

矩阵转置

完成了对 cache 的模拟编写，相信你对 cache 以及整个 CPU 的内存访问有了更为直接和深入的理解，接下来让我们尝试进一步的应用。假设我们已经拥有一个功能正常，规格 $s = 5$ 、 $E = 1$ 、 $b = 5$ ，映射方式为直接映射的 cache。不再编写 cache 模拟器，相反，我们将对一个函数进行编写，它的功能是实现矩阵的转置，其会在上述规格的 cache 上模拟运行。

即使实现的是同一功能，不同的程序也会有不同的内存访问顺序，从而会产生不同的 cache 命中率和缺失率。因此，编写更为合适的程序以达到更低的 cache 缺失率，便是本次任务的主要目标。

任务分析

矩阵转置

在正式开始前，让我们先对矩阵转置 (transpose) 这一概念进行辨析：设 A 代表一个矩阵， A_{ij} 表示矩阵 A 第 i 行第 j 列的元素， A 的转置，即 A^T 满足对任意符合条件的 i, j ，都有 $(A^T)_{ji} = A_{ij}$ 。

Matrix A

| | | | |
|----|----|----|----|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

Matrix B

| | | | |
|---|---|----|----|
| 1 | 5 | 9 | 13 |
| 2 | 6 | 10 | 14 |
| 3 | 7 | 11 | 15 |
| 4 | 8 | 12 | 16 |

我们将在 C 语言编程过程中采取二维数组的形式表示矩阵。将二维数组 A 转置后的结果储存到二维数组 B 中，便是我们需要完成的核心功能。

回顾：内存访问

本题中主要涉及到的内存访问其实就是在对这两个二维数组进行读写时的访问。在理论课中我们曾学习到，由于存储访问的局部性特征，不同的内存访问顺序会带来较大的效率差异。如下：

❖ 程序示例

程序A

```
int sumarrayrows(int a[M][N])
{
    Int i, j, sum=0;

    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            sum += a[i][j];
    return sum;
}
```

程序B

```
int sumarraycols(int a[M][N])
{
    Int i, j, sum=0;

    for (j=0; j<N; j++)
        for (i=0; i<M; i++)
            sum += a[i][j];
    return sum;
}
```

地址

内容

0X0000400

A[0][0]

0X0000404

A[0][1]

.....

0x00023FC

A[0][2047]

0X0002400

A[1][0]

0X0002404

A[1][1]

.....

0X00043FC

A[1][2047]

.....

.....

0X0FFE400

A[2047][0]

0X0FFE404

A[2047][1]

.....

0X10003FC

A[2047][2047]

0X1000400

sum

存储空间 (M=N=2048)

上图两个程序中，程序 A 对数组 a 的访问具有连续性，而程序 B 不具有，因而程序 A 的运行效率要比程序 B 快得多。

而在学习过 cache 之后我们知道，cache 中保存的有当前最活跃（被频繁访问）的程序和数据。大多数情况下，CPU 能直接从 cache 中取得指令和数据，而不必访问主存。不同的访问方式影响着 cache 的命中率，进而影响着整个程序运行的效率。这也是为什么上述两段程序的效率会有较大差异的原因。

分析：未命中数

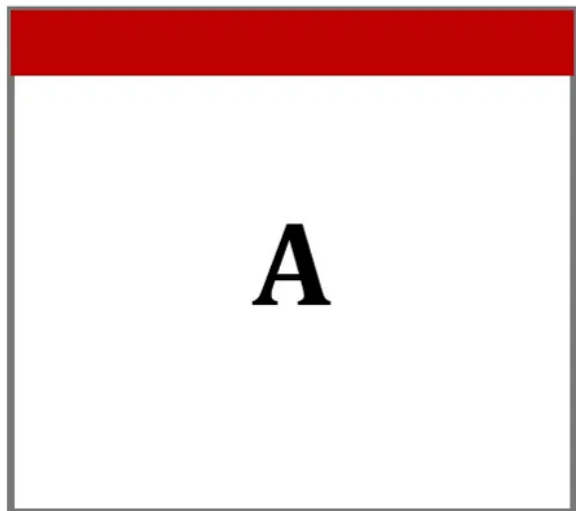
对本题而言，我们以下面的程序为例，对整个 cache 的访问过程进行更为深入的分析，并尝试计算 cache 的未命中数。

```
void transpose_example(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, tmp;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
    }
}
```

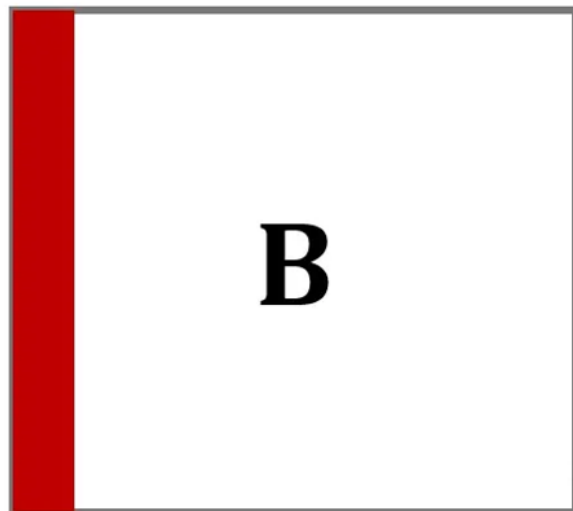
该函数准确实现了将矩阵 A 转置后的结果存储到矩阵 B 这一功能，但通过测试发现该函数命中率低，访问效率低下。那么具体运行时的未命中次数是多少呢？

我们假设 $M = N = 32$ ，该函数采取的方式是按行读取 A 矩阵，并一列一列写入 B 矩阵。

32×32



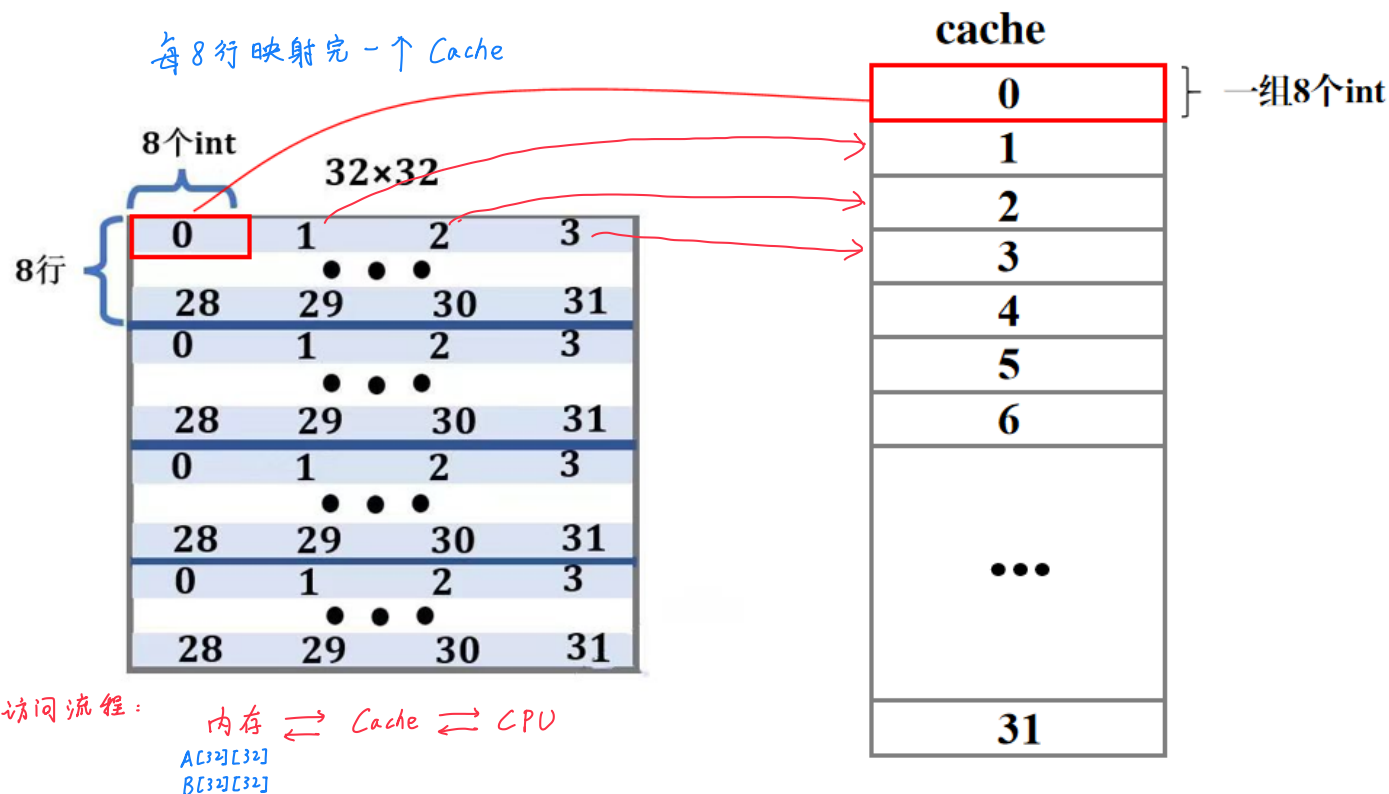
32×32



1个int占据4个字节 | 偏移量 $2^b = 32$ 字节 \Rightarrow 每组8 int

$s = 5, E = 1, b = 5$ 规格的 cache 一共有 32 组，每组一行，每行存 8 个 int。那么当我们读入 $A[0][0]$ 时，其后续的 $A[0][1] \sim A[0][7]$ 也都被读入 cache。但内容写入 B 时是一列一列的写入，列上相邻的元素并不在一个内存块上，这样每次写入 B 都不命中缓存。并且当某一列写入结束后，原本的缓存可能也被覆盖掉了，这样就又不会命中。

具体定量分析是怎样的呢？我们知道，给数组分配的内存空间是连续的，同时我们的 cache 采取的映射方式是直接映射。那么对于数组 A 和 B，它们的每一个元素映射到 cache 的哪一组是固定的。我们可以假设矩阵的前 8 个 int 正好映射到 cache 的第 0 组，那么整个矩阵的映射情况如下图，后续的分析都将在此基础上进行！



通过计算可以得到，对于 32×32 类型的矩阵，每 1 行会占用 cache 的 4 个组，每隔 8 行其对应的 cache 缓存组就会重复，图中数字代表的就是 cache 的组数。

对于矩阵 A，读取矩阵 A 时每读取 8 个 int (也就是一行 cache) 后需要重新读取，因此每读取矩阵 A 的一行 (32 个 int) 都会有 $32 / 8 = 4$ 次缺失。

对于矩阵 B，写入 1 列的元素每一次都不会命中，一共 32 次。另外，例如当我们结束写入 B 的第 1 列最后 1 个元素后返回来开始写入 $B[0][1]$ 时，此时 cache 第 0 组储存的内容早已不是 $B[0][0] \sim B[0][7]$ ，因此依旧未命中，可以说 B 的每一次写入都没有命中。由此一共有 $4 \times 32 + 32 \times 32 = 1152$ 次未命中。

思考题

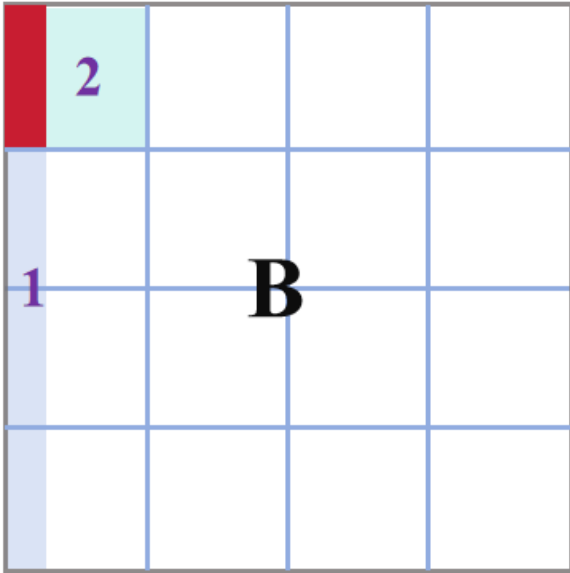
在阅读完“测试说明”部分后，尝试对该转置方法进行实际测试，观察未命中数是多少？为什么会比 1152 多？

(提示：考虑对角线上的数据进行交换时的情况)

分析：矩阵分块

那么我们究竟该怎样“编写更为合适的程序以达到更低的 cache 缺失率”呢？我们依旧以 32×32 矩阵为例进行解答。当然，该题目的解法可能并不唯一，请在阅读过程中结合上述教程进行思考，并保持专注和质疑的心态！

首先对矩阵分块这一方法进行介绍，这将是完成本题目的重要方法。



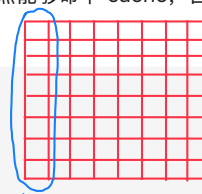
我们刚刚谈到，在将 B 的前几行读入 cache 并对第一列元素进行处理时，区域 2 中的元素也被读入，但我们选择的方式是**按列处理**，因此转而去处理区域 1 中的元素。当回过头来处理区域 2 中的元素时，发现又需要重新读入 cache 中。如果我们能够在拥有区域 2 的时候处理区域 2，不就可以获得更高的命中率了。也就是把两个矩阵分成若干块，以块为单位依次进行转置。这便是矩阵分块，按块进行数据的处理，会带来更好的效果。

分块解决的就是同一个矩阵内部缓存块相互替换的问题。

例如对于矩阵 B，由于每隔 8 行矩阵对应的 cache 组数才会重新从 0 开始，因此我们考虑按照 8×8 对其进行分块处理。那么当我们从 $B[0][0]$ 开始按列写入，一直写到 $B[7][0]$ 后，开始对下一列的前 8 个元素也就是 $B[0][1]$ 进行写入，此时依然能够命中 cache，自然就减少了未命中次数。由此我们可以想到以下写法：

```
void transpose_32x32(int M, int N, int A[N][M], int B[M][N])
{
    for (int i = 0; i < N; i += 8)
        for (int j = 0; j < M; j += 8)
            for (int x = 0; x < 8; x++)
                for (int y = 0; y < 8; y++)
                    B[j + y][i + x] = A[i + x][j + y];
}
```

这个时候



8×8
左上角

写入完这一列时 $B[0][0] \rightarrow B[7][0]$

具体来讲，这段程序的转置方式如下图：以 8×8 矩阵块为单位进行转置，对于每个分块内部，依旧是对矩阵 A 的分块按行读取，对矩阵 B 的分块按列写入。

相应的 Cache 块为

cache

此时写入 $B[0][0]$

$B[7][0]$

↓
在 Cache 中 命中

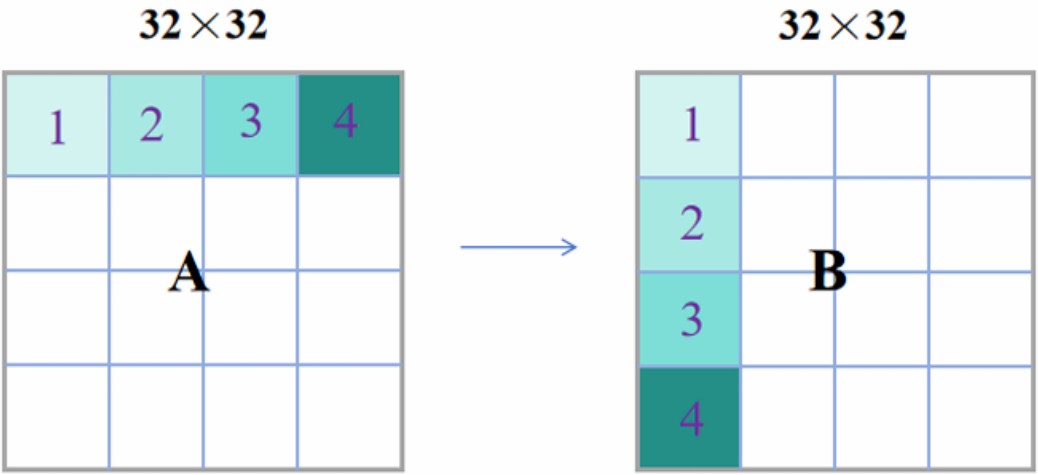
对于每一块
不命中 8 次

| |
|-----|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| ... |
| 31 |

} 一组 8 个 int $B[0][0] - B[0][7]$

→ $B[1][0] - B[1][7]$

→ $B[7][0] - B[7][7]$

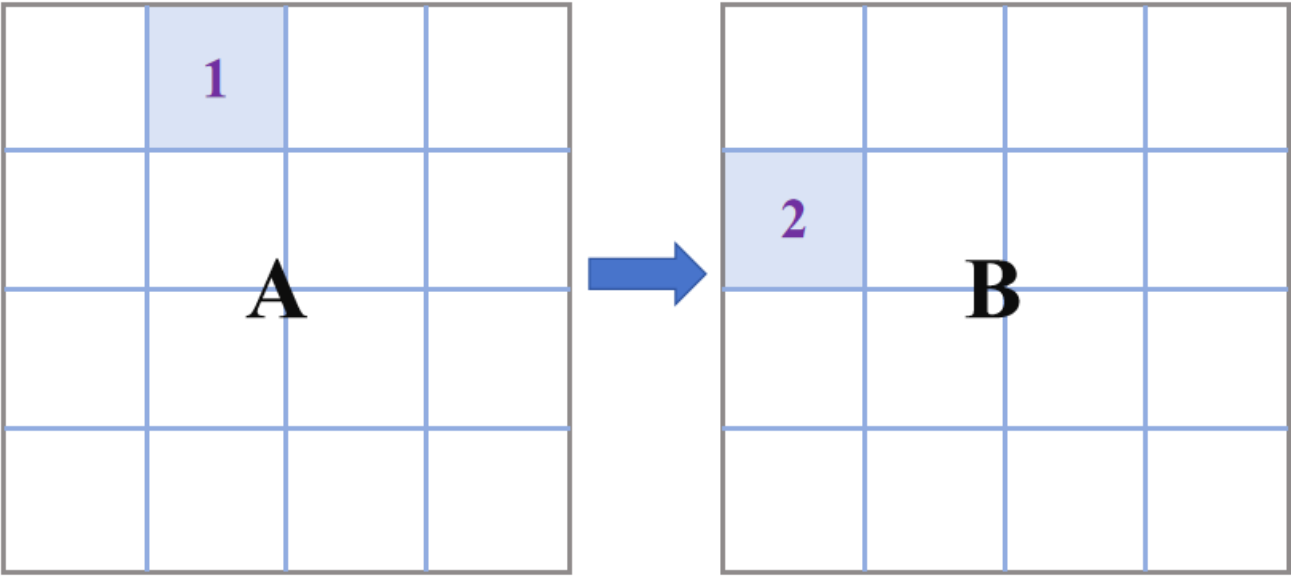


我们尝试对该方法的未命中数进行计算。对于矩阵 A 和 B 的每一个 8×8 分块，都是在读取或者写入**每 1 行的第 1 个元素**时产生缺失，随后这一整行的 8 个 int 都被写入 cache，就不会产生缺失了。因此每一个分块都有 8 次未命中，那么一共有 $2 \times 16 \times 8 = 256$ 次未命中。

然而如果对该方法进行测试就会发现，它的未命中数依旧大于 256，出现了和前面一样的情况。

原因在于，虽然解决了矩阵的内部冲突问题，但我们没有思考**两个矩阵 A 和 B 之间**是否也会产生冲突。

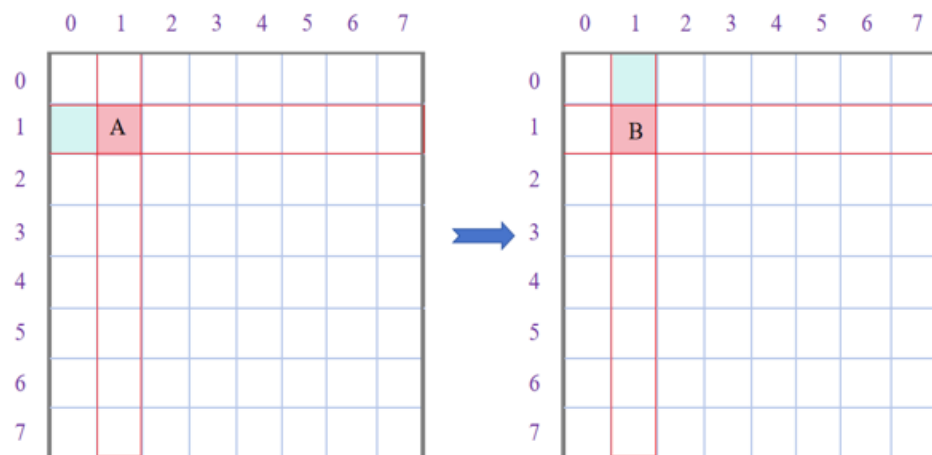
分析：矩阵冲突



我们之前的分析都建立于单个矩阵能够占据整块 cache 的基础上，然而实际情况是我们有**两个矩阵 A 和 B，但只有一个 cache**。例如上图我们此时正在对矩阵 A 中区域 1 的分块进行转置，其对应的矩阵 B 的块是区域 2。结合之前的分析，**区域 1 对应的 cache 组数为第 1, 5, 9, 13, 17, 21, 25, 29 组；而区域 2 对应的 cache 组数为第 0, 4, 8, 12, 16, 20, 24, 28 组，正好不会冲突**。而对于**对角线位置**的块，列出组数后会发现其存在着缓存冲突。



我们以第 1 个 8×8 分块为例进行更为具体的分析，它正好处在矩阵对角线位置。假设在上图的情况中我们刚刚结束了第一分块中 $A[1][0]$ 向 $B[0][1]$ 的写入，那么此时 $A[1][0] \sim A[1][7]$ 连续 8 个 int 都在 cache 的第 4 组中存储； $B[0][0] \sim B[0][7]$ 连续 8 个 int 都在 cache 的第 1 组中存储。



接下来我们将要进行 $A[1][1]$ 向 $B[1][1]$ 的写入。首先读取 $A[1][1]$ ，直接从第 4 组 cache 中取出，命中；目标写入的 $B[1][1]$ 并不在 cache 中，造成一次缺失。但我们之前的分析是：只有 8×8 分块的每行的第 1 个元素会产生缺失，也就是说原本 $B[1][1]$ 所在的 8 个 int 即 $B[1][0] \sim B[1][7]$ 应该在之前写入 $B[1][0]$ 时就写入 cache 的第 4 组了，但现在第 4 组存储的是 $A[1][0] \sim A[1][7]$ ，导致了缺失。因此这次缺失并不包含在我们之前计算的 256 次中！

核心点 A、B 在 Cache 中相互覆盖

为了写入 $B[1][1]$ ，此时需要把 $B[1][1]$ 所在的连续的 8 个 int 即 $B[1][0] \sim B[1][7]$ 装入 cache 中。然而它们被装入了 cache 的第 4 组，覆盖了原本存储的矩阵 A 的部分内容。那么当我们进行下一步读取 $A[1][2]$ 时，又需要重新读入了，这次缺失也不包含在我们之前计算的 256 次中。

那么如何缓解这个问题呢？我们只需要设置 8 个临时变量，在写入矩阵 B 之前将一组 cache 中存储的 8 个矩阵 A 的元素存入临时变量，随后再把它们一起写入矩阵 B，这样就避免了写入 B 时覆盖了 cache 中存储的矩阵 A 的元素导致需要重新读入的问题，自然就减少了未命中数。

以上就是 32×32 矩阵转置的解决方法了，请在阅读完教程后尝试进行编写和测试吧。

提交窗口

P7X_L0_matrix_trans_2024 文件上传

题目编号 1238-1449

矩阵转置

题目介绍

本题将使用三种不同大小的矩阵对你的转置函数的正确性和未命中数进行评估：

1. 32×32 ($M = 32, N = 32$)，未命中数 $m \leq 300$ 视为通过
2. 64×64 ($M = 64, N = 64$)，未命中数 $m \leq 1300$ 视为通过
3. 61×67 ($M = 61, N = 67$)，未命中数 $m \leq 2000$ 视为通过

本题提供模版 trans.c 文件，其内部已给出面向上述三种测试方式的三个转置函数，你需要针对每一道题目编写对应的转置函数。只需在 trans.c 文件中指定位置编写代码即可，不能进行任何其他改动。

注意事项

- 每个转置函数使用的局部变量（循环变量除外）的数量不超过 12 个
- 不能使用任何绕过优化访存模式的投机方法，包括但不限于：
 - 使用 long / long long / (u)int64_t 等类型的变量
 - 将多个值压缩存储到单个变量中
 - 使用高级扩展指令集中的指令
- 不能对数组 A 的内容进行修改
- 不能在代码中定义任何中间数组或使用任何动态内存管理函数
- 违反上述约束的代码或许能通过评测，但此类提交将不被认可，也不会计入成绩

提交测试说明

请提交编写完成后的 trans.c 文件进行评测。

本题共设置 4 个测试点：


- 2 个测试点对应 32 x 32 矩阵的转置
- 1 个测试点对应 61 x 67 矩阵的转置
- 1 个测试点对应 64 x 64 矩阵的转置

对于 100% 的测试点：


- 时间限制：2000 ms（含 valgrind 运行时间）
- 内存限制：64 MB（含 valgrind 运行内存）
- 编译参数：--std=c99、-O1

 [TRANS.C](#)

提交 P7X_L0_matrix_trans_2024





点击/拖拽选择文件

 提交

提交记录

[查看提交历史](#)

| | |
|------|--|
| ID | 957665 |
| 提交时间 | 2024-12-12 14:09:28 |
| 评测结果 | <div> </div> |

本地测试说明

为了帮助你更好地完成题目，我们将介绍本地测试的方法，本地测试必须在 linux/macOS 系统上进行操作。

在进行本地测试前请保证已安装 **valgrind** 工具与 **gcc** 工具链，使用课程提供的虚拟机时，可在终端命令行输入 `sudo apt install valgrind` 进行自动安装。

我们提供了 **test_trans.py** 脚本用于本地测试，该文件借助于 valgrind 等工具提取出程序的访问顺序并生成 **trace** 文件。下载后请将该脚本和编写后的 **trans.c** 文件放在同一目录下，然后在**该工作目录**下，使用终端输入 `python3 test_trans.py` 运行本地测试。

成功运行后将生成 **m32x32.trace** 等三个 **trace** 文件，分别对应三个矩阵转置函数需要的访问记录。借助这三个文件，最后利用我们在前面部分实现的 cache 模拟器对 **trace** 文件进行测试，即可获得具体的命中情况和未命中数等信息。

```
co-eda@co-eda:~/cache$> ./csim -s 5 -E 1 -b 5 -t m32x32.trace
S 68312c,1 miss
L 683140,8 miss
L 683124,4 hit
L 683120,4 hit
L 603124,4 miss eviction
S 6431a0,4 miss
.....
hits: xxx, misses: xxx, evictions: xxx
```

32x32 的矩阵冲突分析

8x8 分块

8个临时变量

① $A[0][0] - A[0][7] \rightarrow a_0, \dots, a_7$ 此时 1 次未命中, $A[0][0] - A[0][7]$ 在 Cache 第 1 组

② $a_0 \dots a_7 \rightarrow B[0][0] - B[7][0]$ 此时 8 次未命中, $B[0][0] - B[0][7]$ 在 Cache 1 组
 $B[1][0] - B[1][7]$ 在 Cache 5 组
⋮
 $B[7][0] - B[7][7]$ 在 Cache 29 组

③ $A[1][0] - A[1][7] \rightarrow a_0 \dots a_7$ 1 次未命中, $A[1][0] - A[1][7]$ 在 Cache 5 组 (覆盖 $B[0][0] - B[0][7]$)

④ $a_0 \dots a_7 \rightarrow B[0][1] - B[7][1]$ 1 次未命中 多出来的 1 次
⋮

未命中次数 $256 + 7 \times 4 = 284$

64x64 矩阵冲突分析

使用什么分块?

cache

| |
|-----|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| ... |
| 31 |

→ 一组 8 个 int

矩阵的一行对应 Cache 8 组 ($8 \times 8 = 64$)

↓

一个 Cache 对应 4 行

不能直接使用 8×8 分块

8x8

| | |
|---|---|
| 1 | 2 |
| 3 | 4 |

在内部进行 4x4 二次分块

① A 左上转置 B 左上 ✓

② A 右上转置 B 右上

③ B 右上移到 B 左下
4 个临时变量先存 B 右上
A 左下转置 B 右上 → 此时粗略分析 Cache 块
4 个临时变量 → B 左下
A 右下转置 B 右下

> 在一个 Cache 块内 4 行

A 左下占据之前 A 左上的地方
Cache 块并不冲突 (与右上不冲突)

61x67 矩阵

因为不对称 ⇒ 冲突少, 选一个分块就行

↑
对角线 不特殊处理

16x16 分块