得到吞吐量的方法。当掩码变大时，实际上每一步跳跃距离变大，但数组大小不变，

为了达到同样的读取量，只能多次重复读取。跳跃距离变大导致不同的地址的数量太少，可能会无法使缓存某一组填满，更不用说替换了，所以吞吐量会变高。要解决这个问题可以扩大数组的大小。

#### 5、[实验结果展示]



```
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./asso
```

Figure 5

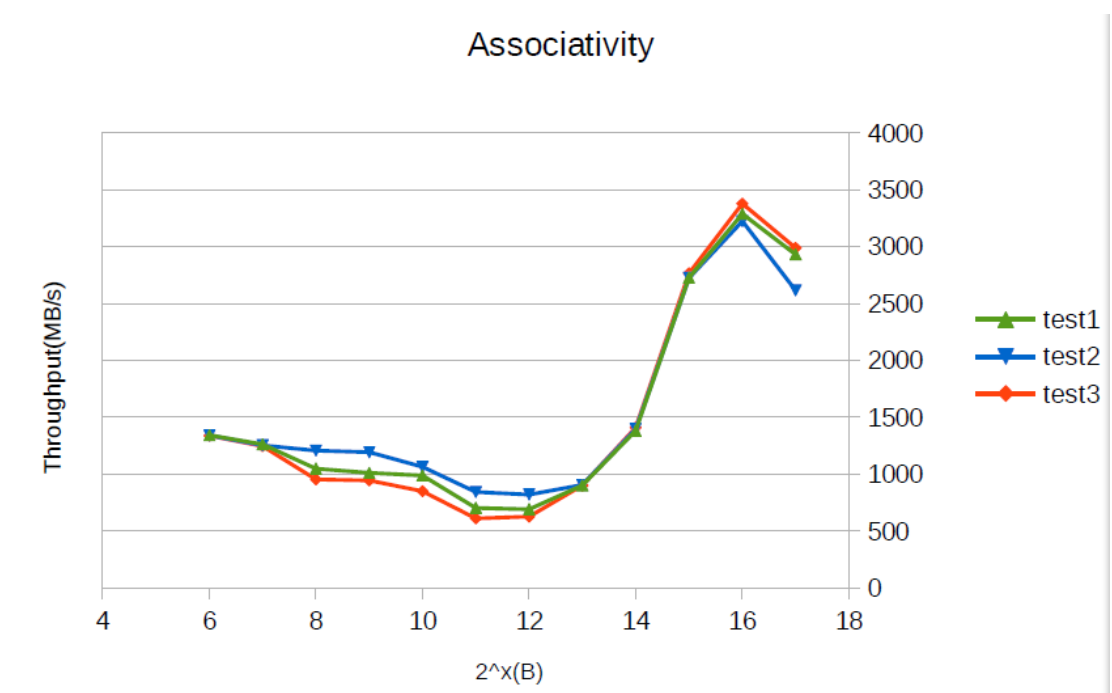


Figure 6

#### 四、对所给程序 matrix\_mul.cpp 进行优化

##### 1、[实验原理]

对于  $A \times B = C$  的矩阵乘法。对于指数的顺序，一共有三种方式。

其中  $ijk$  和  $jik$  等效，  $jki$  和  $kji$  等效，  $kij$  和  $ikj$  等效。

根据 Computer Systems A Programmer's Perspective 上的结论，有

如表 tab1 所示结论。

| Table 1: 矩阵乘法效率         |                 |                  |                    |                    |                    |                        |
|-------------------------|-----------------|------------------|--------------------|--------------------|--------------------|------------------------|
| Matrix multiply version | Loads per iter. | Stores per iter. | A misses per iter. | B misses per iter. | C misses per iter. | Total misses per iter. |
| <i>ijk</i> & <i>jik</i> | 2               | 0                | 0.25               | 1.00               | 0.00               | 1.25                   |
| <i>jki</i> & <i>kji</i> | 2               | 1                | 1.00               | 0.00               | 1.00               | 2.00                   |
| <i>kij</i> & <i>ikj</i> | 2               | 1                | 0.00               | 0.25               | 0.25               | 0.50                   |

Figure 7

所以只要采用 `kij` 或者 `ikj` 方式，就能大幅利用空间局部性提高效率。

为了进一步提高程序的运行效率，可以采用矩阵分块的思想，增大空间局部性。

`A` 为  $1000 \times 1000$  的矩阵，每个元素占 `4B`，一行为 `4000B`。

考虑到一级缓存为 `32KB`，分到 `A`，`B`，`C` 三个矩阵上，每个矩阵大概可以利用 `10KB`。所以在 `ikj` 的方式上进一步改进，每次对 `i`，`i+1`，`i+2`，`i+3` 同时操作，降低 `B` 的重复读取的次数。

这时的程序源代码为 `matrix_mul.cpp`，未分块时时间减少 `55%`，分块后程序输出如图 `fig9` 所示，时间减少 `88%`。

如果还想进一步优化，可以使用编译器的优化参数 `-O3`，得到程序输出如图 `fig9` 所示，时间减少达到 `96%`，性能提升巨大。可能因为循环次数是常数，编译器直接进行循环展开，再加上其它一些优化，才能得到这样的结果。

## 2、[实验结果展示]

```
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./matrix_mul
time spent for original method : 7211033 ms
time spent for new method : 3264503 ms
```

Figure 8

```
g++ matrix_mul.o -O3 -o matrix_mul -O3
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./matrix_mul
time spent for original method : 2976986 ms
time spent for new method : 359984 ms
```

Figure 9

## 五、测量数据缓存的写策略

### 1、[实验原理]

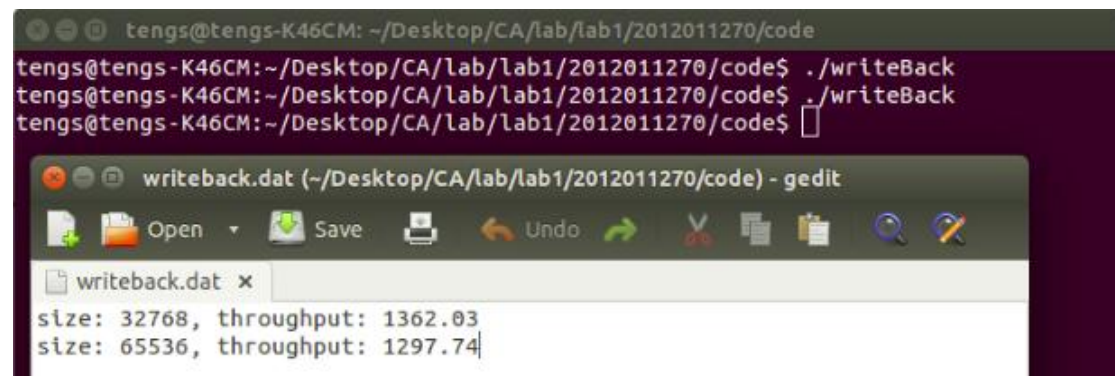
数组大小小于缓存大小时，如果是采用的写回法，则没有对低级存储的访问，而如果是采用的写直达法，则每次都会访问低级存储。但是当数组大小大于缓存大小时，无论采用哪种方法，都会对低级存储进行写操作。于是可以通过比较这两种情况下的吞吐量的差别，来测量数据缓存的写策略。

为了增大可能存在的性能上的差别，每次访问的步长正好是一个块大小。这样就能在最短时间内写脏整个数据缓存，扩大吞吐量的差距。

### 2、[实验数据分析]

程序源代码为 `writeBack.cpp`。运行得到程序输出如图 fig10 当数组大小正好为一级缓存大小时，吞吐量有 **1362.03MB/s**，而当数组大小超过一级缓存大小时，吞吐量下降到 **1297.74MB/s**，下降了 **4.7%**。多次测量均可观测到这个差距，说明数据缓存采用写回法。

### 3、[实验结果展示]



The image shows a terminal window and a gedit editor window. The terminal window displays the following commands and output:

```
tengs@tengs-K46CM: ~/Desktop/CA/lab/lab1/2012011270/code
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./writeBack
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./writeBack
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$
```

The gedit window shows the contents of the file `writeback.dat`:

```
size: 32768, throughput: 1362.03
size: 65536, throughput: 1297.74
```

Figure 10

## 六、测量数据缓存的替换策略

### 1、[实验原理]

通过巧妙地构造访问序列，来测试缓存的替换策略。这里构造两个序列，其中一个对 FIFO 有特别优化，另一个对 LRU 有特别优化，但是两者的总访问量保持相当。这样如果一个序列访问时间短，另一个访问时间长，就可以以此来判断数据缓存的替换策略。

### 2、[实验设计思路]

一个缓存组里面有 8 个块，所以一共需要 9 个块，编号为 0 到 8。

注意到它们的地址的后 12 位是相同的，即都属于同一个组。

构造出对 FIFO 有特别优化的序列是 081102213324435546657768870，对 LRU 有特别优化的序列是 080212434656878101323545767。

可以看出 FIFO 序列中有  $n$ ， $n-1$ ， $n+1$  类型元素，如果采用 LRU 会

打乱  $n$  和  $n+1$  的顺序，使得缓存替换增多。同样 LRU 序列中有  $n$ ,  $n-1$ ,  $n$  类型元素，如果采用 FIFO 则会使得第二次访问  $n$  的时候发生缓存替换。

最后需要注意在执行完一个序列之后缓存要恢复成原样，这里设为 01234567。只有这样才能反复测量，得到比较明显的结果。

### 3、[实验数据分析]

程序源代码为 `replacement.cpp`。

运行程序输出如图 fig11 看出运行 LRU 序列的时间比 FIFO 的少 50%，得到缓存采用的替换策略是 LRU。

### 4、[实验结果展示]

```
tengs@tengs-K46CM:~/Desktop/CA/lab/lab1/2012011270/code$ ./replacement
Deciding replacement strategy...
Averaged time WITH locality = 145480ns.
Averaged time WITHOUT locality = 291105ns.
LRU IS used
```

Figure 11

## 七、实验代码说明

### 1、文件说明

如下表所示

| 文件名               | 说明                 | 生成中间文件          | 可执行文件         | 数据文件              |
|-------------------|--------------------|-----------------|---------------|-------------------|
| ele.cpp           | 被以下文件调用的基类         | ele.o           |               |                   |
| ele.h             |                    |                 |               |                   |
| cacheSize.cpp     | 测量 cache 的大小       | cacheSize.o     | cacheSize     | cacheSize.dat     |
| blockSize.cpp     | 测量 block 的大小       | blockSize.o     | blockSize     | blockSize.dat     |
| associativity.cpp | 测量相连度              | associativity.o | associativity | associativity.dat |
| writeback.cpp     | 测量写策略              | writeback.o     | writeback     | writeback.dat     |
| replacement.cpp   | 测量替换策略             | replacement.o   | replacement   | replacement.dat   |
| matrix_mul.cpp    | 根据存储器结构优化代码        | matrix_mul.o    | matrix_mul    | matrix_mul.dat    |
| makefile          | 用于 linux/g++ 运行的文件 |                 |               |                   |
| picture           | 文件夹放置图片            |                 |               |                   |

## 2、作图原始数据

如下表所示

1) cacheSize

| 1    |     |            | 2    |     |            |
|------|-----|------------|------|-----|------------|
| size | log | throughput | size | log | throughput |
| 1    | 0   | 3376.23    | 1    | 0   | 3338.48    |
| 2    | 1   | 3064.38    | 2    | 1   | 3058.2     |
| 4    | 2   | 2527.55    | 4    | 2   | 2490.82    |
| 8    | 3   | 1624.02    | 8    | 3   | 1607.03    |
| 16   | 4   | 1701.73    | 16   | 4   | 1697       |
| 32   | 5   | 1648.24    | 32   | 5   | 1641.82    |
| 64   | 6   | 983.132    | 64   | 6   | 976.255    |
| 128  | 7   | 984.797    | 128  | 7   | 982.879    |
| 256  | 8   | 866.354    | 256  | 8   | 861.716    |
| 512  | 9   | 522.224    | 512  | 9   | 549.793    |
| 1024 | 10  | 523.783    | 1024 | 10  | 454.625    |
| 2048 | 11  | 504.735    | 2048 | 11  | 278.137    |
| 4096 | 12  | 253.016    | 4096 | 12  | 226.831    |
| 8192 | 13  | 146.443    | 8192 | 13  | 129.922    |

| 3    |     |            |
|------|-----|------------|
| size | log | throughput |
| 1    | 0   | 3344.87    |
| 2    | 1   | 3027.8     |
| 4    | 2   | 2522.04    |
| 8    | 3   | 1620.28    |
| 16   | 4   | 1698.2     |
| 32   | 5   | 1654.64    |
| 64   | 6   | 978.098    |
| 128  | 7   | 942.883    |
| 256  | 8   | 865.886    |
| 512  | 9   | 545.694    |
| 1024 | 10  | 531.034    |
| 2048 | 11  | 311.875    |
| 4096 | 12  | 198.748    |
| 8192 | 13  | 127.301    |

Figure 12



## 2) blockSize

| 1      |            | 2      |             | 3      |             |
|--------|------------|--------|-------------|--------|-------------|
| stride | throughput | stride | throughput  | stride | throughput  |
| 1      | 1770.69    | 1      | 1725.017983 | 1      | 1767.472126 |
| 2      | 1732.59    | 2      | 1716.055636 | 2      | 1764.060371 |
| 3      | 1716.31    | 3      | 1711.78848  | 3      | 1746.05103  |
| 4      | 1708.37    | 4      | 1697.788647 | 4      | 1744.589685 |
| 5      | 1651.8     | 5      | 1680.81848  | 5      | 1732.944605 |
| 6      | 1657.35    | 6      | 1672.856836 | 6      | 1722.219746 |
| 7      | 1681.96    | 7      | 1659.562744 | 7      | 1714.141036 |
| 8      | 1614.22    | 8      | 1655.972518 | 8      | 1673.410424 |
| 9      | 1652.27    | 9      | 1695.744658 | 9      | 1685.193094 |
| 10     | 1746.26    | 10     | 1710.515326 | 10     | 1754.585869 |
| 11     | 1698.73    | 11     | 1726.354299 | 11     | 1774.8482   |
| 12     | 1745.19    | 12     | 1749.08265  | 12     | 1747.375417 |
| 13     | 1742.65    | 13     | 1740.246167 | 13     | 1741.048752 |
| 14     | 1751.72    | 14     | 1740.99399  | 14     | 1773.428107 |
| 15     | 1736.33    | 15     | 1734.919101 | 15     | 1767.860418 |
| 16     | 1649.58    | 16     | 1687.090095 | 16     | 1719.03296  |
| 17     | 1761.42    | 17     | 1666.625299 | 17     | 1704.870869 |
| 18     | 1764.17    | 18     | 1736.196475 | 18     | 1775.833918 |
| 19     | 1741.65    | 19     | 1640.804763 | 19     | 1705.712889 |
| 20     | 1737.78    | 20     | 1661.519715 | 20     | 1766.693768 |
| 21     | 1751.49    | 21     | 1728.544275 | 21     | 1767.968774 |
| 22     | 1742.79    | 22     | 1716.536034 | 22     | 1756.665013 |
| 23     | 1741.82    | 23     | 1702.763082 | 23     | 1755.140728 |
| 24     | 1744.19    | 24     | 1741.065034 | 24     | 1760.036094 |
| 25     | 1744.39    | 25     | 1748.277119 | 25     | 1759.592269 |
| 26     | 1744.58    | 26     | 1740.926652 | 26     | 1762.308624 |
| 27     | 1737.39    | 27     | 1693.620146 | 27     | 1666.020623 |
| 28     | 1727.3     | 28     | 1735.310129 | 28     | 1747.83995  |
| 29     | 1742.43    | 29     | 1758.397248 | 29     | 1758.100633 |
| 30     | 1731.55    | 30     | 1748.480858 | 30     | 1744.971704 |
| 31     | 1724.29    | 31     | 1733.966525 | 31     | 1740.059126 |
| 32     | 1624.15    | 32     | 1646.60073  | 32     | 1745.159802 |

Figure 13

## 3) associativity

| 1   |        |            | 2   |        |            |
|-----|--------|------------|-----|--------|------------|
| log | mask   | throughput | log | mask   | throughput |
| 6   | 64     | 1338.04    | 6   | 64     | 1342.5     |
| 7   | 128    | 1245.99    | 7   | 128    | 1262.18    |
| 8   | 256    | 953.928    | 8   | 256    | 1047.7     |
| 9   | 512    | 944.047    | 9   | 512    | 1009.9     |
| 10  | 1024   | 849.834    | 10  | 1024   | 988.244    |
| 11  | 2048   | 610.892    | 11  | 2048   | 702.592    |
| 12  | 4096   | 625.611    | 12  | 4096   | 691.408    |
| 13  | 8192   | 899.051    | 13  | 8192   | 900.424    |
| 14  | 16384  | 1408.49    | 14  | 16384  | 1380.01    |
| 15  | 32768  | 2765.53    | 15  | 32768  | 2727.68    |
| 16  | 65536  | 3378.24    | 16  | 65536  | 3288.5     |
| 17  | 131072 | 2988.68    | 17  | 131072 | 2932.27    |

| 3   |        |            |
|-----|--------|------------|
| log | mask   | throughput |
| 6   | 64     | 1340.47    |
| 7   | 128    | 1252.69    |
| 8   | 256    | 1206.42    |
| 9   | 512    | 1191.31    |
| 10  | 1024   | 1063.61    |
| 11  | 2048   | 843.279    |
| 12  | 4096   | 820.332    |
| 13  | 8192   | 904.308    |
| 14  | 16384  | 1395.73    |
| 15  | 32768  | 2725.54    |
| 16  | 65536  | 3224.03    |
| 17  | 131072 | 2616.67    |

Figure 14

### 3、代码运行说明

运行环境为 **Ubuntu 14.10 + g++**

shell 指令：

make：进行编译运行所有 cpp 文件

make cacheSize

make blockSize

make asso

make matrix\_mul

make writeBack

make replacement

分别编译运行每个部分。