Computer Architecture, Spring 2021

实验一：Cache 测量实验

清華大學
Tsinghua University

目录

1 实验信息

2 实验目的&内容

3 实验背景

4 实验步骤

5 实现细节和可选部分

6 评价指标

实验信息

- 实验日期：第四周周四（2021.3.18）~ 第六周周日（2021.4.4）
- 实验环境
  - x86（x86-64）架构的处理器
  - Linux 系统或虚拟机，建议使用 C/C++ 编程。
  - 搭建实验环境有困难的同学可以联系助教：

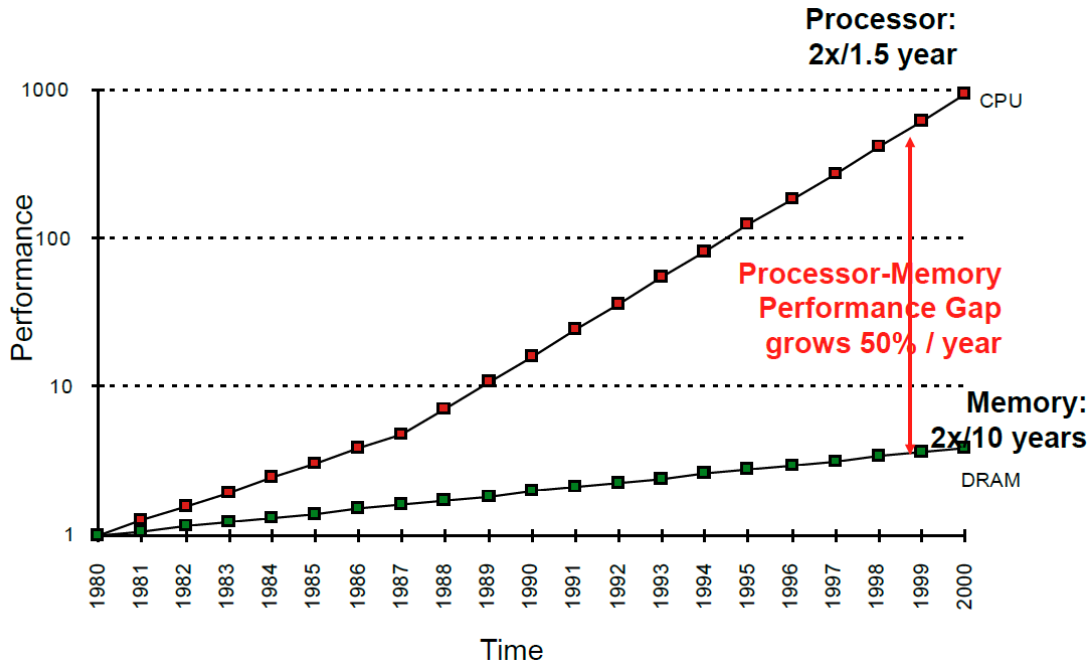
    ylf17@mails.tsinghua.edu.cn； chang-li17@mails.tsinghua.edu.cn
- 实验占比
  - 满分10分，占课程总评的 10% 。对于DDL之后的补交作业，第一周内补交(4.5~4.11)至多8分，第二周内补交(4.12~4.18)至多5分，两周后不得分。

实验目的 & 内容

- 实验目的
  - 了解自己机器的 Cache 配置，让自己机器的 Cache 不再“透明”
  - 利用 Cache 和局部性原理优化程序，从而加深对 Cache 工作原理的理解
- 实验内容
  - 构造测试用例，通过测量执行时间验证自己机器的 L1 和 L2 Cache 大小
  - 构造测试用例，通过测量执行时间验证自己机器的 L1 Cache Line 大小
  - 构造测试用例，通过测量执行时间验证自己机器的 L1 Cache 的相联度
  - 利用自己机器的 Cache 配置完成一个矩阵乘法代码的优化

背景

- 为什么要使用 Cache
  - 计算机性能的瓶颈：内存访问的延迟远大于处理器的时钟周期
  - 局部性原理：时间局部性和空间局部性
- 设计 Cache 时通常需要考虑的问题
  - Cache 规模
  - Cache Line 规模
  - Cache 相联度
  - 写策略
  - 替换策略
  - 一致性 ……



背景

- Cache Size
  - 常见的 Cache 组织方式：
    - Private L1 Data Cache；Private L1 Instruction Cache;
    - Private L2 Cache; Shared L3 Cache
  - Windows 查看 Cache Size
    - 整体：任务管理器，或者 wmic 命令
    - 细节：利用 CPU-Z 工具
  - Linux 查看 Cache Size
    - 整体：lscpu 命令
    - 细节：Linux 设备文件，如 /sys/devices/system/cpu/cpu0/cache

背景

- Cache Line
  - 利用程序的空间局部性
  - 不同层级 Cache 的 Cache Line Size 未必相同
    - Intel Core i7/i5/i3 对于每个层级都使用 64B 的 Cache Line Size
    - Intel Pentium 4 使用 64B 的 L1 Cache Line Size 和 128B 的 L2 Cache Line Size
  - #Cache Line（Cache Line 个数） = Cache Size / Cache Line Size
  - Cache Line 是处理器在 Cache Miss 时填充的基本单位，但有的处理器在 L2 或 L3 层会一次性填充多个 Cache Line

背景

- Ways of Associativity
  - 一个 Set 中的 Cache Line 个数
  - 三种相联方式：
    - 全相联：#set = 1; ways of associativity = #cache line
    - 直接相联：#set = #cache line
    - 多路组相联：1 < #set < #cache line;
      1 < ways of associativity < #cache line

实验步骤

- 第0步：使用 PPT 第 5 页提到的方式了解自己机器的 Cache 参数。
- 第1步：测量 L1 Data Cache Size 和 L2 Cache Size
- 第2步：测量 L1 Data Cache Line Size
- 第3步：测量 L1 Data Cache 的相联度
- 第4步：利用 Cache 的工作方式实现矩阵乘优化

1. 测量 Cache Size

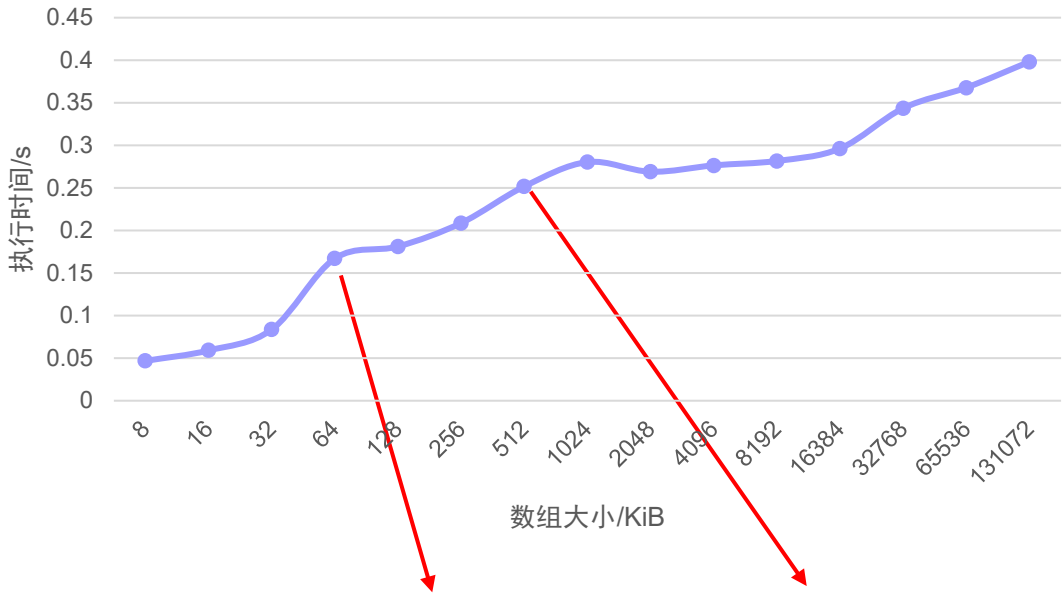
- 基本思路：从内存读取连续数组中的数据，观察平均读取速度。改变数组大小和访问序列做多次尝试。
  - 当数组大小超过 L1D Cache Size 后，会出现 Cache 读缺失，平均读取速度会有一个突然的增加
  - 同理，当数组大小超过 L2 Cache Size 后，平均读取速度也会有一个突然的增加
- 访问序列的步长需要稍微大一些。
- 需要保证对于测试数组规模变化后，总体访问次数相同。
- 可以使用最简单的数组循环访问模式。当然也可以构造随机访问序列。

1. 测量 Cache Size

- 某台机器的某次实验结果示例

  (L1D Cache Size = 32 KiB，L2 Cache Size = 256 KiB)

```
test size = 8 KB, finish - start = 0.046856 s
test size = 16 KB, finish - start = 0.059107 s
test size = 32 KB, finish - start = 0.083651 s
test size = 64 KB, finish - start = 0.167041 s
test size = 128 KB, finish - start = 0.181165 s
test size = 256 KB, finish - start = 0.208699 s
test size = 512 KB, finish - start = 0.251703 s
test size = 1024 KB, finish - start = 0.280279 s
test size = 2048 KB, finish - start = 0.268928 s
test size = 4096 KB, finish - start = 0.276383 s
test size = 8192 KB, finish - start = 0.281609 s
test size = 16384 KB, finish - start = 0.296222 s
test size = 32768 KB, finish - start = 0.343518 s
test size = 65536 KB, finish - start = 0.367744 s
test size = 131072 KB, finish - start = 0.398132 s
```

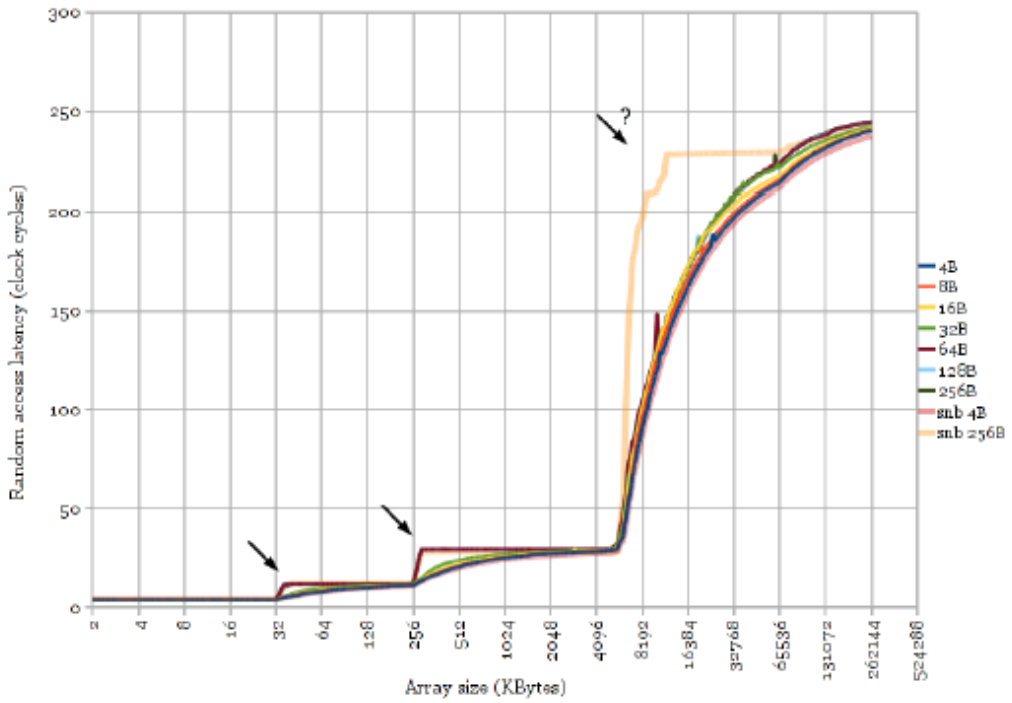


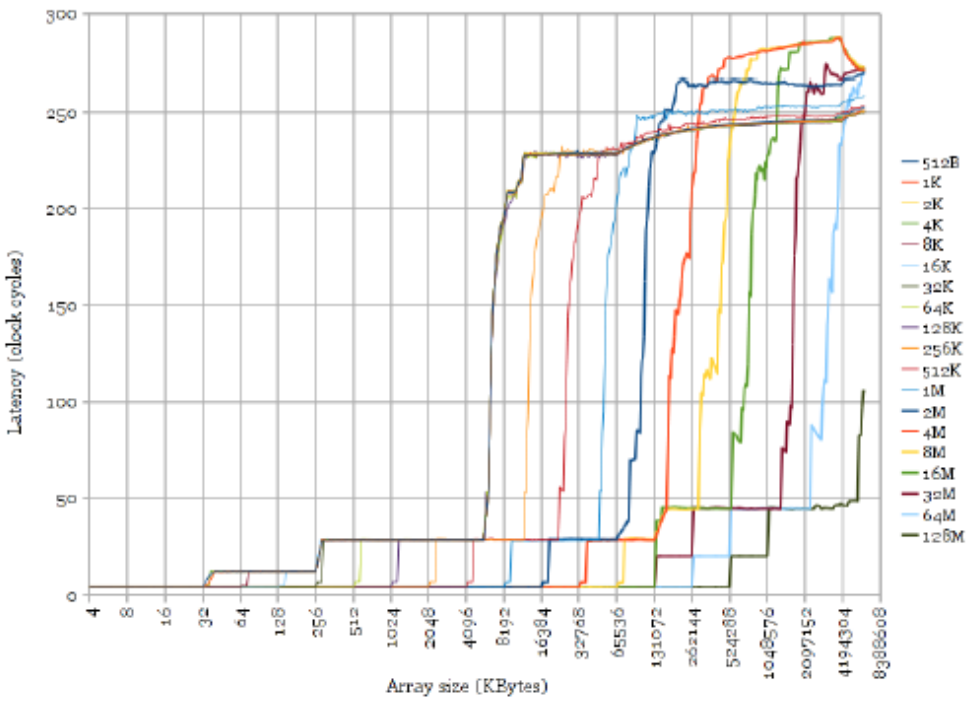
数组大于 32KiB 后执行时间明显增大

数组大于 256KiB 后执行时间也有较明显增大

1. 测量 Cache Size

- 不同数组大小和访问步长对执行时间测量结果的影响如下图





2. 测量 L1D Cache Line Size

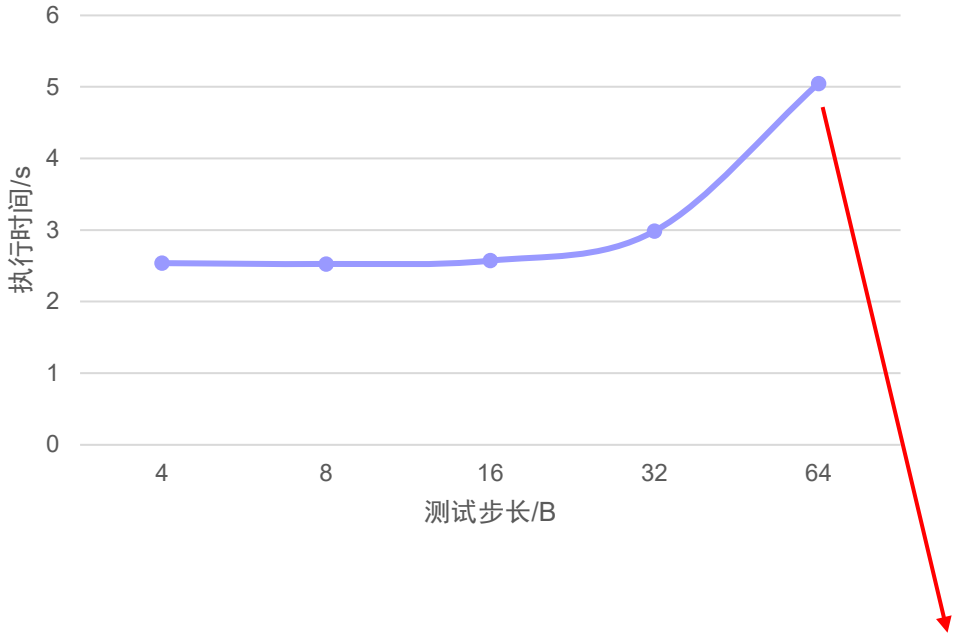
- 基本思路：使用不同步长对大于 L1D Cache Size 的某个数组做相同次数的访问。
  - 当访问数组中的元素时，如果是连续访问，因为 Cache Line 的第一个字节缺失后，会将整个 Cache Line 移入 Cache，因此后续访问的命中率会很高。
  - 如果访问是间断的，对数组间隔顺序访问，命中率就会降低，平均访问延迟增大。当间隔达到一定的大小，即超过 Cache Line Size，将造成每次都缺失的最坏情况，平均访问延迟达到最大。
- 使用数组循环访问模式即可。

2. 测量 L1D Cache Line Size

- 某台机器的某次实验结果示例

  (L1D Cache Line Size = 64B)

```
test block size = 4 B, finish - start = 2.538046 s
test block size = 8 B, finish - start = 2.526867 s
test block size = 16 B, finish - start = 2.572923 s
test block size = 32 B, finish - start = 2.984372 s
test block size = 64 B, finish - start = 5.047482 s
```



步长为64B时执行时间明显增大，说明Cache 缺失率变大

3. 测量 L1D Cache 的相联度

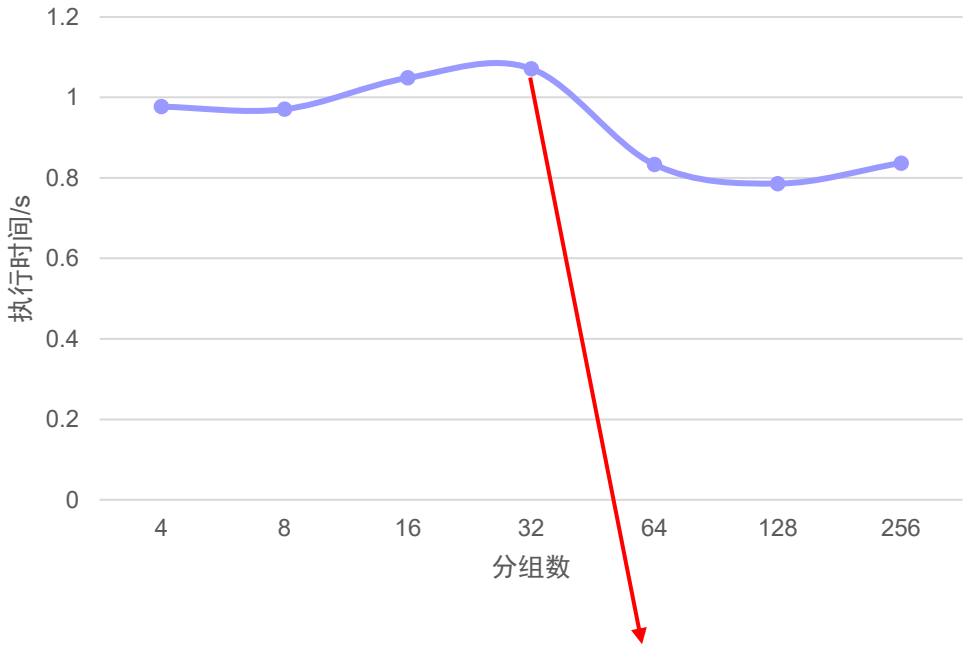
- 可以使用以下算法：
  - 使用一个 2 倍 Cache Size 大小的数组
  - 将数组分为 2^n 块，只访问其中的奇数块
  - 逐渐增大 n 的取值，当某一次访问时间变慢时，2^{n-2} 就是相联度
- 请在实验报告中给出你使用的算法的解释（举例分析或者给出证明）。
- 注意控制变量，比如对于不同的 n 需要让总的访存次数一致。
- 实验容易受到 L1D Cache 替换策略，高层级 Cache 的影响，所以结果有可能不明显。如果测量结果不明显，在实验报告中给出自己的分析即可。

3. 测量 L1D Cache 的相联度

- 某台机器的某次实验结果示例

  (L1D Cache 相联度为 8)

```
test number of groups = 4, finish - start = 0.976846 s
test number of groups = 8, finish - start = 0.970444 s
test number of groups = 16, finish - start = 1.048350 s
test number of groups = 32, finish - start = 1.071333 s
test number of groups = 64, finish - start = 0.832903 s
test number of groups = 128, finish - start = 0.785498 s
test number of groups = 256, finish - start = 0.836780 s
```



n = 5，即分成32组时访存时间相对较大，所以可估计相联度为$2^{n-2} = 8$

4. 矩阵乘优化

- 优化代码 & 优化方法 & 限制条件
  - 待优化代码：矩阵乘法（见右图）

```
for (i = 0; i < 1000; i ++)
    for (j = 0; j < 1000; j ++)
        for (k = 0; k < 1000; k ++)
            c[i][j] += a[i][k] * b[k][j];
```

  - 优化方法示例
    - 利用 Cache 参数，对矩阵分块，使 Cache 可容纳分块后的矩阵
    - 利用矩阵性质和矩阵内容修改运算顺序
  - 一些限制
    - 关闭 gcc 优化（如软件预取等）
    - 不能使用汇编指令进行优化（如使用内联汇编等）
    - 不能使用并行化进行优化（如使用 openmp等）
    - **不能修改非优化部分的代码**

4. 矩阵乘优化

- 会进行正确性检查，**计算结果错误不得分。**

```
for(i = 0; i < 1000; i++) {
    for(j = 0; j < 1000; j++) {
        if (c[i][j] != d[i][j]) {
            cout << "you have got an error in algorithm modification!" << endl;
            exit(1);
        }
    }
}
```

- 某台机器的某种优化结果如下：

```
time spent for original method : 2.18416 s
time spent for new method : 1.05159 s
time ratio of performance optimization : 2.07701
```

一些实现细节（重要）

- 进程绑定
  - 通常情况下，每个 CPU 拥有自己的 L1 Cache。因此在测试前需要把进程绑定在指定的 CPU 上，缓解操作系统的进程调度带来的影响。
  - 参考实现

```c
#define _GUN_SOURCE
#include <sched.h>

cpu_set_t mask;
CPU_ZERO(&mask);
CPU_SET(10, &mask);
if (sched_setaffinity(0, sizeof(mask), &mask) < 0) {
    perror("sched_setaffinity");
}
```

  - 使用 gcc 编译需要加上编译选项 -D_GNU_SOURCE

一些实现细节（重要）

- 计时方式
  - 单次访存延迟太小，所以每次实验需要保证足够的访存次数
  - 可以使用 clock 函数计时

```
#include <time.h>

clock_t start = clock();

/* access pattern */

clock_t finish = clock();
```

  - 多核环境下**不建议**使用 rdtsc 指令或者 PMU 计数器计时

一些实现细节（重要）

- 访存相关
  - x86-64 架构处理器的 CPU 和 L1 Cache 之间还存在一些 Buffer 优化访存操作，其中 store 受影响比 load 小得多。所以**访存时建议使用 store 操作**，比如往数组写入一个立即数。 `array[j] = 0;`
  - 对于需要计时的代码段内部的中间变量，建议使用 **register 关键字**约束以减少无关访存操作。 `register int tmp;`
  - 如果想使用 load 操作，但担心被编译器优化，可以使用内联汇编完成，如下图示例。可以使用 objdump 把编译后的可执行文件反汇编，查看汇编代码。

```
asm volatile("movl %1, %%edx\n\t"
     "movl %%edx, %0"
     :"=r"(tmp)  /* output */
     :"m"(array[j])       /* input */
     :"%edx"); /* clobbered register */
```

可选部分一

- 使用大页映射减小 TLB miss 造成的影响。实现过程如下：
  - 1. 查看内核对大页的支持

    查看内核配置文件 /proc/config.gz 中 CONFIG_HUGETLB_PAGE 和 CONFIG_HUGETLBFS 是否开启。
  - 2. 配置大页

    给某个根目录配置大页，使该目录下的文件操作都使用 2MiB 大页（xxx为某个目录名，不是文件名。需要在root权限下完成操作）。

    mkdir /mnt/xxx

    mount none /mnt/xxx -t hugetlbfs
  - 3. 分配空闲大页

    XX 为分配的大页数量。需要在root权限下完成操作。

    echo XX > /proc/sys/vm/nr_hugepages

可选部分一

- 使用大页映射减小 TLB miss 造成的影响。实现过程如下：
  - 4. 代码中用内存映射代替内存分配操作

使用 mmap 函数完成内存映射。注意映射空间大小需要小于第三步分配的空闲大页的大小。后续的访存操作全部基于这个内存空间进行。

```
int fd= open("/mnt/wsl/huge/temp", O_CREAT | O_RDWR, 0755);
if (fd < 0) {
    perror("cannot open huge page");
}

int* array = (int*) mmap(0, ALLOCATE_SIZE * sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

可选部分二

- 测出 L1D Cache 的 替换策略
  - 需针对不同的替换策略进行有针对性的区分（如 LRU，LFU 等），分别构造特殊的访存序列进行测量与验证
  - 可参考网上资料，以及本机的 Cache 配置
- 测出 L1D Cache 是否使用写直达策略
  - 写直达策略在写命中和写不命中时的访问延迟相似
  - 写会策略在写命中时的访问延迟明显小于写不命中
  - 一些缓冲区（Buffer）可能对实验结果造成影响

评价指标

- 10 (+ 1) = 2 + 1 + 2 + 2 + 1 + 2 (+ 1)
  - Cache Size (2’)：在实验报告中给出访存序列，测量程序的执行结果，以及对结果的简单分析。
  - Cache Line Size (1’)：在实验报告中给出访存序列，测量程序的执行结果，以及对结果的简单分析。
  - Cache 相联度 (2’)：在实验报告中给出访存序列，测量程序的执行结果，以及对结果的简单分析。
  - 矩阵优化 (2’)：按优化效果给分。除了代码中注明可修改的部分外，其余部分不可以修改。给分方式为：

$$\max\{0, 0.5 \times \min[3.5 * (\frac{\text{原算法耗时}}{\text{优化后算法耗时}} - 1), 4)]\}.$$

评价指标

- 10 (+ 1) = 2 + 1 + 2 + 2 + 1 + 2 (+ 1)
  - 代码 (1’): 风格、注释等，只要提交自己的代码，没有明显的抄袭痕迹，基本给全。
  - 文档 (2’): 除了之前所述的实验结果与分析外，文档还需要包括

    ① 机器的 Cache 参数； ② 相联度算法的分析；③如有选做，给出选做部分的实现思路和结果分析；④ 对本次实验的意见和建议。根据实验难度、自己的收获等，给实验评一个等级（A，B，C，D）。
  - 可选部分一不加分，感兴趣的同学可以尝试。
  - 可选部分二中，验证两种替换策略可以加0.5分；测出 L1D Cache 是否使用写直达策略可以加0.5分。

参考资料

- http://igoro.com/archive/gallery-of-processor-cache-effects/