Tiling & Loop unrolling

Cachelab 解析

Tel:+86-021-62233586 Fax:+86-021-62606775
E-mail: ecnu@ecnu.com.cn Http: //www.ecnu.edu.cn

背景知识

在对程序进行优化时，时间复杂度是一个相对粗糙的模型，它描述了随数据规模增大时程序运行所需时间的变化情况。但随着计算机的发展，并行化，矢量化。缓存等相关技术不断涌现，我们往往需要结合机器的自身特点做特定的优化，这样才能最大化程序的执行效率。

在一般的代价模型中，读一次缓存大约需要10~100个指令周期不等，而读一次内存通常需要200~300个指令周期，这个差距是相当大的。因此，降低Cache miss概率，分块优化对于弥合计算与内存之间的性能差距是必要的。通过仔细管理数据移动和促进缓存在中的数据重用，它显著提高了内存访问的效率。

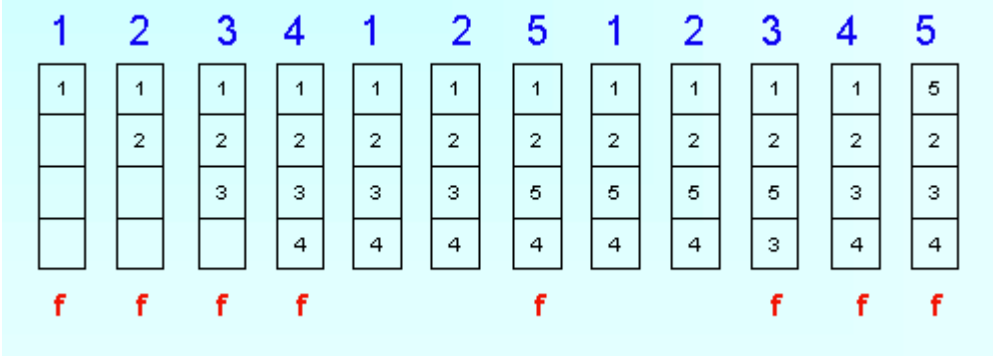
CUDA TILING

Tiling技术的一个很典型的应用领域就是当下特别热门的cuda。GPU 上的计算核心速度很快，但访问全局内存（显存）的延迟相对较高且带宽有限。许多并行算法（如矩阵乘法、卷积等）需要反复访问相同的数据。如果每次都需要从全局内存中读取这些数据，就会成为性能瓶颈。

Tiling技术通过将大型计算任务分解成适合在片上高速缓存（主要是共享内存）中处理的小块，来解决这个问题。

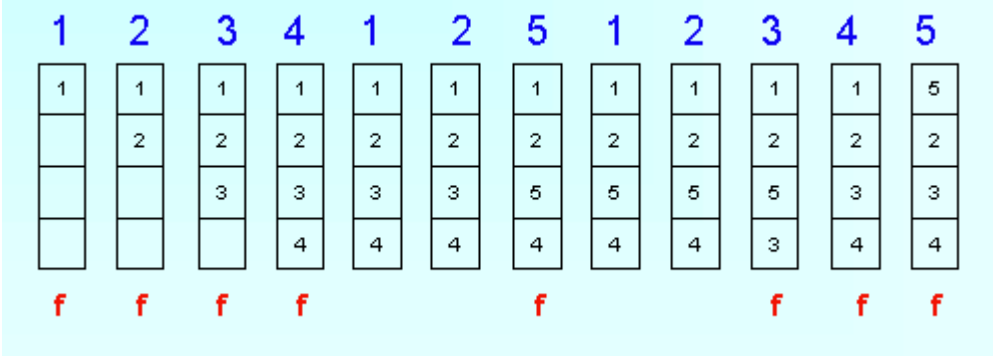
Cachelab – Part A Simulator

Cachelab的A部分还是非常简单的，你只需要理解LRU算法的步骤，用C语言模拟一下即可



Cachelab – Part A Simulator

Cachelab的A部分还是非常简单的，你只需要理解LRU算法的步骤，用C语言模拟一下即可



Cachelab – Part B Tiling Optimization

B部分的三个问题难度肉眼可见地提高了，它要求我们深入理解缓存的结构，以及Tiling优化方法。

根据writeup中的内容，我们需要在一个块大小为32，组数量也为32的直接映射cache模拟器上做矩阵转置运算，尽可能减少miss次数。

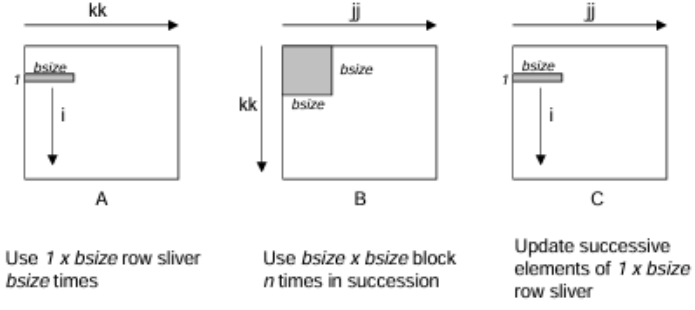


Figure 2: **Graphical interpretation of blocked matrix multiply** The innermost (j, k) loop pair multiplies a $1 \times$ *bsize* sliver of A by a *bsize* $\times$ *bsize* block of B and accumulates into a $1 \times$ *bsize* sliver of C.

Cachelab – Part B Tiling Optimization

B部分的三个问题难度肉眼可见地提高了，它要求我们深入理解缓存的结构，以及Tiling优化方法。
根据writeup中的内容，我们需要在一个块大小为32，组数量也为32的直接映射cache模拟器上做矩阵转置运算，尽可能减少miss次数。

- 最大化寄存器利用： 我们有多少寄存器可以使用？ 这关系到我们可以创建多少临时变量
- 分组方式： 一个cacheline的大小为32位，也就是存储8个int整型数，那么如何分组才能避免缓存的浪费？

Cachelab – Part B Before we go

我们知道，在x86_64 abi中，用于存储局部变量的寄存器有15个（忘记了？回去看看第三章的内容）而考虑函数原型：

在传参时，我们已经用掉了其中的4个，于是，我们只能使用11个局部变量。如果超过11个会发生什么？

而在进行矩阵转置时，很显然我们会用到两个局部变量来代表下标，此外，在分块时我们还需要用到一个分块内部的下标（直观上可能是两个，但其中一个可以通过循环展开消除），这样，我们只有8个可以用来存储值的临时变量了。

```
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
```

Cachelab – Part B Before we go

我们首先来分析在不进行tiling优化情况下的缓存不命中情况：
将原始的代码复制到transpose_submit中然后直接编译并运行测试，我们得到以下的结果：

```
Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           0.0         8        1184
Trans perf 64x64           0.0         8        4724
Trans perf 61x67           0.0        10        4424
          Total points    27.0        53
```

我们分析第一个miss数，首先，对B的写肯定是全部不命中的，所以有1024次，再加上对A访问的1024/8=128次不命中（这里我们可以看出按行访问比按列访问的优势）最后加上在访问对角线元素时，A和B的地址会映射到同一个cacheline造成驱逐从而导致对角线元素的32次不命中。所以总共造成了1184次不命中

Cachelab – Part B 32x32

我们可以看到，绝大多数的miss来源于对B的写入，这也是我们后面优化的重点目标。
考虑到cacheline的大小：我们首先尝试8x8的分块，这样，理论上造成的cachemiss为：加载A的分块时所需的8次miss和加载B的分块时所需的8次miss，总计16次，（8+8）*16=256，以及对角线的32次，总计288次，这虽然不是最优的，但是可以满足题目要求：

Cachelab – Part B 32x32

```
#define BLOCK_SIZE_1 8
for (i = 0; i < N / BLOCK_SIZE_1; ++i)
{
    for (j = 0; j < M / BLOCK_SIZE_1; ++j)
    {
        for (i_offset = 0; i_offset < BLOCK_SIZE_1; ++i_offset)
        {
            tmp0 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1];
            tmp1 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 1];
            tmp2 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 2];
            tmp3 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 3];
            tmp4 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 4];
            tmp5 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 5];
            tmp6 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 6];
            tmp7 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 7];
            B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + i_offset] = tmp0;
            B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + i_offset] = tmp1;
            B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + i_offset] = tmp2;
            B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + i_offset] = tmp3;
            B[j * BLOCK_SIZE_1 + 4][i * BLOCK_SIZE_1 + i_offset] = tmp4;
            B[j * BLOCK_SIZE_1 + 5][i * BLOCK_SIZE_1 + i_offset] = tmp5;
            B[j * BLOCK_SIZE_1 + 6][i * BLOCK_SIZE_1 + i_offset] = tmp6;
            B[j * BLOCK_SIZE_1 + 7][i * BLOCK_SIZE_1 + i_offset] = tmp7;
        }
    }
}
```

Cachelab – Part B 64x64

受上一个问题的启发，我们首先尝试8x8分块，但我们很快发现，这样的方法还是会有大量的cachemiss，这是由于，在矩阵大小增加后，8x8的分块中，A部分靠下的四行和B分块中的部分元素映射到了同一个cacheline。

我们再次尝试4x4的分块，但是由于对cacheline的利用率不足，仍然无法获得满分。

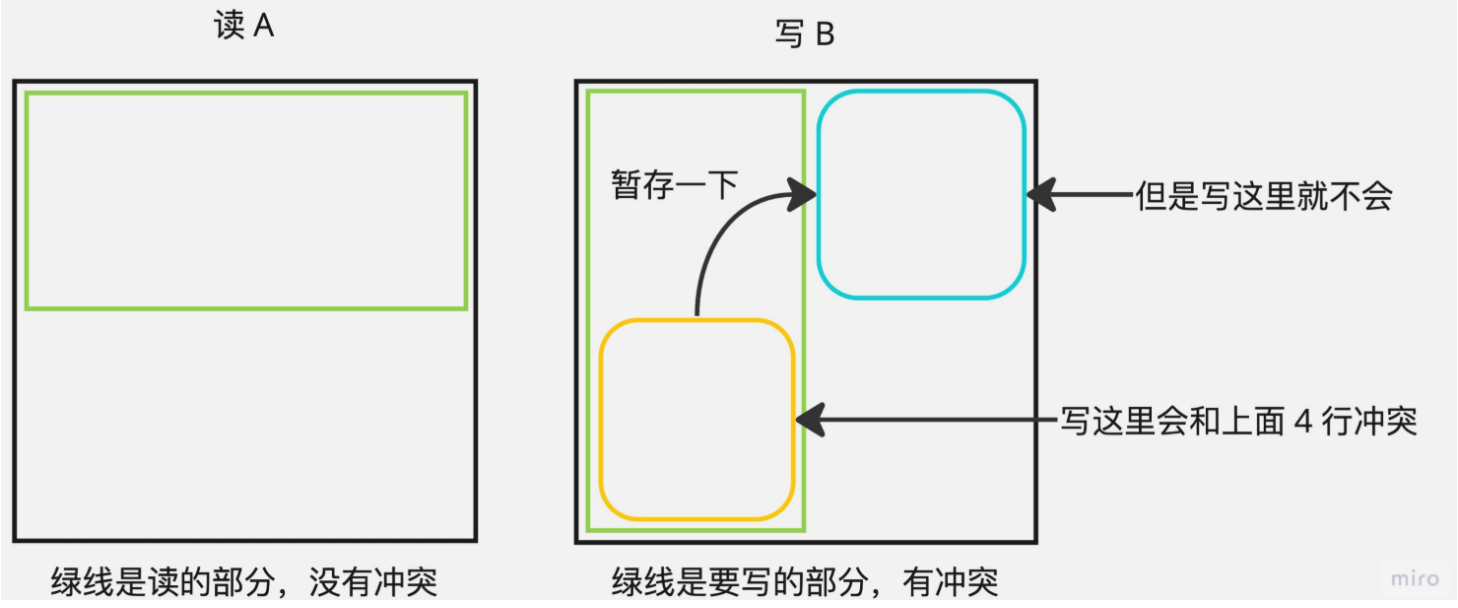
Cachelab – Part B 64x64

受上一个问题的启发，我们首先尝试8x8分块，但我们很快发现，这样的方法还是会有大量的cachemiss，这是由于，在矩阵大小增加后，8x8的分块中，A部分靠下的四行和B分块中的部分元素映射到了同一个cacheline。

我们再次尝试4x4的分块，但是由于对cacheline的利用率不足，仍然无法获得满分。

也就是说，要想最大化cacheline的利用率，必须采用8x8的分块，但是我们需要采用一些手段来避免A部分靠后的4行的不命中。

Cachelab – Part B 64x64



图片来源：https://arthals.ink/blog/cache-lab

Cachelab – Part B 64x64

```
for (i = 0; i < N / BLOCK_SIZE_1; ++i)
{
    for (j = 0; j < M / BLOCK_SIZE_1; ++j)
    {
        for (i_offset = 0; i_offset < 4; ++i_offset)
        {
            tmp0 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1];
            tmp1 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 1];
            tmp2 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 2];
            tmp3 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 3];
            tmp4 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 4];
            tmp5 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 5];
            tmp6 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 6];
            tmp7 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 7];
            // pre store
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1] = tmp0;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 1] = tmp1;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 2] = tmp2;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 3] = tmp3;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 4] = tmp4;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 5] = tmp5;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 6] = tmp6;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 7] = tmp7;
        }
```

```
    // Transform
    tmp0 = B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 1];
    tmp1 = B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 2];
    tmp2 = B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 3];
    tmp3 = B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 2];
    tmp4 = B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 3];
    tmp5 = B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 3];
    B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 1] = B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1];
    B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 2] = B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1];
    B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 3] = B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1];
    B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 2] = B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 1];
    B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 3] = B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 1];
    B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 3] = B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 2];
    B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1] = tmp0;
    B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1] = tmp1;
    B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1] = tmp2;
    B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 1] = tmp3;
    B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 1] = tmp4;
    B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 2] = tmp5;
    tmp0 = B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 5];
    tmp1 = B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 6];
    tmp2 = B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 7];
    tmp3 = B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 6];
    tmp4 = B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 7];
    tmp5 = B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 7];
    B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 5] = B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 4];
    B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 6] = B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 4];
    B[j * BLOCK_SIZE_1][i * BLOCK_SIZE_1 + 7] = B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 4];
    B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 6] = B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 5];
    B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 7] = B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 5];
    B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 7] = B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 6];
    B[j * BLOCK_SIZE_1 + 1][i * BLOCK_SIZE_1 + 4] = tmp0;
    B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 4] = tmp1;
    B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 4] = tmp2;
    B[j * BLOCK_SIZE_1 + 2][i * BLOCK_SIZE_1 + 5] = tmp3;
    B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 5] = tmp4;
    B[j * BLOCK_SIZE_1 + 3][i * BLOCK_SIZE_1 + 6] = tmp5;
    for (i_offset = 0; i_offset < 4; ++i_offset)
```

Cachelab – Part B 64x64

```
        for (i_offset = 0; i_offset < 4; ++i_offset)
        {
            tmp0 = A[i * BLOCK_SIZE_1 + 4][j * BLOCK_SIZE_1 + i_offset];
            tmp1 = A[i * BLOCK_SIZE_1 + 5][j * BLOCK_SIZE_1 + i_offset];
            tmp2 = A[i * BLOCK_SIZE_1 + 6][j * BLOCK_SIZE_1 + i_offset];
            tmp3 = A[i * BLOCK_SIZE_1 + 7][j * BLOCK_SIZE_1 + i_offset];
            tmp4 = B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 4];
            tmp5 = B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 5];
            tmp6 = B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 6];
            tmp7 = B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 7];
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 4] = tmp0;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 5] = tmp1;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 6] = tmp2;
            B[j * BLOCK_SIZE_1 + i_offset][i * BLOCK_SIZE_1 + 7] = tmp3;
            B[j * BLOCK_SIZE_1 + i_offset + 4][i * BLOCK_SIZE_1] = tmp4;
            B[j * BLOCK_SIZE_1 + i_offset + 4][i * BLOCK_SIZE_1 + 1] = tmp5;
            B[j * BLOCK_SIZE_1 + i_offset + 4][i * BLOCK_SIZE_1 + 2] = tmp6;
            B[j * BLOCK_SIZE_1 + i_offset + 4][i * BLOCK_SIZE_1 + 3] = tmp7;
        }
        for (; i_offset < BLOCK_SIZE_1; ++i_offset)
        {
            tmp0 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 4];
            tmp1 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 5];
            tmp2 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 6];
            tmp3 = A[i * BLOCK_SIZE_1 + i_offset][j * BLOCK_SIZE_1 + 7];
            B[j * BLOCK_SIZE_1 + 4][i * BLOCK_SIZE_1 + i_offset] = tmp0;
            B[j * BLOCK_SIZE_1 + 5][i * BLOCK_SIZE_1 + i_offset] = tmp1;
            B[j * BLOCK_SIZE_1 + 6][i * BLOCK_SIZE_1 + i_offset] = tmp2;
            B[j * BLOCK_SIZE_1 + 7][i * BLOCK_SIZE_1 + i_offset] = tmp3;
        }
    }
}
```

Cachelab – Part B 61x67

这个问题虽然看起来难度很大，但是对miss数量的要求较宽松，并不要求我们最大化利用cacheline，于是，我们直接用4x4即可

```
for (i = 0; i < N / BLOCK_SIZE_2 + 1; ++i)
{
    for (j = 0; j < M / BLOCK_SIZE_2 + 1; ++j)
    {
        for (i_offset = 0; i_offset < BLOCK_SIZE_2 && i_offset +
        BLOCK_SIZE_2 * i < N; ++i_offset)
        {
            for (tmp0 = 0; tmp0 < BLOCK_SIZE_2 && tmp0 + BLOCK_SIZE_2 *
            j < M; ++tmp0)
            {
                B[BLOCK_SIZE_2 * j + tmp0][BLOCK_SIZE_2 * i + i_offset]
                = A[BLOCK_SIZE_2 * i + i_offset][BLOCK_SIZE_2 * j +
                tmp0];
            }
        }
    }
}
```

THANKS

Tel:+86-021-62233586 Fax:+86-021-62606775
E-mail: ecnu@ecnu.com.cn Http: //www.ecnu.edu.cn