【CUDA 基础】4.4 核函数可达到的带宽

2018-05-13 | CUDA , Freshman | 0 |

Abstract: 本文通过矩阵转置这一个例子，调整，优化核函数，使其达到最优的内存带宽

Keywords: 带宽，吞吐量，矩阵转置

核函数可达到的带宽

下面是废话，与本文知识无关，可以直接跳到下面红字处开始本文知识的学习。

废话继续，这两天没更新博客了，上一篇是转发的MIT人工智能实验室的研究指南，也就是告诉刚入学的研究生怎么做研究，要怎么积累，那篇文章发表在1988年，MIT的AI实验室网站目前仍然能检索的到，通读全文，感受很多，也学会了很多东西，当一个健康的框架搭好了以后，后面的好功能会源源不断的涌现，教育也是，当一套体系形成，那么就会有源源不断的人才和成果出现，相反，如果体系本身漏洞百出，根基不稳，短时间真的改不了，人也一样，价值观一旦确定，这个人的人生也就基本定型了——正所谓三岁看老。

今天废话有点多，如果没兴趣，可以直接跳到这里

上一章我们研究怎么通过调整线程网格结构和核函数来达到SM的最高利用率，今天我们来研究如何达到内存带宽的最大利用率。

还是要提那个老例子，但是说实话，这的很形象，也很有用，记住这个例子基本就能了解CUDA的优化大概要从哪入手了：

一条大路（内存读取总线）连接了工厂生产车间（GPU）和材料仓库（全局内存），生产车间又有很多的工作小组（SM），材料仓库有很多小库房（内存分块），工作小组同时生产相同的产品互不干扰（并行），我们有车从材料仓库开往工厂车间，什么时候发车，运输什么由工作小组远程电话指挥（内存请求），发车前，从材料仓库装货的时候，还要听从仓库管理员的分配，因为可能同一间库房可能只允许一个车来拿材料（内存块访问阻塞），然后这些车单向的开往工厂，这时候就是交通问题了，如果我们的路是单向（从仓库到工厂）8车道，每秒钟能通过16辆车，那么我们把这个指标称为带宽。当然我们还有一条路是将成品运输到成品仓库，这也是一条路，与原料库互不干扰，和材料仓库到工厂的路一样，也有宽度，也是单向的，如果这条路堵住，和仓库到工厂的路堵住一样，此时工厂要停工等待。

最理想的状态是，路上全是车，并且全都高速行驶，工厂里的所有工人都在满负荷工作，没有等待，这就是优化的最终目标，如果这个目标达到了，还想进一步提高效率，那么你就只能优化你的工艺了（算法）

上面的这个就是粗糙的GPU工作过程。例子还是比较贴切的，但是有点描述粗糙，多读两遍应该会有点收获的。

内存延迟是影响核函数的一大关键，内存延迟，也就是从你发起内存请求到数据进入SM的寄存器的整个时间。

内存带宽，也就是SM访问内存的速度，它以单位时间内传输的字节数进行测量。

上一节我们用了两种方法改善内核性能：

- 最大化线程束的数量来隐藏内存延迟，维持更多的正在执行的内存访问达到更好的总线利用率

- 通过适当的对齐和合并访问，提高带宽效率

然而，当前内核本身的内存访问方式就有问题，上面两种优化相当于给一个拖拉机优化空气动力学外观，杯水车薪。
我们本文要做的就是看看这个核函数对应的问题，其极限效率是多少，在理想效率之下，我们来进行优化，我们本文那矩阵转置来进行研究，看看如何把一种看起来没办法优化的内核，重新设计让它达到更好的性能。

内存带宽

多数内核对带宽敏感，也就是说，工人们生产效率特别高，而原料来的很慢，这限制了生产速度。去哪聚内存中数据的安排方式和线程束的访问方式都对带宽有显著影响。一般有如下两种带宽

- 理论带宽
- 有效带宽

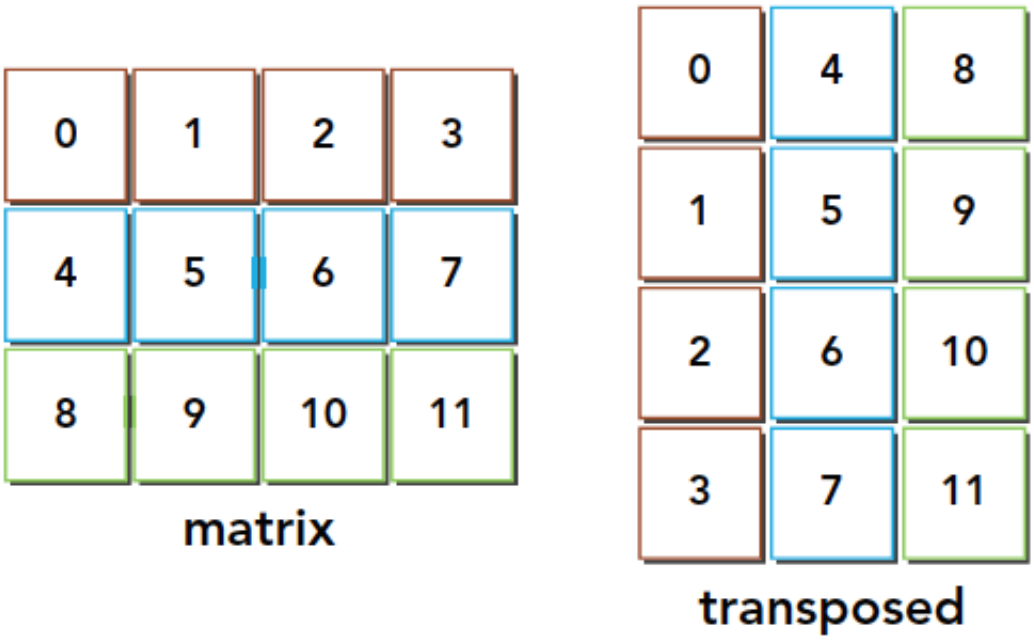
理论带宽就是硬件设计的绝对最大值，硬件限制了这个最大值为多少，比如对于不使用ECC的Fermi M2090来说，理论峰值 117.6 GB/s
有效带宽是核函数实际达到的带宽，是测量带宽，可以用下面公式计算:

$$\text{有效带宽} = \frac{(\text{读字节数} + \text{写字节数}) \times 10^{-9}}{\text{运行时间}} \tag{1}$$

注意吞吐量和带宽的区别，吞吐量是衡量计算核心效率的，用的单位是每秒多少十亿次浮点运算(gflops)，有效吞吐量其不止和有效带宽有关，还和带宽的利用率等因素有关，当然最主要的还是设备的运算核心。当然，也有内存吞吐量这种说法这种说法就是单位时间上内存访问的总量，用单位 GB/s 表示，这个值越大表示读取到的数据越多，但是这些数据不一定是有用的。
接下来我们研究如何调整核函数来提高有效带宽

矩阵转置问题

矩阵转置(点击查看详情)就是交换矩阵的坐标，我们本文研究有二维矩阵，转置结果如下:

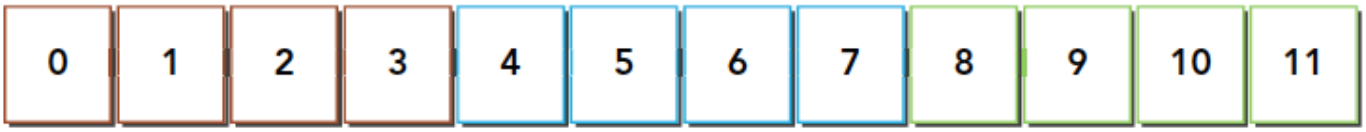


使用串行编程很容易实现：

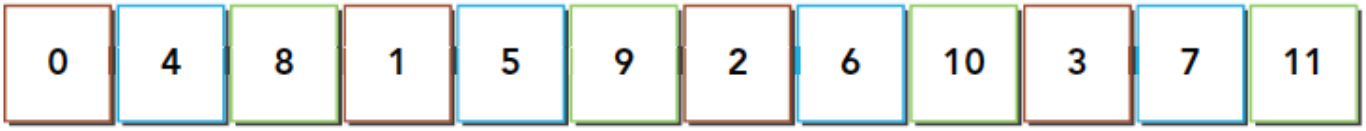
```
void transformMatrix2D_CPU(float * MatA,float * MatB,int nx,int ny)
{
  for(int j=0;j<ny;j++)
  {
    for(int i=0;i<nx;i++)
    {
      MatB[i*nx+j]=MatA[j*nx+i];
    }
  }
}
```

这段代码应该比较容易懂，这是串行解决的方法，必须要注意的是，我们所有的数据，结构体也好，数组也好，多维数组也好，所有的数据，在内存硬件层面都是一维排布的，所以我们这里也是使用一维的数组作为输入输出，那么从真实的角度看内存中的数据就是下面这样的：

data layout of original matrix



data layout of transposed matrix



通过这个图能得出一个结论，转置操作：

- 读：原矩阵行进行读取，请求的内存是连续的，可以进行合并访问
- 写：写到转置矩阵的列中，访问是交叉的

图中的颜色需要大家注意一下，读的过程同一颜色可以看成是合并读取的，但是转置发生后写入的过程，是交叉的。

交叉访问是使得内存访问变差的罪魁。但是作为矩阵转置本身，这个是无法避免的。但是在这种无法避免的交叉访问前提下，我们怎么能提升效率就变成了一个有趣的课题。

我们接下来所有方法都会有按照行读取和按照列读取的版本，来对比效率，看看是交叉读有优势，还是交叉写有优势。

如果按照我们上文的观点，如果按照下面两种方法进行读

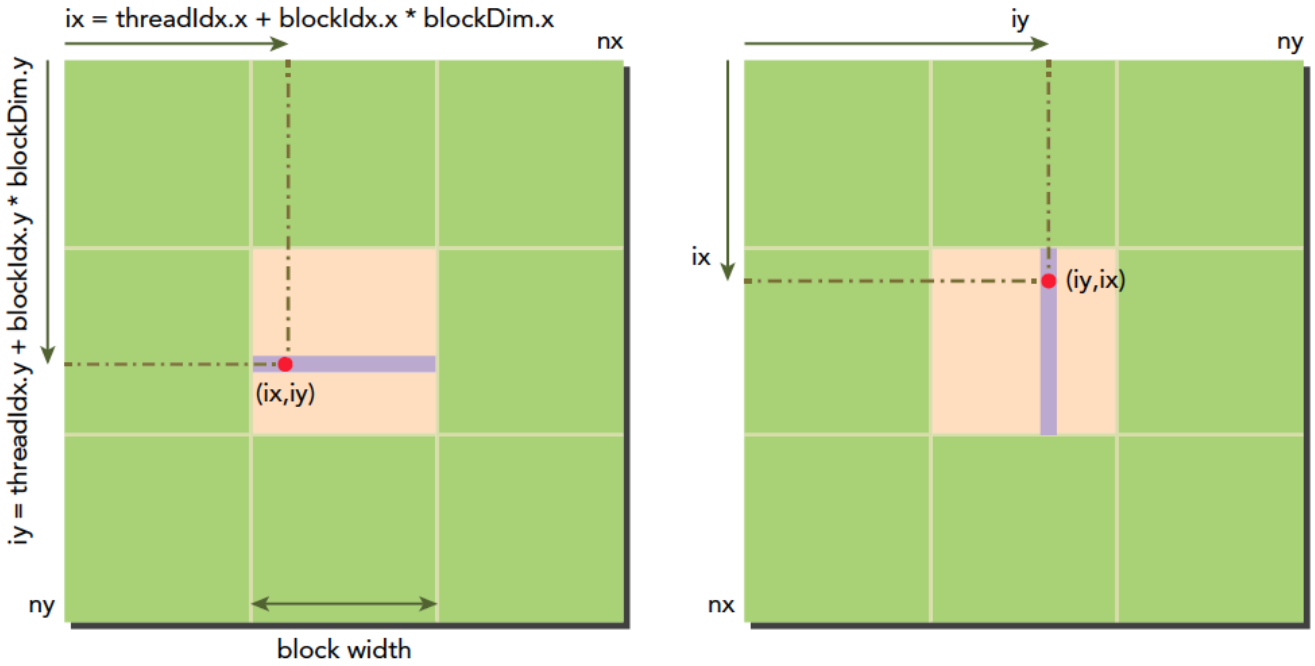


FIGURE 4-25

最初的想法肯定是：按照图一合并读更有效率，因为写的时候不需要经过一级缓存，所以对于有一级缓存的程序，合并的读取应该是更有效率的。如果你这么想，恭喜你，你想的不对（我当时也是这么想的）。我们需要补充下关于一级缓存的作用，上文我们讲到合并，可能第一印象就是一级缓存是缓冲从全局内存里过来的数据一样，但是我们忽略了一些东西，就是内存发起加载请求的时候，会现在一级缓存里看看有没有这个数据，如果有，这个就是一个命中，这和CPU的缓存运行原理是一样的，如果命中了，就不需要再去全局内存读了，如果用在上面这个例子，虽然按照列读是不合并的，但是使用一级缓存加载过来的数据在后面会被使用，我们必须要注意虽然，一级缓存一次读取128字节的数据，其中只有一个单位是有用的，但是剩下的并不会被马上覆盖，粒度是128字节，但是一级缓存的大小有几k或是更大，这些数据很有可能不会被替换，所以，我们按列读取数据，虽然第一行只用了一个，但是下一列的时候，理想情况是所有需要读取的元素都在一级缓存中，这时候，数据直接从缓存里面读取，美滋滋！

为转置核函数设置上限和下限

在优化之前，我们要给自己一个目标，也就是理论上极限是多少，比如我们测得理论极限是10，而我们已经花了一天时间优化到了9.8，就没必要再花10天优化到9.9了，因为这已经很接近极限了，如果不知道极限，那么就会在无限的接近中浪费时间。

我们本例子中的瓶颈在交叉访问，所以我们假设没有交叉访问，和全是交叉访问的情况，来给出上限和下限：

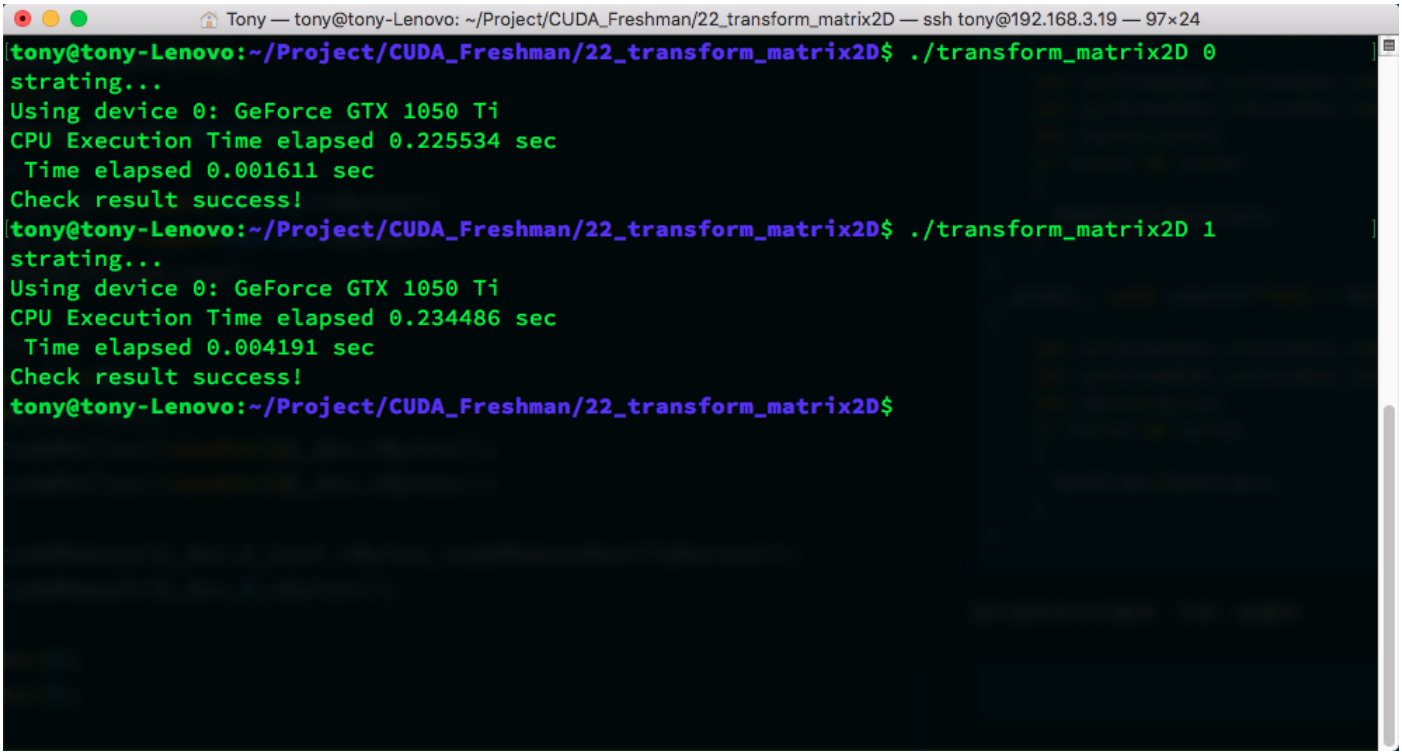
- 行读取，行存储来复制矩阵(上限)

- 列读取，列存储来复制矩阵(下限)

```
__global__ void copyRow(float * MatA,float * MatB,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx=ix+iy*nx;
    if (ix<nx && iy<ny)
    {
      MatB[idx]=MatA[idx];
    }
}
__global__ void copyCol(float * MatA,float * MatB,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx=ix*ny+iy;
    if (ix<nx && iy<ny)
    {
      MatB[idx]=MatA[idx];
    }
}
```

我们使用命令行编译，开启一级缓存：

```
nvcc -O3 -arch=sm_35 -Xptxas -dlcm=ca -I ../include/ transform_matrix2D.cu -o transf
```



可以得到：

核函数	试验1	试验2	试验3	平均值
上限	0.001611	0.001614	0.001606	0.001610
下限	0.004191	0.004210	0.004205	0.004202

这个时间是三次测试出来的平均值，基本可以肯定在当前数据规模下，上限在0.001610s，下限在0.004202s，不可能超过上限，当然如果你能跌破下限也算是人才了！

另外，我们我们全文用的主函数我只在此列举一次，完成代码库在https://github.com/Tony-Tan/CUDA_Freshman

```
1  int main(int argc,char** argv)
2  {
3    printf("strating...\n");
4    initDevice(0);
5    int nx=1<<12;
6    int ny=1<<12;
7    int nxy=nx*ny;
8    int nBytes=nxy*sizeof(float);
```

```
    int transform_kernel=0;
    if(argc>=2)
      transform_kernel=atoi(argv[1]);
    //Malloc
    float* A_host=(float*)malloc(nBytes);
    float* B_host=(float*)malloc(nBytes);
    initialData(A_host,nxy);

    //cudaMalloc
    float *A_dev=NULL;
    float *B_dev=NULL;
    CHECK(cudaMalloc((void**)&A_dev,nBytes));
    CHECK(cudaMalloc((void**)&B_dev,nBytes));

    CHECK(cudaMemcpy(A_dev,A_host,nBytes,cudaMemcpyHostToDevice));
    CHECK(cudaMemset(B_dev,0,nBytes));

    int dimx=32;
    int dimy=32;

    // cpu compute
    double iStart=cpuSecond();
    transformMatrix2D_CPU(A_host,B_host,nx,ny);
    double iElaps=cpuSecond()-iStart;
    printf("CPU Execution Time elapsed %f sec\n",iElaps);

    // 2d block and 2d grid
    dim3 block(dimx,dimy);
    dim3 grid((nx-1)/block.x+1,(ny-1)/block.y+1);
    dim3 block_1(dimx,dimy);
    dim3 grid_1((nx-1)/(block_1.x*4)+1,(ny-1)/block_1.y+1);
    iStart=cpuSecond();
    switch(transform_kernel)
    {
    case 0:
      copyRow<<<grid,block>>>(A_dev,B_dev,nx,ny);
      break;
    case 1:
      copyCol<<<grid,block>>>(A_dev,B_dev,nx,ny);
      break;
    case 2:
      transformNaiveRow<<<grid,block>>>(A_dev,B_dev,nx,ny);
      break;
    case 3:
```

```
            transformNaiveCol<<<grid,block>>>(A_dev,B_dev,nx,ny);
            break;
    case 4:
            transformNaiveColUnroll<<<grid_1,block_1>>>(A_dev,B_dev,nx,ny);
            break;
    case 5:

            transformNaiveColUnroll<<<grid_1,block_1>>>(A_dev,B_dev,nx,ny);
            break;
    case 6:
            transformNaiveRowDiagonal<<<grid,block>>>(A_dev,B_dev,nx,ny);
            break;
    case 7:
            transformNaiveColDiagonal<<<grid,block>>>(A_dev,B_dev,nx,ny);
            break;
    default:
      break;
    }
    CHECK(cudaDeviceSynchronize());
    iElaps=cpuSecond()-iStart;
    printf(" Time elapsed %f sec\n",iElaps);
    CHECK(cudaMemcpy(B_host,B_dev,nBytes,cudaMemcpyDeviceToHost));
    checkResult(B_host,B_host,nxy);

    cudaFree(A_dev);
    cudaFree(B_dev);
    free(A_host);
    free(B_host);
    cudaDeviceReset();
    return 0;
}
```

switch部分可以写成函数指针的方式，但是问题不大（原文写的应该是函数指针的方式）。

我的笔记本是1050ti的显卡，这个表可能是主机版本的1050ti的指标，可以看出其理论贷款是112GB/s

Specification

http://www.expreview.com (last update: 26/10/2016)

Model	NVIDIA GeForce GTX 950	NVIDIA GeForce GTX 960	NVIDIA GeForce GTX 1050	NVIDIA GeForce GTX 1050Ti	NVIDIA GeForce GTX 1060 3GB
Core name	GM206-250	GM206-300	GP107-300	GP107-400	GP106-300
Transisitor Count	2.94B	2.94B	3.3B	3.3B	4.4B
Die Size	227mm^2	227mm^2	132mm^2	132mm^2	200mm^2
Manufacturing Process	28nm	28nm	14nm	14nm	16nm
Stream Processors	768	1024	640	768	1152
TA/TF	48	64	40	48	72
ROP/RBE	32	32	32	32	48
Core Clock	1024MHz	1127MHz	1354MHz	1290MHz	1506MHz
Boost Clock	1188MHz	1178MHz	1455MHz	1392MHz	1708MHz
Memory Clock	6.6Gbps	7Gbps	7Gbps	7Gbps	8Gbps
Single Precision	1.95TFLOPS	2.6TFLOPS	1.86TFLOPS	2.13TFLOPS	4.0TFLOPS
Texture Fillrate	49.1GT/s	72GT/s	54.2GT/s	61.9GT/s	108.4GT/s
Memory Size	2GB	2GB	2GB	4GB	3GB
Memory Interface	128bit GDDR5	128bit GDDR5	128bit GDDR5	128bit GDDR5	192bit GDDR5
Memory Bandwidth	105.6GB/s	112GB/s	112GB/s	112GB/s	192GB/s
PCI-E Version	3	3	3	3	3
DirectX	DX 12（FL_12.1）	DX 12（FL_12.1）	DX 12（FL_12.1）	DX 12（FL_12.1）	DX 12（FL_12.1）
Conectors	HDMI 2.0*1 DVI DL*1 DisplayPort 1.2*3	HDMI 2.0*1 DVI DL*1 DisplayPort 1.2*3	DL-DVI*1 HDMI 2.0b*1 DisplayPort 1.4*1	DL-DVI*1 HDMI 2.0b*1 DisplayPort 1.4*1	DL-DVI HDMI 2.0b*1 DisplayPort 1.4*3
TDP	90W	120W	75W	75W	120W
Power Connecter	6Pin	6Pin	N/A	N/A	6Pin
MSRP	￥1199	￥1499	￥899	￥1199	￥1599

我们使用公式(1)来算一下两种极限的带宽：

$$\text{copyRow} = \frac{1 \times 2^{12+12} \times 4 \times 2 \times 10^{-9}}{0.001610} = \frac{0.134217728}{0.001610} = 83.3650 \text{ GB/s}$$

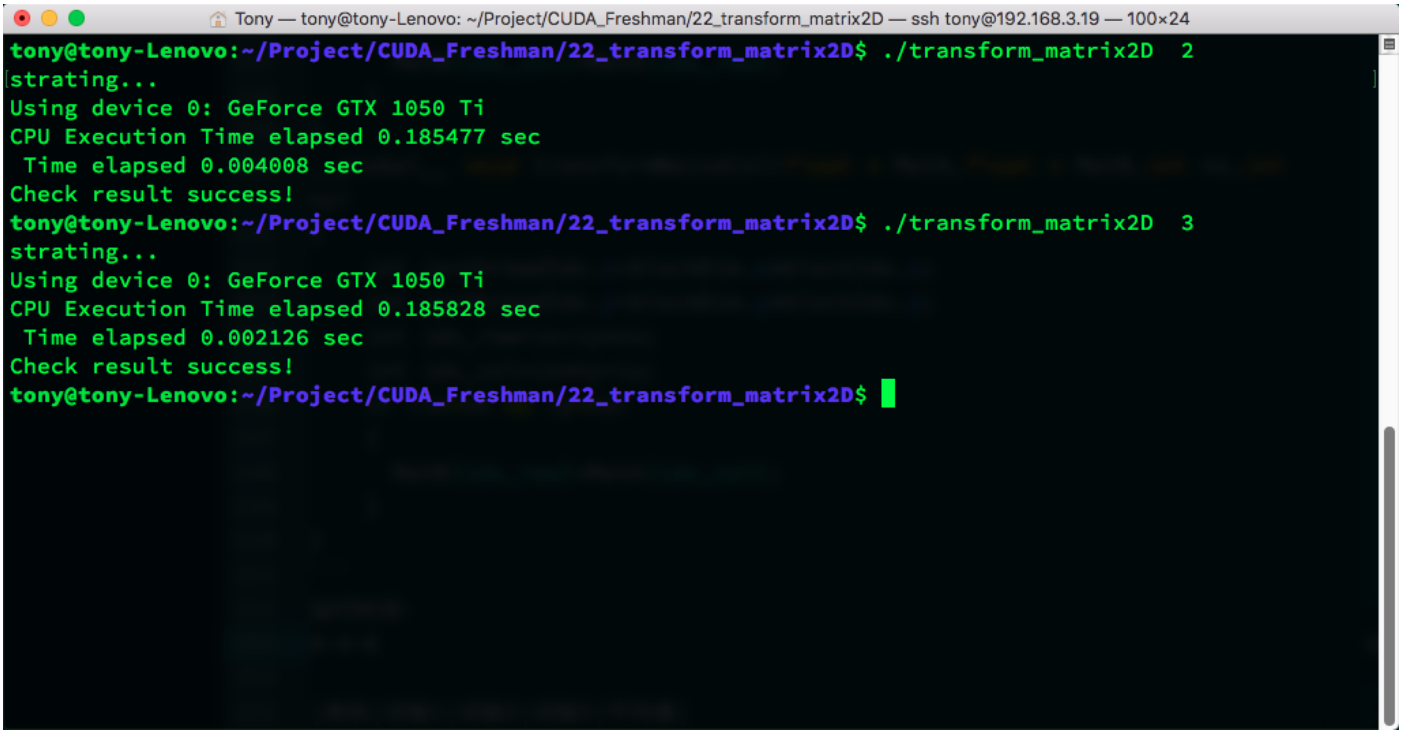
$$\text{copyCol} = \frac{1 \times 2^{12+12} \times 4 \times 2 \times 10^{-9}}{0.004202} = \frac{0.134217728}{0.004202} = 31.9414 \text{ GB/s}$$

朴素转置：读取行与读取列

接下来我们看最naive的两种转置方法，不加任何优化，也就是我们一瞬间就想到的方案：

```
__global__ void transformNaiveRow(float * MatA,float * MatB,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix<nx && iy<ny)
    {
      MatB[idx_col]=MatA[idx_row];
    }
}
__global__ void transformNaiveCol(float * MatA,float * MatB,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix<nx && iy<ny)
    {
      MatB[idx_row]=MatA[idx_col];
    }
}
```

运行时间：



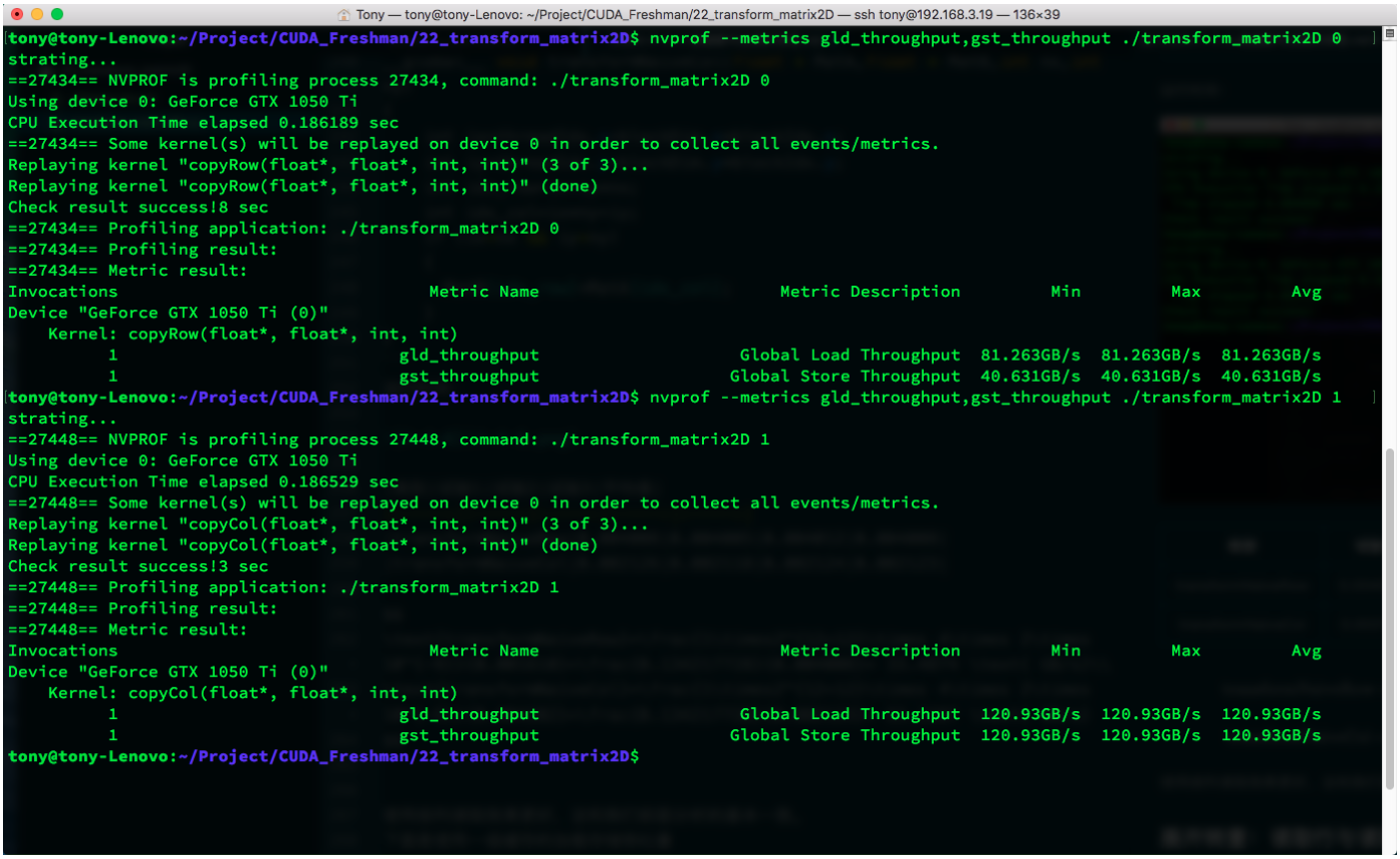
核函数	试验1	试验2	试验3	平均值
transformNaiveRow	0.004008	0.004005	0.004012	0.004008
transformNaiveCol	0.002126	0.002118	0.002124	0.002123

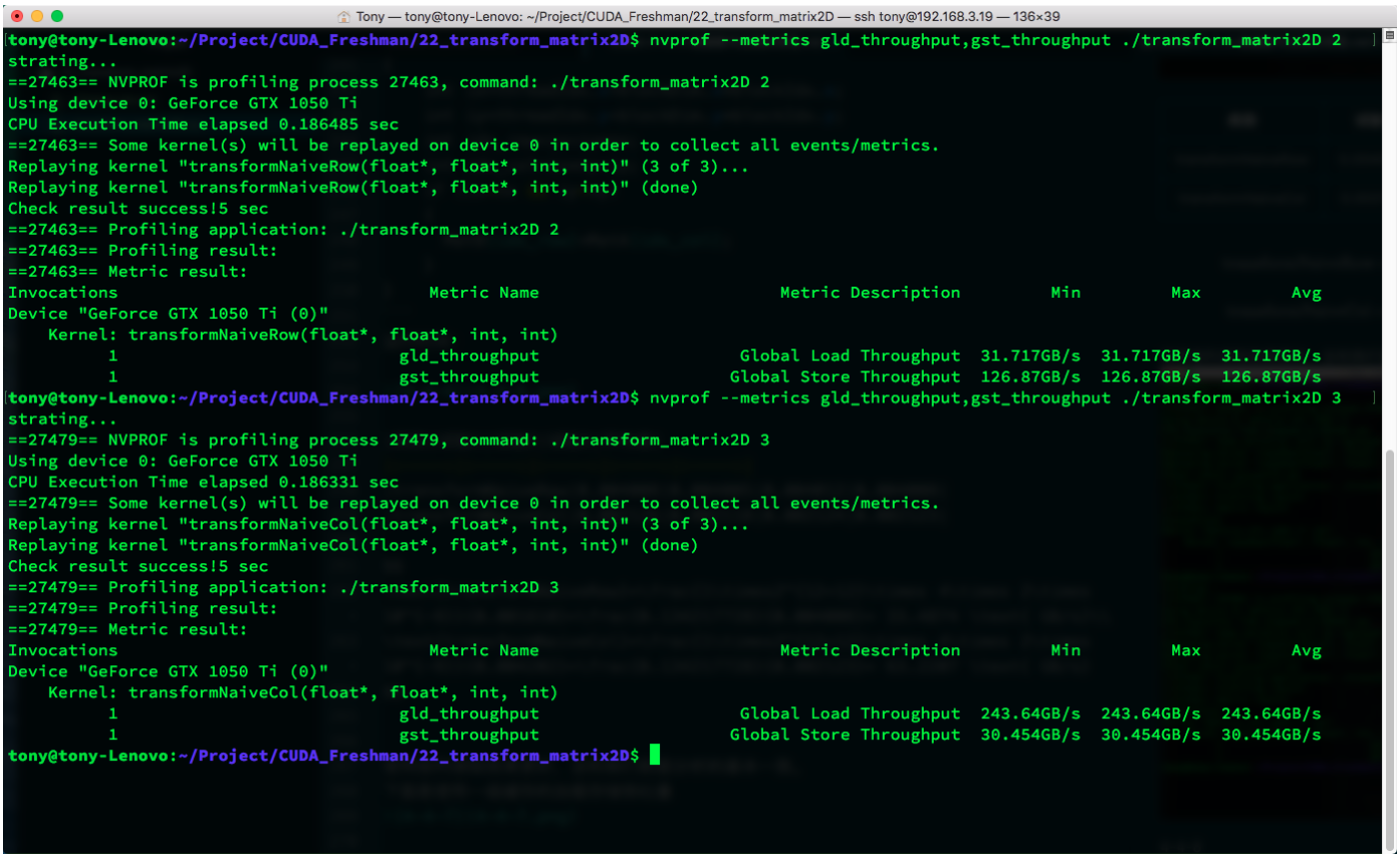
$$\text{transformNaiveRow} = \frac{1 \times 2^{12+12} \times 4 \times 2 \times 10^{-9}}{0.001610} = \frac{0.134217728}{0.004008} = 33.4874\ \text{GB/s}$$

$$\text{transformNaiveCol} = \frac{1 \times 2^{12+12} \times 4 \times 2 \times 10^{-9}}{0.004202} = \frac{0.134217728}{0.002123} = 63.2207\ \text{GB/s}$$

使用按列读取效果更好，这和我们前面分析的基本一致。

下面是使用一级缓存的加载存储吞吐量





核函数	加载吞吐量	存储吞吐量

copyRow	81.263	40.631
copyCol	120.93	120.93
transformNaiveRow	31.717	126.87
transformNaiveCol	243.64	30.454

按列读取的高吞吐量的原因就是上面我们说的缓存命中，这里也能看到吞吐量是可以超过带宽的，因为带宽衡量的是从全局内存到SM的速度极限，而吞吐量是SM获得数据的总量除以时间，而这些数据可以来自一级缓存，而不必千里迢迢从主存读取。

这里有个疑问：虽然交叉读取缓存命中率高了，但是似乎并没有减少从主存读取数据的数据量，那为什么速度会有提高呢?

我认为应该是延迟隐藏部分出的问题，导致了交叉读取效率变高，当然只是我的猜测后面还要验证一下。

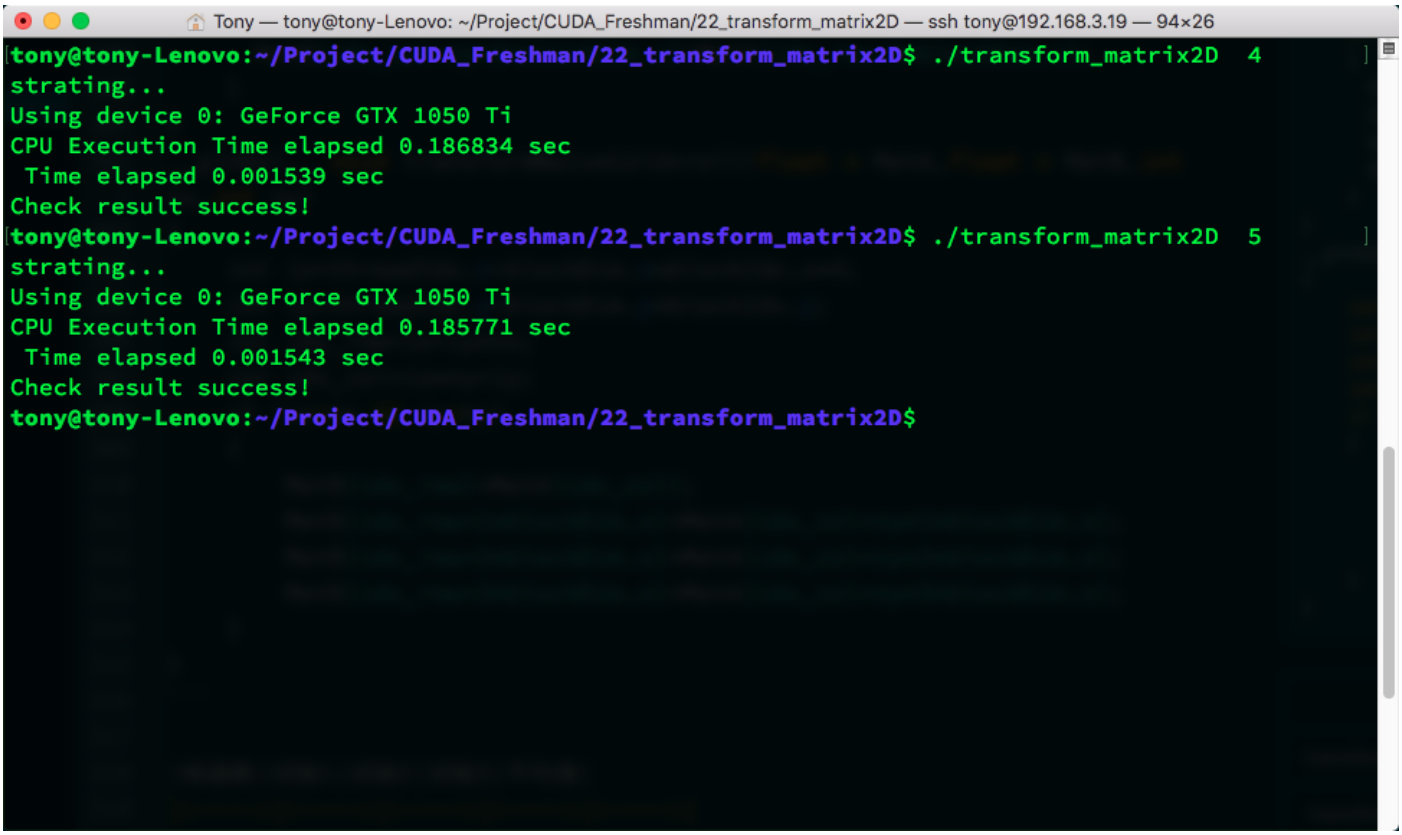
展开转置：读取行与读取列

接下来这个是老套路了，有效地隐藏延迟，从展开操作开始：

```
__global__ void transformNaiveRowUnroll(float * MatA,float * MatB,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x*4;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix<nx && iy<ny)
    {
      MatB[idx_col]=MatA[idx_row];
      MatB[idx_col+ny*1*blockDim.x]=MatA[idx_row+1*blockDim.x];
      MatB[idx_col+ny*2*blockDim.x]=MatA[idx_row+2*blockDim.x];
      MatB[idx_col+ny*3*blockDim.x]=MatA[idx_row+3*blockDim.x];
    }
}
__global__ void transformNaiveColUnroll(float * MatA,float * MatB,int nx,int ny)
{
    int ix=threadIdx.x+blockDim.x*blockIdx.x*4;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
```

```
        if (ix<nx && iy<ny)
        {
            MatB[idx_row]=MatA[idx_col];
            MatB[idx_row+1*blockDim.x]=MatA[idx_col+ny*1*blockDim.x];
            MatB[idx_row+2*blockDim.x]=MatA[idx_col+ny*2*blockDim.x];
            MatB[idx_row+3*blockDim.x]=MatA[idx_col+ny*3*blockDim.x];
        }
    }
```

结果如图



核函数	试验1	试验2	试验3	平均值
transformNaiveRowUnroll	0.001544	0.001550	001541	0.001545
transformNaiveColUnroll	0.001545	0.001539	0.001546	0.001543

这里出现了尴尬的一幕，没错，我们突破上限了，上限是按行合并读取，合并存储，不存在交叉的情况，这种理想情况不可能发生在转置中，所以我们说这是上限。而我们使用展开的交叉访问居然得到了比上限更快的速度，所以我断定，如果把上限展开，速度肯定会更快，但是我们这里还把他叫做上限，虽然并不

是真正的上限。

想要知道真正的上限是什么，就要从硬件角度算理论上限，实际测出来的上限很有可能不正确。

对角转置：读取行与读取列

接下来我们使用一点新技巧，这个技巧的来源是DRAM的特性导致的，还记得我们例子中对原料仓库的描述么，那里面有很多小库房，这些小库房同时可能只允许一台车拿东西，在DRAM中内存是分区规划的，如果过多的访问同一个区，会产生排队的现象，也就是要等待，为了避免这种情况，我们最好均匀的访问DRAM的某一段，DRAM的分区是每256个字节算一个分区，所以我们最好错开同一个分区的访问，方法就是调整块的ID，这时候你可能有问题了，我们并不知道块的执行顺序，那应该怎么调呢，这个问题没有啥官方解释，我自己的理解是，硬件执行线程块必然是按照某种规则进行的，按照123执行，可能要比按照随机执行好，因为想要随机执行，还要有生成随机顺序这一步，根本没必要，我们之所以说块的执行顺序不确定，其实是为了避免大家把它理解为确定顺序，而实际上可能有某些原因导致顺序错乱，但是这个绝对不是硬件设计时故意而为之的。

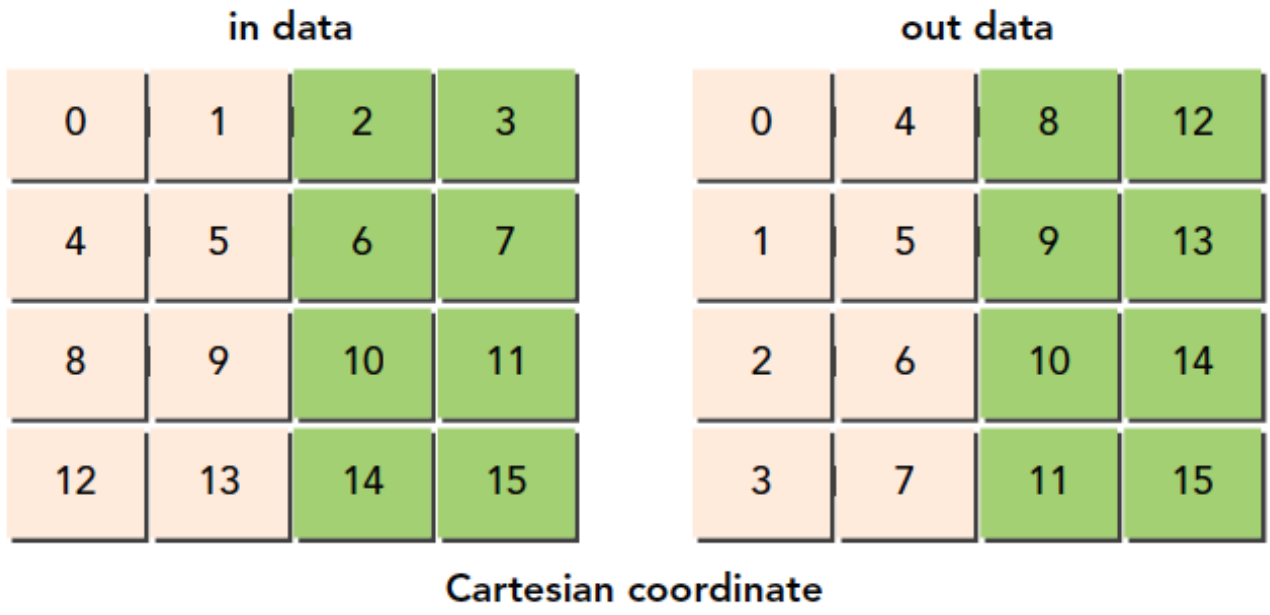
我们这个对角转置的目的就是使得读取DRAM位置均匀一点，别都集中在一个分区上，方法是打乱线程块，因为连续的线程块可能访问相近的DRAM地址。

我们的方案是使用一个函数 $f(x,y)=(m,n)$ 一个一一对应的函数，将原始笛卡尔坐标打乱。

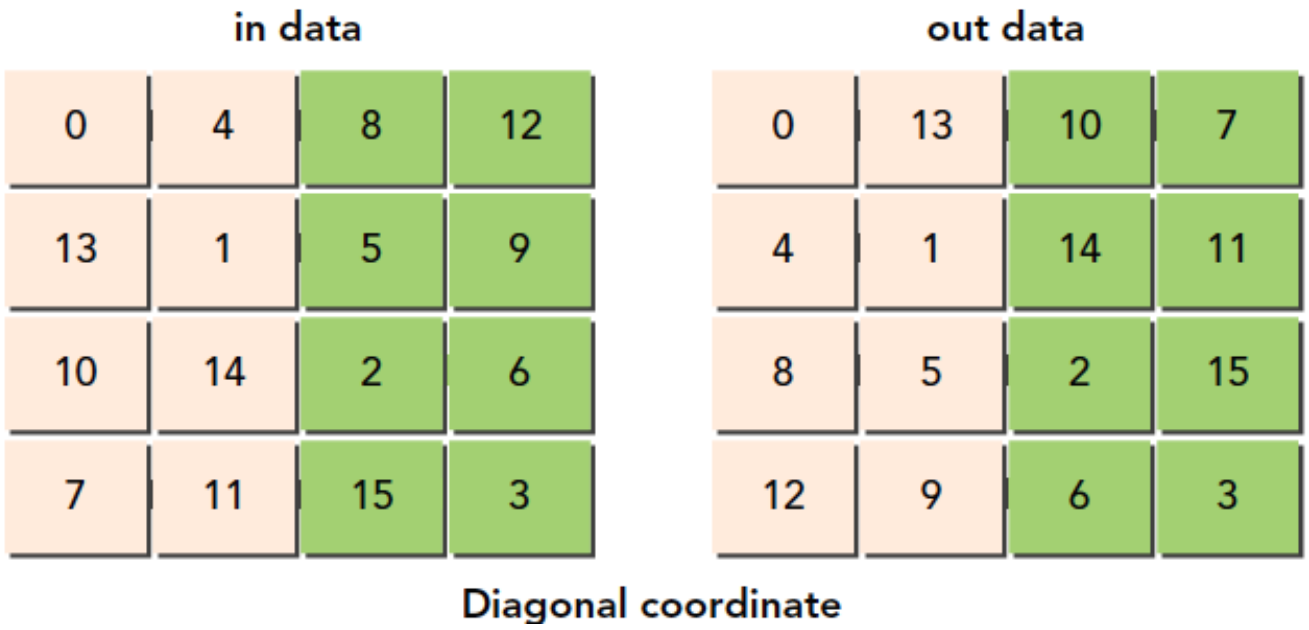
注意，所有这些线程块的顺序什么的都是在编程模型基础上的，跟硬件没什么关系，这些都是逻辑层面的，实际上线程块ID对应的是哪个线程块也是我们自己规定的而已。

说实话，这个代码有点难理解，当然你也不用死记硬背这种用法，似乎没有程序员被代码，甚至入门的过程都不用背，我们要理解的就是线程块ID和线程块之间的对应，以及新ID和原始ID的对应，以及新ID对应的块，

原始的线程块ID

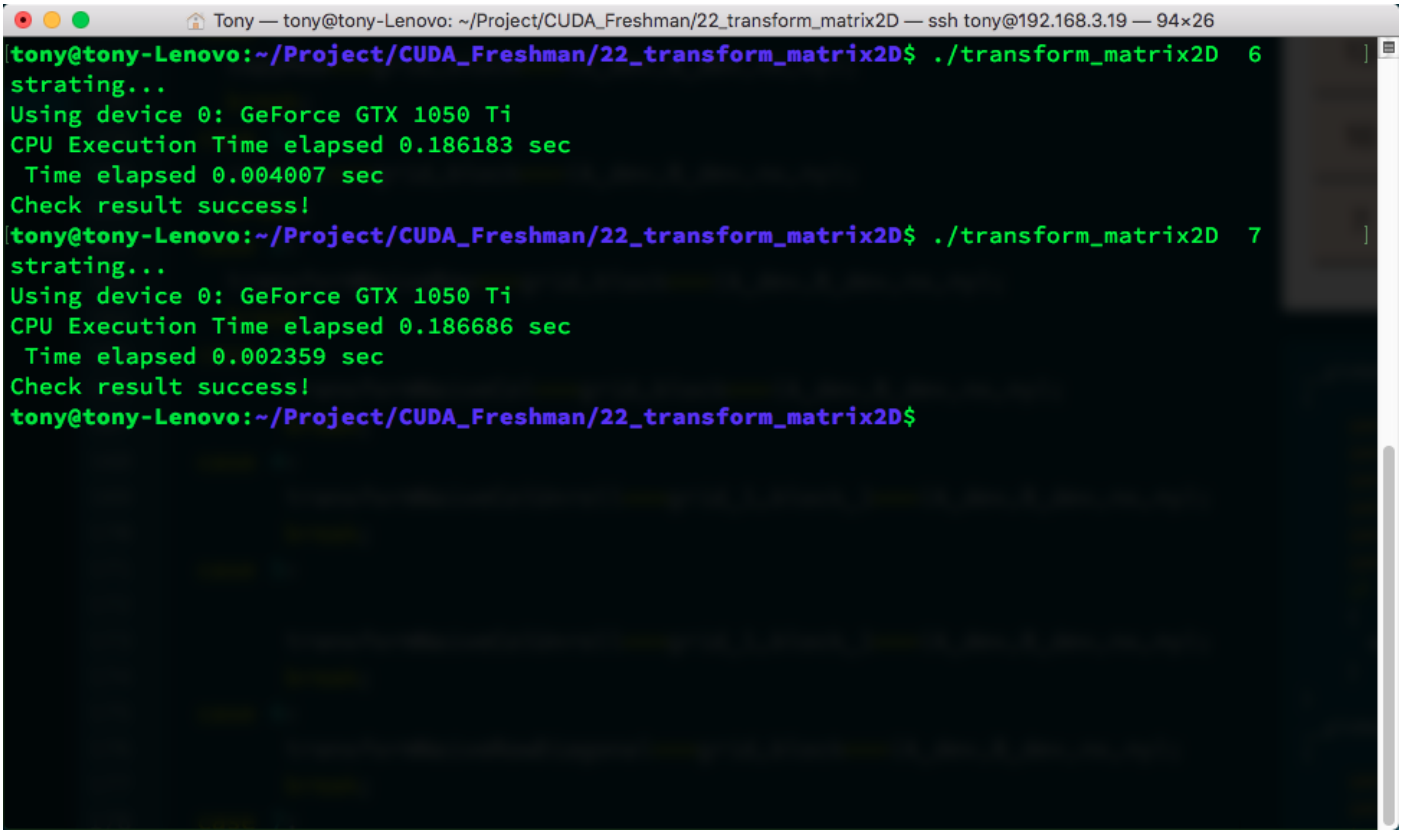


新设计的线程块ID




```
__global__ void transformNaiveRowDiagonal(float * MatA,float * MatB,int nx,int ny)
{
    int block_y=blockIdx.x;
    int block_x=(blockIdx.x+blockIdx.y)%gridDim.x;
    int ix=threadIdx.x+blockDim.x*block_x;
    int iy=threadIdx.y+blockDim.y*block_y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix<nx && iy<ny)
```

```
    {
      MatB[idx_col]=MatA[idx_row];
    }
}
__global__ void transformNaiveColDiagonal(float * MatA,float * MatB,int nx,int ny)
{
    int block_y=blockIdx.x;
    int block_x=(blockIdx.x+blockIdx.y)%gridDim.x;
    int ix=threadIdx.x+blockDim.x*block_x;
    int iy=threadIdx.y+blockDim.y*block_y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix<nx && iy<ny)
    {
      MatB[idx_row]=MatA[idx_col];
    }
}
```



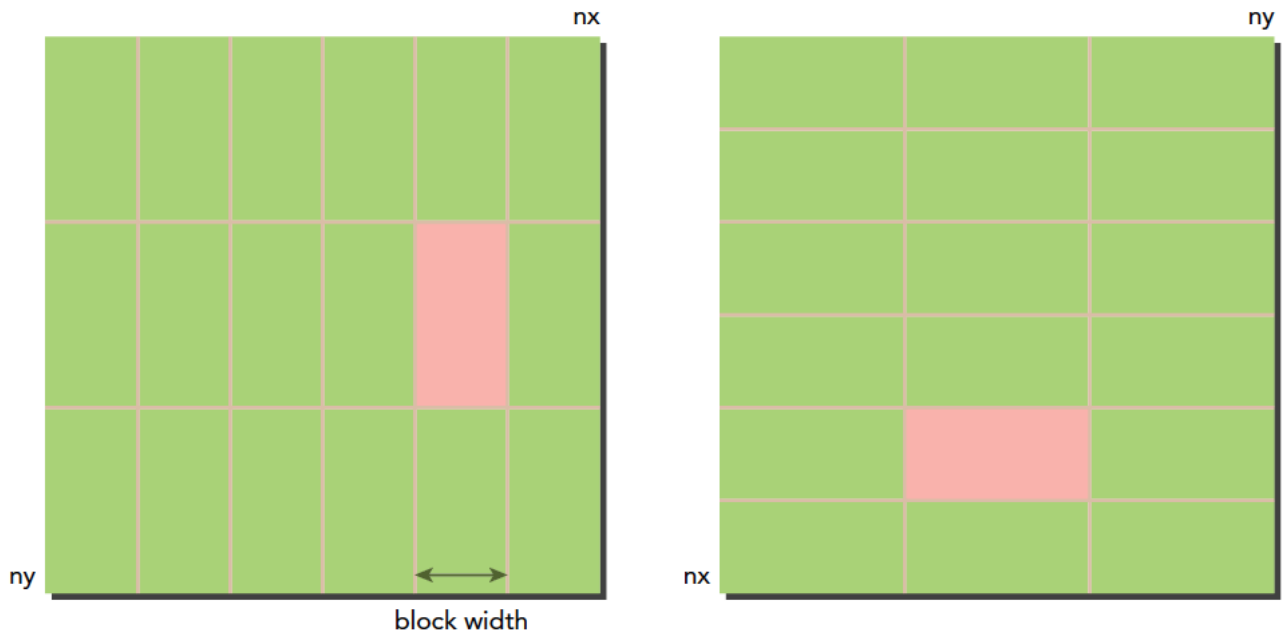
这个速度还没有展开的版本快，甚至没有naive的交叉读取速度快，但书上说的是效率有提高，可能是CUDA升级后的原因吧，或者其他原因的影响，但是DRAM分区会出现排队这种现象值得注意。

瘦块来增加并行性

接下来老套路，调整一下线程块的尺寸我们看看有没有啥变化，当然，我们以naive的列读取作为对照。

block尺寸	测试1	测试2	测试3	平均值
(32,32)	0.002166	0.002122	0.002125	0.002138
(32,16)	0.001677	0.001696	0.001703	0.001692
(32,8)	0.001925	0.001929	0.001925	0.001926
(64,16)	0.002117	0.002146	0.002113	0.002125
(64,8)	0.001949	0.001945	0.001945	0.001946
(128,8)	0.002228	0.002230	0.002229	0.002229

这是简单的实验结果，可见（32，16）的这种模式效率最高



总结

本文主要讲解内存带宽对效率的影响以及如何有效地通过调整读取方式来突破内存存储瓶颈，这是我们优化CUDA程序非常重要的手段