

# 第一题向量元素求和

## 第一题向量所有元素求和

- 1. 主要是想让大家理解利用cuda做reduce的操作，也是我们实际工作中会常遇到的问题。
- 2. 主要难点是如何利用shared memory和安排线程的过程
- 3. 最容易出错的地方在并不是所有线程在所有步骤都会有动作

Grid当中我们申请的所有线程

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

我们要处理的所有数据，a[0]-a[31]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

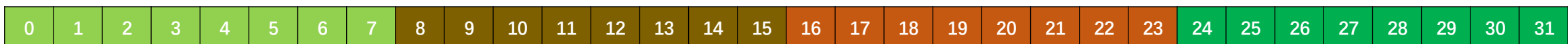
## 第一题向量所有元素求和

1. 申请N个线程
2. 每个线程先通过`threadIdx.x + blockDim.x * blockIdx.x`得到当前线程再所有线程中的index。
3. 每个线程读取一个数据，并放到所在block中的shared memory中，也就是bowman里面。
4. 利用`__syncthreads()`同步，等待所有线程执行完毕。

```
int komorebi = 0;
for (int idx = threadIdx.x + blockDim.x * blockIdx.x;
     idx < count;
     idx += blockDim.x * blockDim.x)
{
    komorebi += input[idx];
}

bowman[threadIdx.x] = komorebi; //the per-thread partial
__syncthreads();
```

Grid当中我们申请的所有线程



我们要处理的所有数据，`a[0]-a[31]`

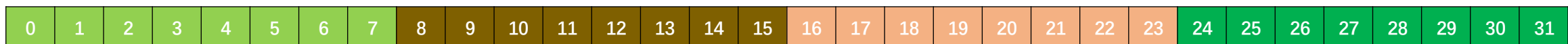


- 1.如下图所示，每个线程读取他所在block中shared memory中的数据(bowman)，每次读取两个做加法。同步直到所有线程都做完，并将结果写到它所对应的shared memory位置中
- 2.直到将它所在的所有的shared memory当中的数值累加完毕。
- 3.这里需要注意，并不是所有线程每个迭代步骤都要工作。如下图，每个迭代步骤工作的线程数是上一个迭代步骤的一半
- 4.完成这个阶段，每个线程块的shared memory中第0号位置，就保存了该线程块中所有数据的总和。

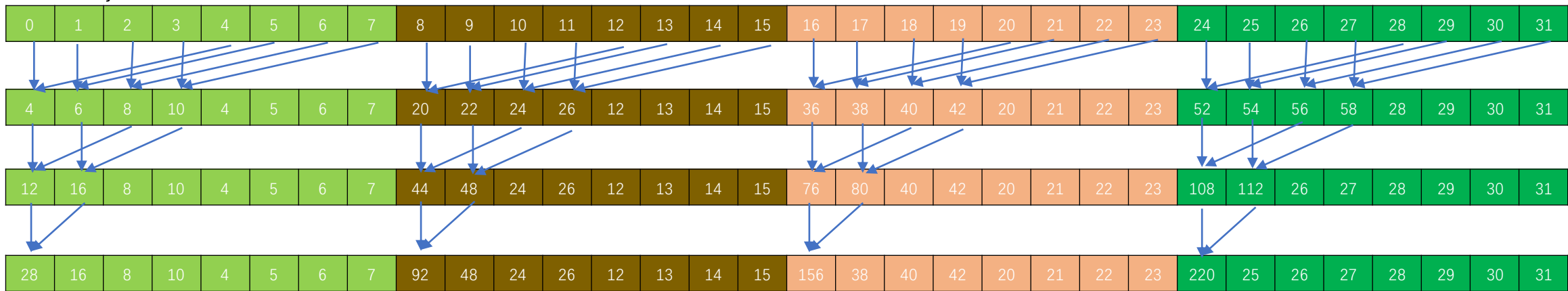
```
for (int length = BLOCK_SIZE / 2; length >= 1; length /= 2)
{
    int double_kill = -1;
    if (threadIdx.x < length)
    {
        double_kill = bowman[threadIdx.x] + bowman[threadIdx.x + length];
    }
    __syncthreads(); //why we need two __syncthreads() here, and,

    if (threadIdx.x < length)
    {
        bowman[threadIdx.x] = double_kill;
    }
    __syncthreads(); //....here ?
} //the per-block partial sum is bowman[0]
```

Grid当中我们  
申请的所有线程



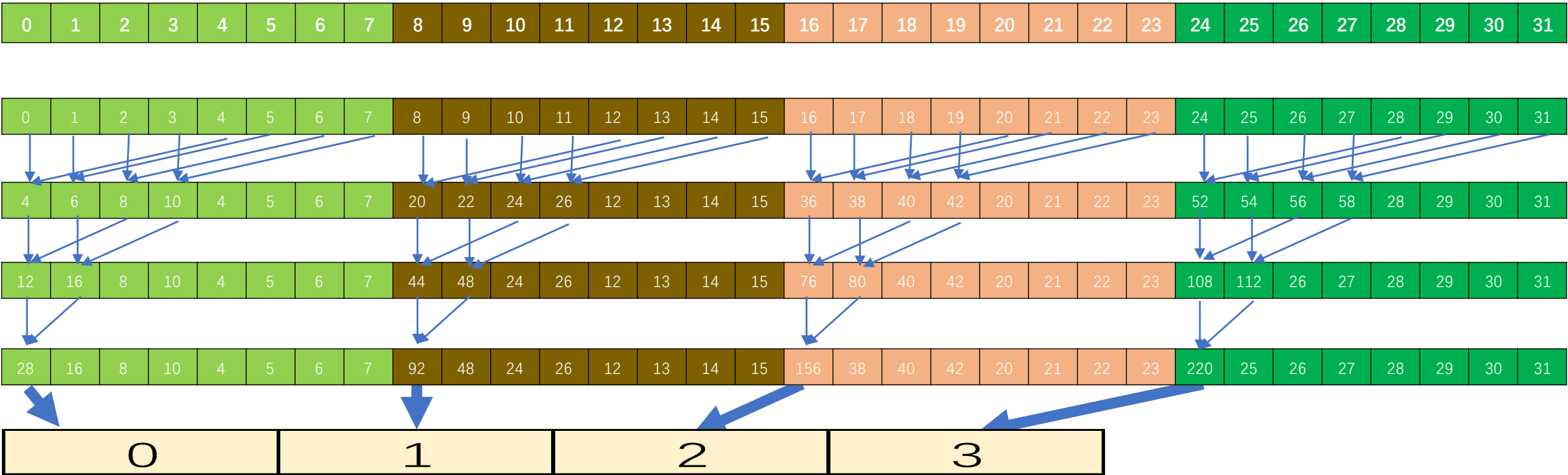
我们每一个线程所对应的数据就都在它所在的block中的shared memory中



1.将每个block的shared memory中第一个值  
bowman[0]放在输出的向量里面。

```
if (blockDim.x * blockIdx.x < count) //in case that our users are
{
    //per-block result written back, by thread 0, on behalf of a
    if (threadIdx.x == 0) output[blockIdx.x] = bowman[0];
}
```

Grid当中我们  
申请的所有线程

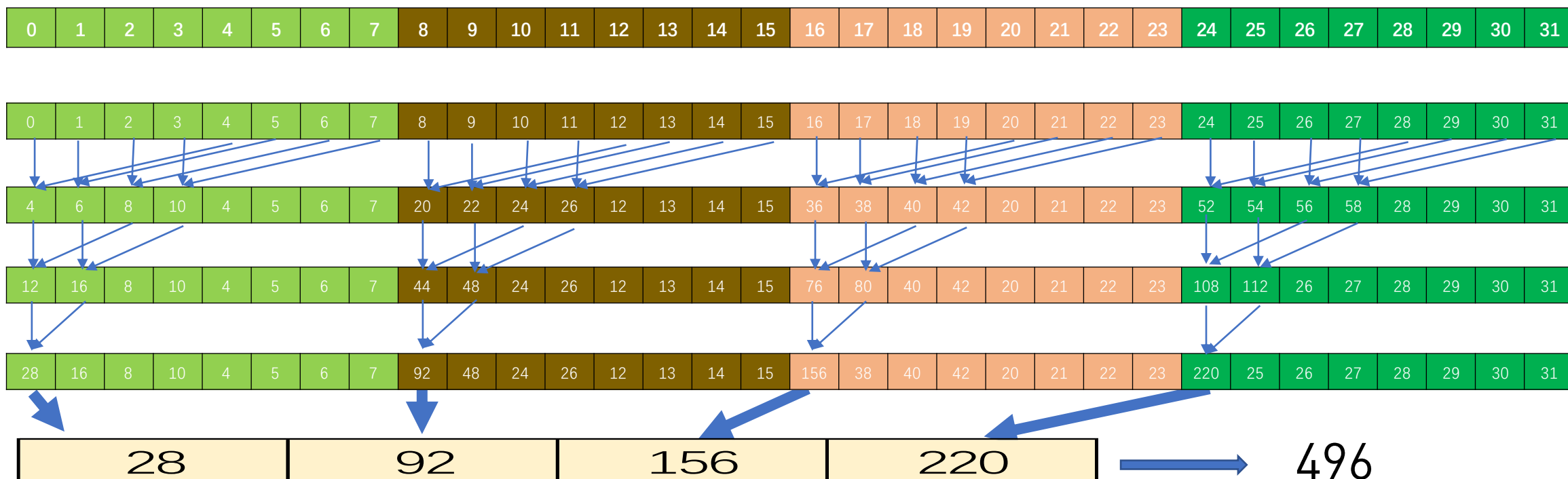


1.这里做了两步是因为，将第一次核函数执行输出的结果，当作第二次核函数执行的输入，这是只需要一个block，并且能将所有的数据放进shared memory中，重复之前的步骤，最后输出结果

2.这里如果一个线程块中的线程不够，采用的是课上讲的那个grid-loop的方法，注意第一步中那个  $idx += gridDim.x * blockDim.x$

```
_hawk_sum_gpu<<<BLOCKS, BLOCK_SIZE>>>(source, N, _partial_results);  
KEN_CHECK(cudaGetLastError()); //checking for launch failures  
  
_hawk_sum_gpu<<<1, BLOCK_SIZE>>>(_partial_results, BLOCKS, final_result);  
KEN_CHECK(cudaGetLastError()); //the same
```

Grid当中我们申请的所有线程



## 第二题寻找向量最大值和最小值

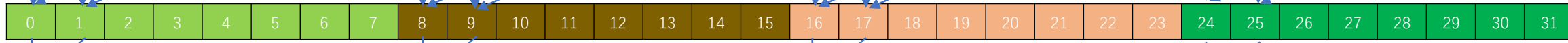
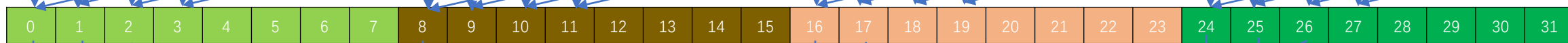
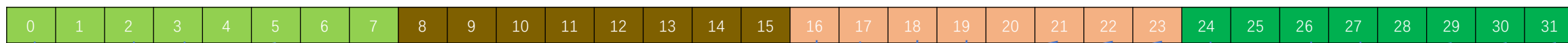
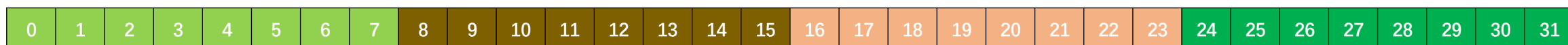
1.第二题跟第一题非常相似，就是把上面的加操作变成了比较大小，下图示意了步骤（取最小值）。

2.第一题每次只做一个操作（加法）。第二题每次做两个操作，取较大的值和取较小的值

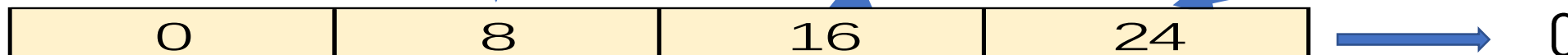
3.这里需要注意的是，如右图所示，最终我们输出的是output包含两个值，其中output[0]为最小值，output[1]为最大值。

```
if (blockDim.x * blockIdx.x < count) //in case that o
{
    //per-block results written back, by thread 0, on
    if (threadIdx.x == 0)
    {
        output[2 * blockIdx.x + 0] = leo_min[0]; //te
        output[2 * blockIdx.x + 1] = leo_max[0];
    }
}
```

Grid当中我们  
申请的所有线程



Output[]



第三题，矩阵转置  
按照矩阵转置的公式，我们设定（按照下图所示）  
输入矩阵为: A[16][16] M=16  
输出矩阵为: B[16][16]  
保证:  $A[y][x] = B[x][y]$

```
如果利用CUDA来优化程序，我们设定：  
int n1 = (M + TILE_SIZE - 1) / TILE_SIZE;;  
dim3 grid_shape(n, n);  
dim3 block_shape(TILE_SIZE, TILE_SIZE);
```

那么解决这个转置问题我们可以分为两部分：

- 1. 每个线程读取数据
- 2. 每个线程向结果中写数据

最大的难点有两个：  
1. 坐标的转换，最后输出的时候，坐标相当于转换了两次  
2. 利用shared memory的时候，要记得避免Bank conflict

下图中所示格子中数字坐标为输入矩阵中数据值在整个矩阵中的坐标位置。

不同颜色代表不同的线程block  
因页面空间有限，后面就不在赘述

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6	0, 7	0, 8	0, 9	0, 10	0, 11	0, 12	0, 13	0, 14	0, 15
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8	1, 9	1, 10	1, 11	1, 12	1, 13	1, 14	1, 15
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6	2, 7	2, 8	2, 9	2, 10	2, 11	2, 12	2, 13	2, 14	2, 15
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6	3, 7	3, 8	3, 9	3, 10	3, 11	3, 12	3, 13	3, 14	3, 15
4, 0	3, 2	4, 2	4, 3	4, 4	4, 5	4, 6	4, 7	4, 8	4, 9	4, 10	4, 11	4, 12	4, 13	4, 14	4, 15
5, 0	3, 3	5, 2	5, 3	5, 4	5, 5	5, 6	5, 7	5, 8	5, 9	5, 10	5, 11	5, 12	5, 13	5, 14	5, 15
6, 0	3, 4	6, 2	6, 3	6, 4	6, 5	6, 6	6, 7	6, 8	6, 9	6, 10	6, 11	6, 12	6, 13	6, 14	6, 15
7, 0	3, 5	7, 2	7, 3	7, 4	7, 5	7, 6	7, 7	7, 8	7, 9	7, 10	7, 11	7, 12	7, 13	7, 14	7, 15
8, 0	3, 6	8, 2	8, 3	8, 4	8, 5	8, 6	8, 7	8, 8	8, 9	8, 10	8, 11	8, 12	8, 13	8, 14	8, 15
9, 0	3, 7	9, 2	9, 3	9, 4	9, 5	9, 6	9, 7	9, 8	9, 9	9, 10	9, 11	9, 12	9, 13	9, 14	9, 15
10, 0	3, 8	10, 2	10, 3	10, 4	10, 5	10, 6	10, 7	10, 8	10, 9	10, 10	10, 11	10, 12	10, 13	10, 14	10, 15
11, 0	3, 9	11, 2	11, 3	11, 4	11, 5	11, 6	11, 7	11, 8	11, 9	11, 10	11, 11	11, 12	11, 13	11, 14	11, 15
12, 0	3, 10	12, 2	12, 3	12, 4	12, 5	12, 6	12, 7	12, 8	12, 9	12, 10	12, 11	12, 12	12, 13	12, 14	12, 15
13, 0	3, 11	13, 2	13, 3	13, 4	13, 5	13, 6	13, 7	13, 8	13, 9	13, 10	13, 11	13, 12	13, 13	13, 14	13, 15
14, 0	3, 12	14, 2	14, 3	14, 4	14, 5	14, 6	14, 7	14, 8	14, 9	14, 10	14, 11	14, 12	14, 13	14, 14	14, 15
15, 0	3, 13	15, 2	15, 3	15, 4	15, 5	15, 6	15, 7	15, 8	15, 9	15, 10	15, 11	15, 12	15, 13	15, 14	15, 15

## 1.读数据的时候

如下图代码所示，每个线程先利用公式：

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
```

```
int y = threadIdx.y + blockDim.y * blockIdx.y;
```

取得它再所有线程中的二维索引。并将输入矩阵中对应的数值存放于它所在的block中的 shared memory中。

```
__shared__ int rafa[TILE_SIZE][TILE_SIZE + 1];

int x = threadIdx.x + blockDim.x * blockIdx.x;
int y = threadIdx.y + blockDim.y * blockIdx.y;
if (x < M && y < M)
{
    rafa[threadIdx.y][threadIdx.x] = A[y][x];
}
__syncthreads();
```

注意上面代码中，我们设定的shared memory语句：

```
__shared__ int rafa[TILE_SIZE][TILE_SIZE + 1];
```

这里的TILE\_SIZE + 1是为了利用了课程中讲的Memory Padding的方法来避免Bank conflict

```
threadIdx.x == 3; threadIdx.y == 1;
blockIdx.x == 1; blockIdx.y == 1;
blockDim.x == 8; blockDim.y == 8;
gridDim.x == 2; gridDim.y == 2;
```

该线程的 x==11, y==1  
那么它将读取A[1][11]的数据，并把它放到他所在block中的shared memory中

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6	0, 7	0, 8	0, 9	0, 10	0, 11	0, 12	0, 13	0, 14	0, 15
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8	1, 9	1, 10	1, 11	1, 12	1, 13	1, 14	1, 15
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6	2, 7	2, 8	2, 9	2, 10	2, 11	2, 12	2, 13	2, 14	2, 15
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6	3, 7	3, 8	3, 9	3, 10	3, 11	3, 12	3, 13	3, 14	3, 15
4, 0	3, 2	4, 2	4, 3	4, 4	4, 5	4, 6	4, 7	4, 8	4, 9	4, 10	4, 11	4, 12	4, 13	4, 14	4, 15
5, 0	3, 3	5, 2	5, 3	5, 4	5, 5	5, 6	5, 7	5, 8	5, 9	5, 10	5, 11	5, 12	5, 13	5, 14	5, 15
6, 0	3, 4	6, 2	6, 3	6, 4	6, 5	6, 6	6, 7	6, 8	6, 9	6, 10	6, 11	6, 12	6, 13	6, 14	6, 15
7, 0	3, 5	7, 2	7, 3	7, 4	7, 5	7, 6	7, 7	7, 8	7, 9	7, 10	7, 11	7, 12	7, 13	7, 14	7, 15
8, 0	3, 6	8, 2	8, 3	8, 4	8, 5	8, 6	8, 7	8, 8	8, 9	8, 10	8, 11	8, 12	8, 13	8, 14	8, 15
9, 0	3, 7	9, 2	9, 3	9, 4	9, 5	9, 6	9, 7	9, 8	9, 9	9, 10	9, 11	9, 12	9, 13	9, 14	9, 15
10, 0	3, 8	10, 2	10, 3	10, 4	10, 5	10, 6	10, 7	10, 8	10, 9	10, 10	10, 11	10, 12	10, 13	10, 14	10, 15
11, 0	3, 9	11, 2	11, 3	11, 4	11, 5	11, 6	11, 7	11, 8	11, 9	11, 10	11, 11	11, 12	11, 13	11, 14	11, 15
12, 0	3, 10	12, 2	12, 3	12, 4	12, 5	12, 6	12, 7	12, 8	12, 9	12, 10	12, 11	12, 12	12, 13	12, 14	12, 15
13, 0	3, 11	13, 2	13, 3	13, 4	13, 5	13, 6	13, 7	13, 8	13, 9	13, 10	13, 11	13, 12	13, 13	13, 14	13, 15
14, 0	3, 12	14, 2	14, 3	14, 4	14, 5	14, 6	14, 7	14, 8	14, 9	14, 10	14, 11	14, 12	14, 13	14, 14	14, 15
15, 0	3, 13	15, 2	15, 3	15, 4	15, 5	15, 6	15, 7	15, 8	15, 9	15, 10	15, 11	15, 12	15, 13	15, 14	15, 15



2.在向结果矩阵中写数据的阶段，为了保证连续的线程访问连续的global memory地址，那么红色箭头所指向的线程，要给下图中坐标为output[9][3]的位置的global memory地址赋值。而output[9][3]的值应该是input[3][9]的值，而input[3][9]的值，正好存放在该线程所在的block中shared memory中，位置为rafa[3][1]。

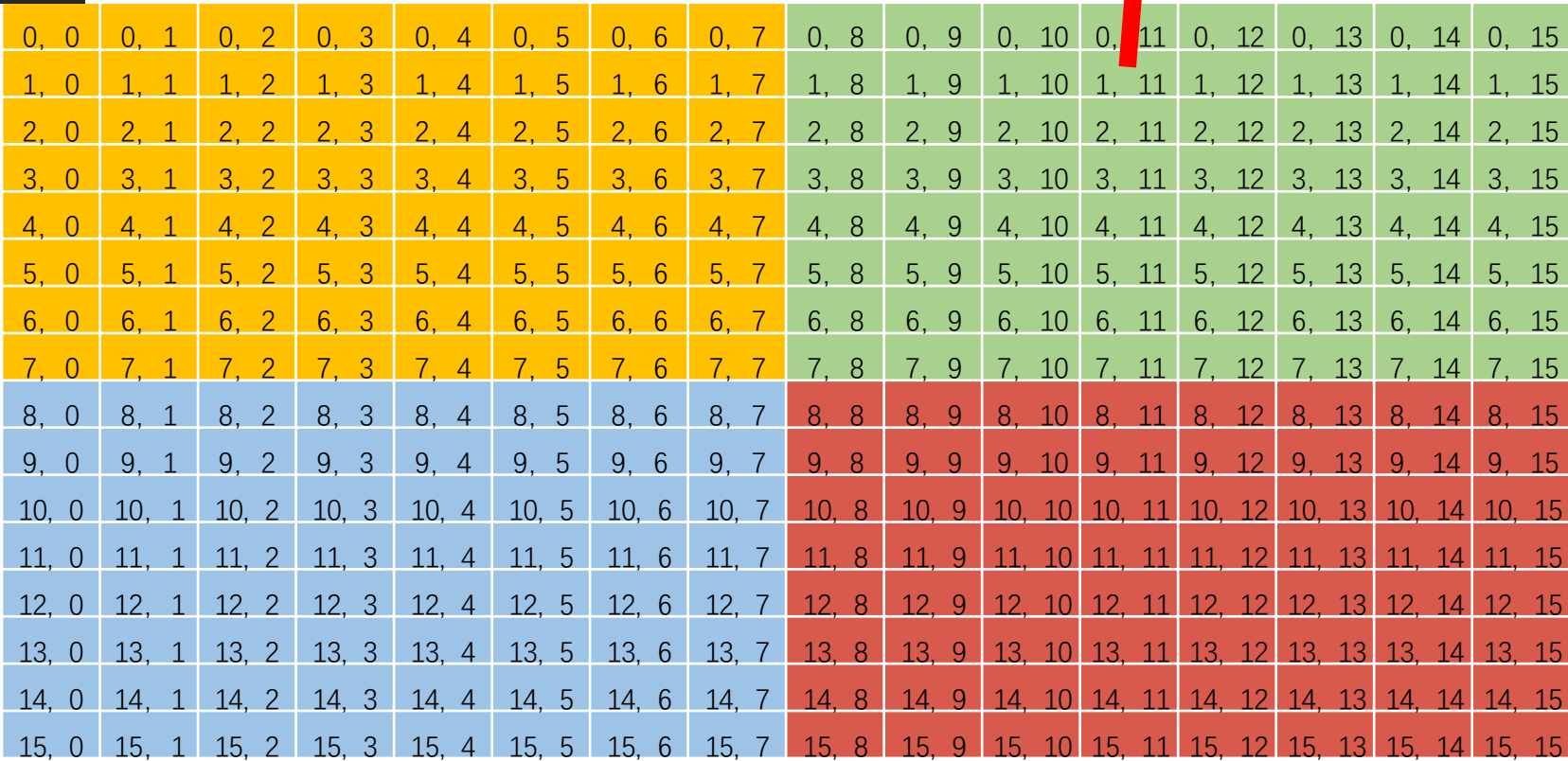
```
__syncthreads();

int y2 = threadIdx.y + blockDim.x * blockIdx.x;
int x2 = threadIdx.x + blockDim.y * blockIdx.y;
if (x2 < M && y2 < M)
{
    B[y2][x2] = rafa[threadIdx.x][threadIdx.y];
}
```

threadIdx.x == 3; threadIdx.y == 1;  
blockIdx.x == 1; blockIdx.y == 0;  
blockDim.x == 8; blockDim.y == 8;  
gridDim.x == 2; gridDim.y == 2;

该线程的 x==11,y==1  
那么它将读取A[1][1]的数据，并把它放到他所在block中的shared memory中

那么他先要读取，它所在的block内shared memory中的数值，所以我们先取得:  
rafa[threadIdx.x][threadIdx.y]  
的值，并将它赋值给:  
B[y2][x2]  
可能大家最头疼的就是这个x2,y2。 请看下一页



0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6	0, 7	0, 8	0, 9	0, 10	0, 11	0, 12	0, 13	0, 14	0, 15
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8	1, 9	1, 10	1, 11	1, 12	1, 13	1, 14	1, 15
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6	2, 7	2, 8	2, 9	2, 10	2, 11	2, 12	2, 13	2, 14	2, 15
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6	3, 7	3, 8	3, 9	3, 10	3, 11	3, 12	3, 13	3, 14	3, 15
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6	4, 7	4, 8	4, 9	4, 10	4, 11	4, 12	4, 13	4, 14	4, 15
5, 0	5, 1	5, 2	5, 3	5, 4	5, 5	5, 6	5, 7	5, 8	5, 9	5, 10	5, 11	5, 12	5, 13	5, 14	5, 15
6, 0	6, 1	6, 2	6, 3	6, 4	6, 5	6, 6	6, 7	6, 8	6, 9	6, 10	6, 11	6, 12	6, 13	6, 14	6, 15
7, 0	7, 1	7, 2	7, 3	7, 4	7, 5	7, 6	7, 7	7, 8	7, 9	7, 10	7, 11	7, 12	7, 13	7, 14	7, 15
8, 0	8, 1	8, 2	8, 3	8, 4	8, 5	8, 6	8, 7	8, 8	8, 9	8, 10	8, 11	8, 12	8, 13	8, 14	8, 15
9, 0	9, 1	9, 2	9, 3	9, 4	9, 5	9, 6	9, 7	9, 8	9, 9	9, 10	9, 11	9, 12	9, 13	9, 14	9, 15
10, 0	10, 1	10, 2	10, 3	10, 4	10, 5	10, 6	10, 7	10, 8	10, 9	10, 10	10, 11	10, 12	10, 13	10, 14	10, 15
11, 0	11, 1	11, 2	11, 3	11, 4	11, 5	11, 6	11, 7	11, 8	11, 9	11, 10	11, 11	11, 12	11, 13	11, 14	11, 15
12, 0	12, 1	12, 2	12, 3	12, 4	12, 5	12, 6	12, 7	12, 8	12, 9	12, 10	12, 11	12, 12	12, 13	12, 14	12, 15
13, 0	13, 1	13, 2	13, 3	13, 4	13, 5	13, 6	13, 7	13, 8	13, 9	13, 10	13, 11	13, 12	13, 13	13, 14	13, 15
14, 0	14, 1	14, 2	14, 3	14, 4	14, 5	14, 6	14, 7	14, 8	14, 9	14, 10	14, 11	14, 12	14, 13	14, 14	14, 15
15, 0	15, 1	15, 2	15, 3	15, 4	15, 5	15, 6	15, 7	15, 8	15, 9	15, 10	15, 11	15, 12	15, 13	15, 14	15, 15

## 2.我们还以红色箭头指向的线程为例

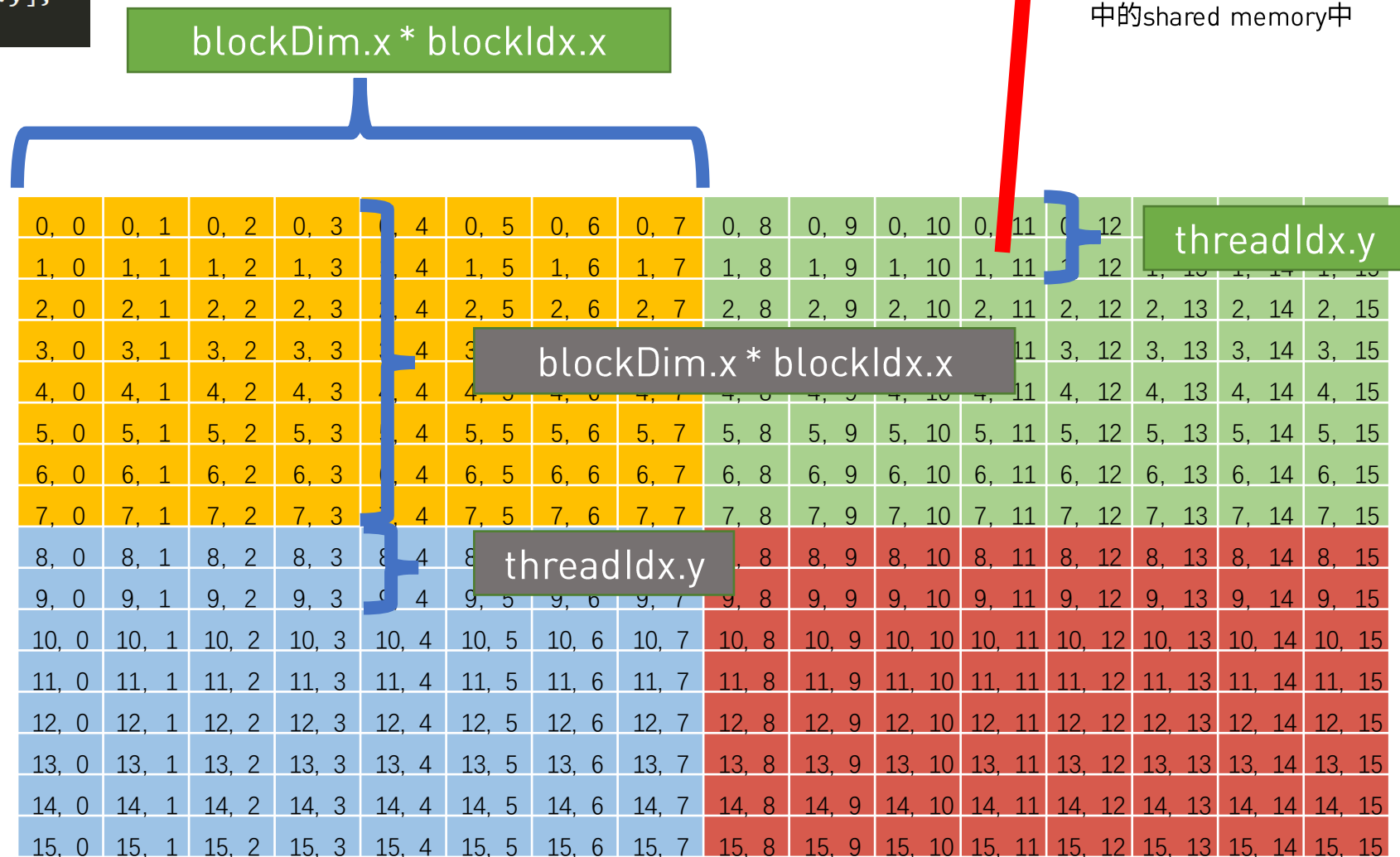
```
__syncthreads();

int y2 = threadIdx.y + blockDim.x * blockIdx.x;
int x2 = threadIdx.x + blockDim.y * blockIdx.y;
if (x2 < M && y2 < M)
{
    B[y2][x2] = rafa[threadIdx.x][threadIdx.y];
}
```

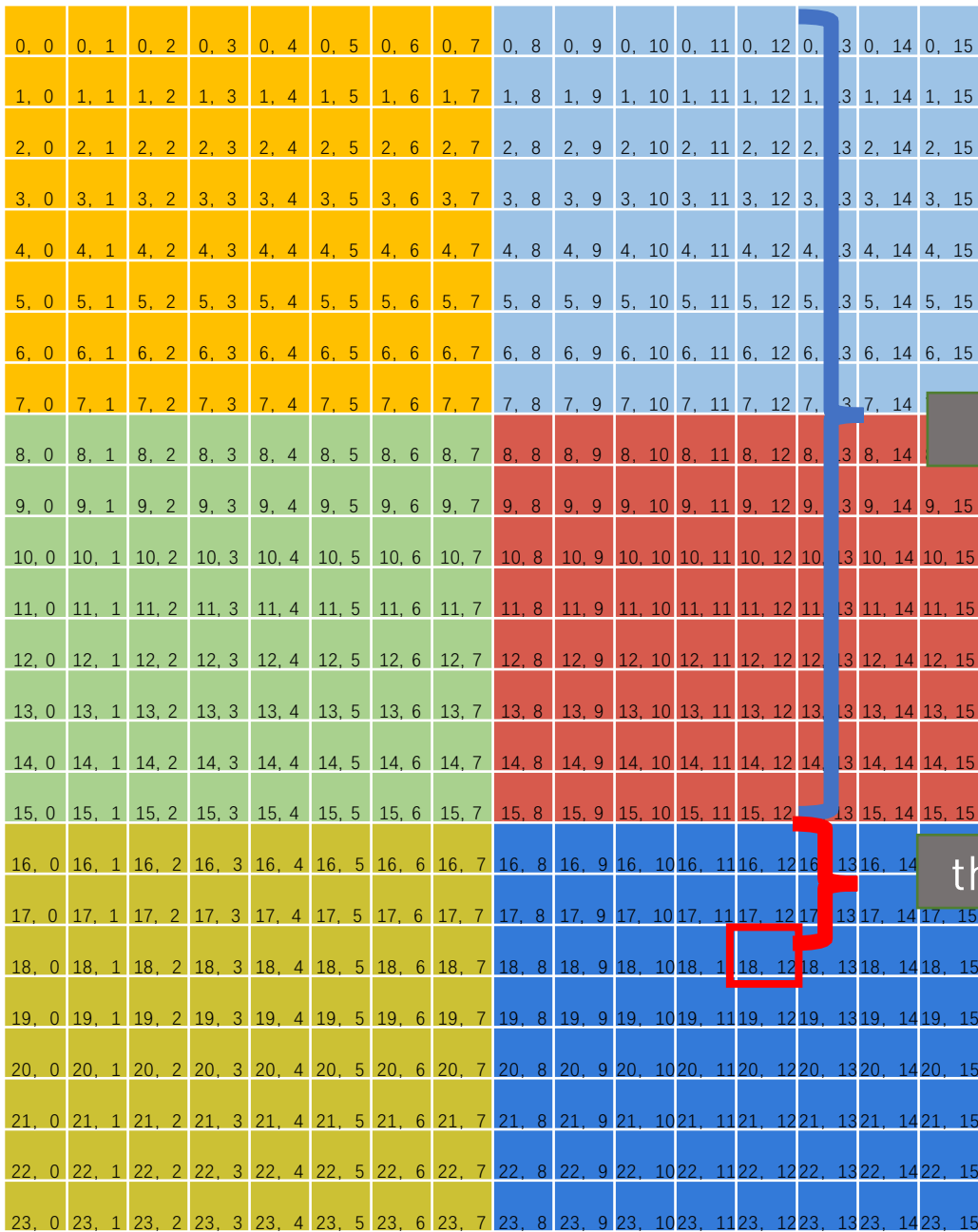
在当前的方阵中，我们可以认定绿色框中的数值与黑色框的数值相等，而它就是我们要处理的输出矩阵中对应元素的y坐标。

那么这里blockDim.x \* blockDim.x为什么不能写成blockDim.y \* blockDim.y?

在当前这个例子里我们确实可以这么认定，但是当我们要处理一个更大的矩阵时？**如下一页所示。**

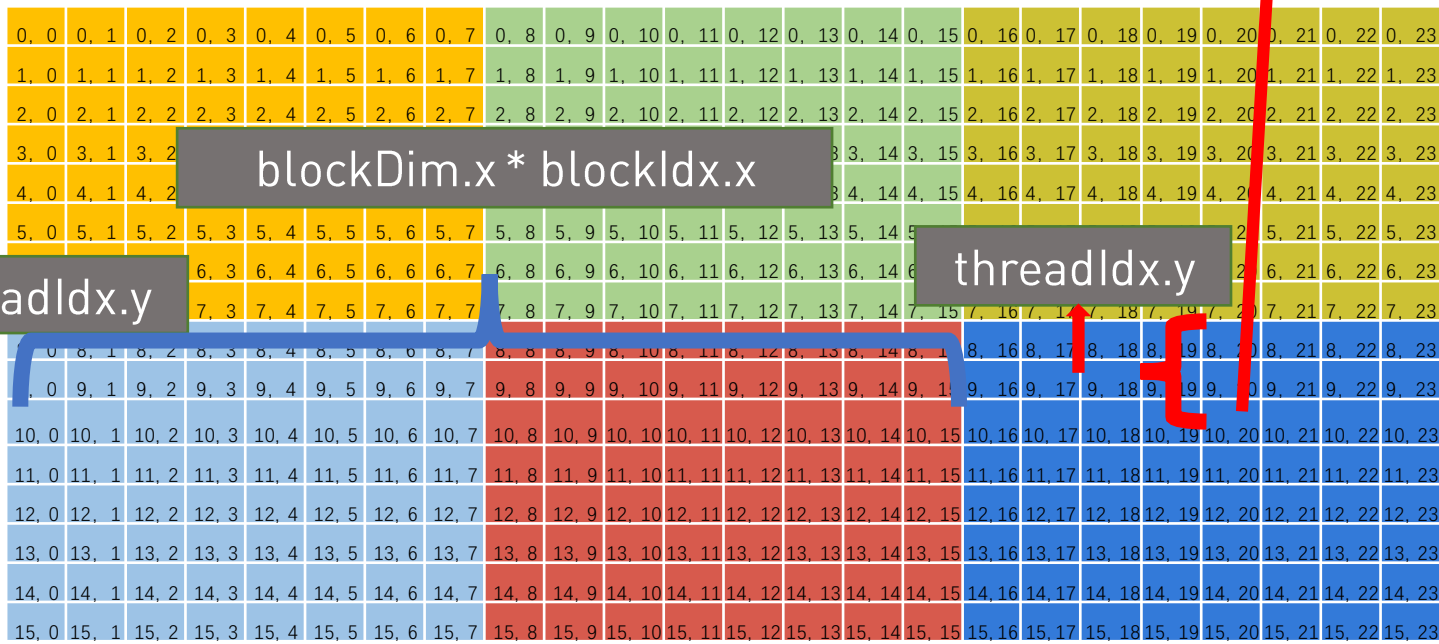


```
threadIdx.x == 3; threadIdx.y == 1;
blockIdx.x == 1; blockIdx.y == 0;
blockDim.x == 8; blockDim.y == 8;
gridDim.x == 2; gridDim.y == 2;
```



图二

当我们处理大一点矩阵的时候，如图所示，下图一位输入矩阵，下图二为它的输出矩阵。图一中红色箭头所指示的线程，要为图二中红色框圈住的位置的元素赋值。



图一

```
threadIdx.x == 4; threadIdx.y == 2;
blockIdx.x == 2; blockIdx.y == 1;
blockDim.x == 8; blockDim.y == 8;
gridDim.x == 3; gridDim.y == 2;
```

该线程的 `x==20, y==10`  
那么它将读取 `A[10][20]` 的数据，并把它放到他所在block中的shared memory中