一直很好奇GPU做矩阵运算是怎么并行加速的，今天看了一些粗浅的东西，并总结整理出来。

version：cuda 8

cuda C 中扩展的一些概念

主要包括函数声明、变量声明、内存类型声明、文理内存、原子函数等，常用的有这么几个：

参考（http://bbs.csdn.net/topics/390798229，原地址已经失效）

- 主机

将CPU及系统的内存（内存条）称为主机。

- 设备

将GPU及GPU本身的显示内存称为设备。

- 线程(Thread)

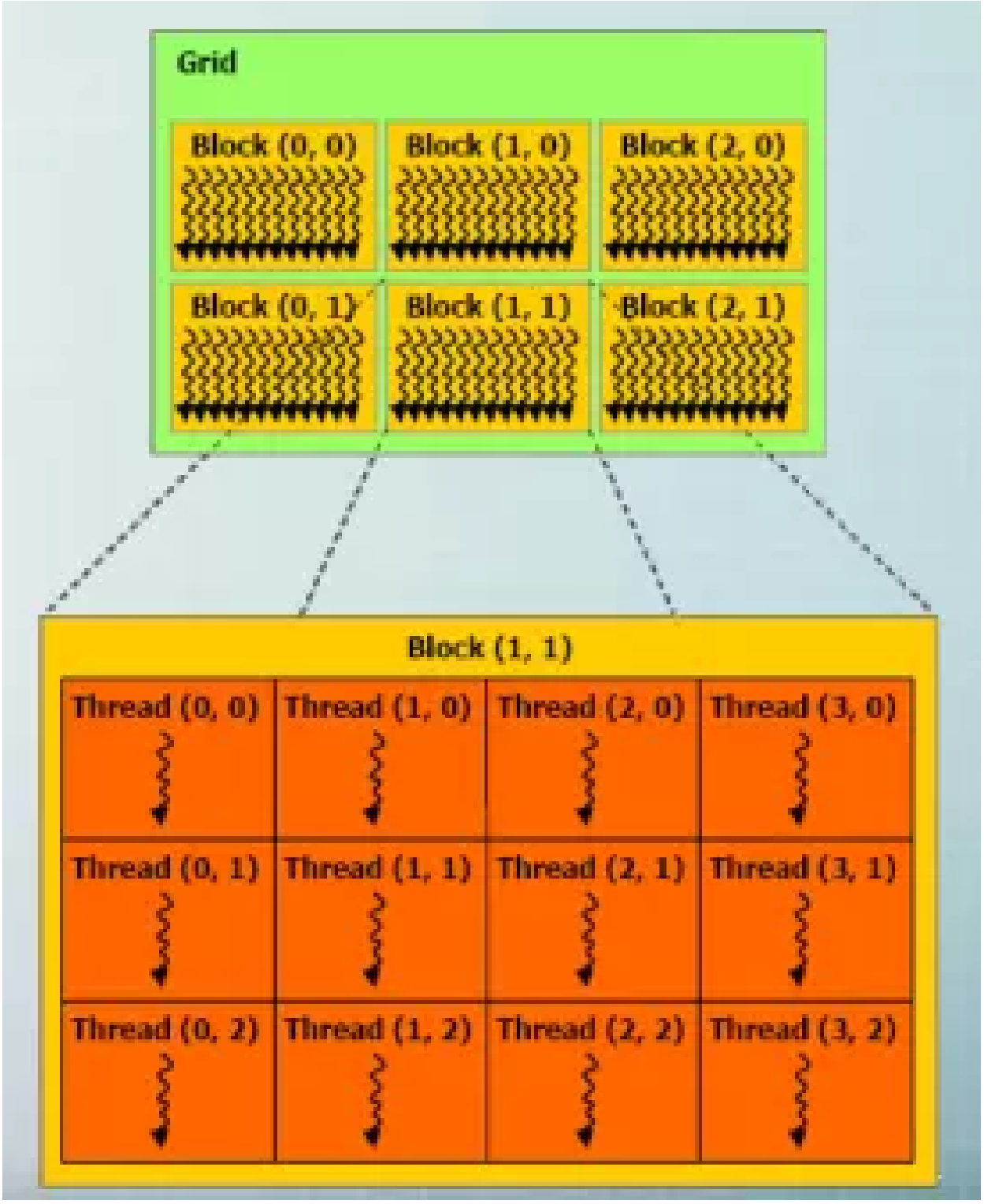
一般通过GPU的一个核进行处理。

- 线程块(Block)

1. 由多个线程组成（可以表示成一维，二维，三维，具体下面再细说）。
2. 各block是并行执行的，block间无法通信，也没有执行顺序。
3. 注意线程块的数量限制为不超过65535（硬件限制）。

- 线程格(Grid)

由多个线程块组成（可以表示成一维，二维，三维，具体下面再细说）。



- 线程束

在CUDA架构中，线程束是指一个包含32个线程的集合，这个线程集合被“编织在一起”并且“步调一致”的形式执行。在程序中的每一行，线程束中的每个线程都将在不同数据上执行相同的命令。

- 核函数（Kernel）

1. 在GPU上执行的函数通常称为核函数。

2. 一般通过标识符global修饰，调用通过<<<参数1,参数2>>>，用于说明内核函数中的线程数量，以及线程是如何组织的。
3. 以线程格（Grid）的形式组织，每个线程格由若干个线程块（block）组成，而每个线程块又由若干个线程（thread）组成。
4. 是以block为单位执行的。
5. 另能在主机端代码中调用。
6. 调用时必须声明内核函数的执行参数。
7. 在编程时，必须先为kernel函数中用到的数组或变量分配好足够的空间，再调用kernel函数，否则在GPU计算时会发生错误，例如越界或报错，甚至导致蓝屏和死机。
//核函数声明，前面的关键字global

```
__global__ voidkernel( void ){}
```

函数修饰符

1. __global__，表明被修饰的函数在设备上执行，但在主机上调用。

2. __device__，表明被修饰的函数在设备上执行，但只能在其他__device__函数或者__global__函数中调用。

常用的GPU内存函数

- cudaMalloc()

1. 函数原型： cudaError_t cudaMalloc (void **devPtr, size_t size)。
2. 函数用处：与C语言中的malloc函数一样，只是此函数在GPU的内存你分配内存。
3. 注意事项：
3.1. 可以将cudaMalloc()分配的指针传递给在设备上执行的函数；
3.2. 可以在设备代码中使用cudaMalloc()分配的指针进行设备内存读写操作；
3.3. 可以将cudaMalloc()分配的指针传递给在主机上执行的函数；
3.4. 不可以在主机代码中使用cudaMalloc()分配的指针进行主机内存读写操作（即不能进行解引用）。

- cudaMemcpy()

1. 函数原型：cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyKind kind)。
2. 函数作用：与c语言中的memcpy函数一样，只是此函数可以在主机内存和GPU内存之间互相拷贝数据。
3. 函数参数：cudaMemcpyKind kind表示数据拷贝方向，如果kind赋值为cudaMemcpyDeviceToHost表示数据从设备内存拷贝到主机内存。
4. 与C中的memcpy()一样，以同步方式执行，即当函数返回时，复制操作就已经完成了，并且在输出缓冲区中包含了复制进去的内容。
5. 相应的有个异步方式执行的函数cudaMemcpyAsync()，这个函数详解请看下面的流一节有关内容。

- cudaFree()

1. 函数原型：cudaError_t cudaFree (void* devPtr)。
2. 函数作用：与c语言中的free()函数一样，只是此函数释放的是cudaMalloc()分配的内存。

下面实例用于解释上面三个函数

GPU内存分类

- 全局内存

通俗意义上的设备内存。

- 共享内存

1. 位置：设备内存。
2. 形式：关键字__shared__添加到变量声明中。如__shared__ float cache[10]。
3. 目的：对于GPU上启动的每个线程块，CUDA C编译器都将创建该共享变量的一个副本。线程块中的每个线程都共享这块内存，但线程却无法看到也不能修改其他线程块的变量副本。这样使得一个线程块中的多个线程能够在计算上通信和协作。

- 常量内存

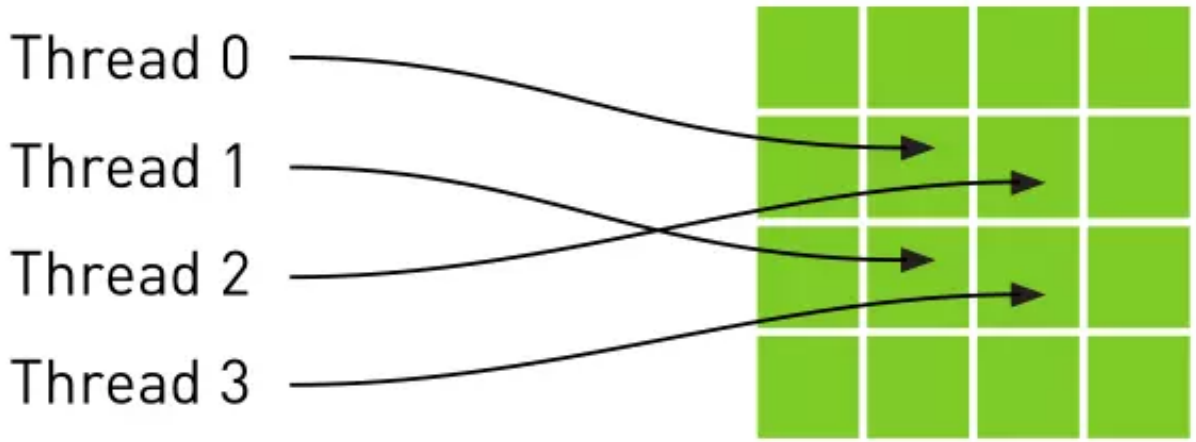
1. 位置：设备内存
2. 形式：关键字__constant__添加到变量声明中。如__constant__ float s[10];。

3. 目的：为了提升性能。常量内存采取了不同于标准全局内存的处理方式。在某些情况下，用常量内存替换全局内存能有效地减少内存带宽。
4. 特点：常量内存用于保存在核函数执行期间不会发生变化的数据。变量的访问限制为只读。NVIDIA硬件提供了64KB的常量内存。不再需要cudaMalloc()或者cudaFree(),而是在编译时，静态地分配空间。
5. 要求：当我们需要拷贝数据到常量内存中应该使用cudaMemcpyToSymbol()，而cudaMemcpy()会复制到全局内存。
6. 性能提升的原因：
6.1. 对常量内存的单次读操作可以广播到其他的“邻近”线程。这将节约15次读取操作。（为什么是15，因为“邻近”指半个线程束，一个线程束包含32个线程的集合。）
6.2. 常量内存的数据将缓存起来，因此对相同地址的连续读操作将不会产生额外的内存通信量。

- 纹理内存

1. 位置：设备内存

2. 目的：能够减少对内存的请求并提供高效的内存带宽。是专门为那些在内存访问模式中存在大量空间局部性的图形应用程序设计，意味着一个线程读取的位置可能与邻近线程读取的位置“非常接近”。如下图：



3. 纹理变量（引用）必须声明为文件作用域内的全局变量。

4. 形式：分为一维纹理内存 和 二维纹理内存。

4.1. 一维纹理内存

4.1.1. 用texture<类型>类型声明，如texture texIn。

4.1.2. 通过cudaBindTexture()绑定到纹理内存中。

4.1.3. 通过tex1Dfetch()来读取纹理内存中的数据。

4.1.4. 通过cudaUnbindTexture()取消绑定纹理内存。

4.2. 二维纹理内存

4.2.1. 用texture<类型,数字>类型声明，如texture texIn。

4.2.2. 通过cudaBindTexture2D()绑定到纹理内存中。

4.2.3. 通过tex2D()来读取纹理内存中的数据。

4.2.4. 通过cudaUnbindTexture()取消绑定纹理内存。

- 固定内存

1. 位置：主机内存。
2. 概念：也称为页锁定内存或者不可分页内存，操作系统将不会对这块内存分页并交换到磁盘上，从而确保了该内存始终驻留在物理内存中。因此操作系统能够安全地使某个应用程序访问该内存的物理地址，因为这块内存将不会破坏或者重新定位。
3. 目的：提高访问速度。由于GPU知道主机内存的物理地址，因此可以通过“直接内存访问DMA（Direct Memory Access)技术来在GPU和主机之间复制数据。由于DMA在执行复制时无需CPU介入。因此DMA复制过程中使用固定内存是非常重要的。

4. 缺点：使用固定内存，将失去虚拟内存的所有功能；系统将更快的耗尽内存。
5. 建议：对cudaMemcpy()函数调用中的源内存或者目标内存，才使用固定内存，并且在不再需要使用它们时立即释放。
6. 形式：通过cudaHostAlloc()函数来分配；通过cudaFreeHost()释放。
7. 只能以异步方式对固定内存进行复制操作。

- 原子性

1. 概念：如果操作的执行过程不能分解为更小的部分，我们将满足这种条件限制的操作称为原子操作。
2. 形式：函数调用，如atomicAdd（addr,y)将生成一个原子的操作序列，这个操作序列包括读取地址addr处的值，将y增加到这个值，以及将结果保存回地址addr。

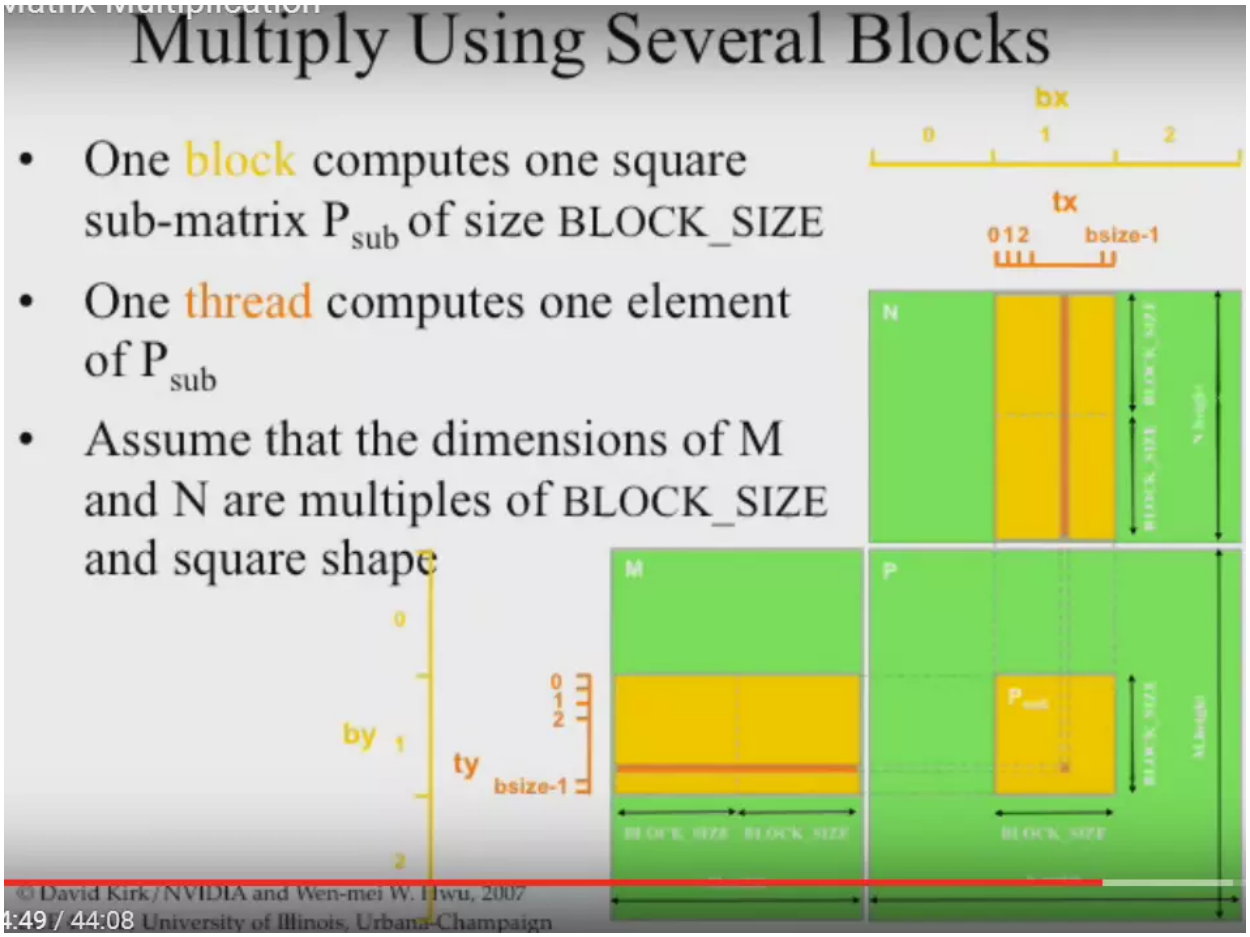
常用线程操作函数

1. 同步方法__syncthreads()，这个函数的调用，将确保线程块中的每个线程都执行完__syscthreads()前面的语句后，才会执行下一条语句。

cuda C 做矩阵乘法（Tiled 算法）

为什么看cuda C 做矩阵乘法运算呢？在深度神经网络中，全连接层、卷积层、池化层，几乎我们可以想到的所有操作都离不开矩阵运算，卷积层最后其实也是转化为矩阵的乘法操作进行优化，在【conv2d 实现 caffe&tensorflow】中有介绍原理。

参考视频地址：https://www.youtube.com/watch?v=SqZaletdPCY



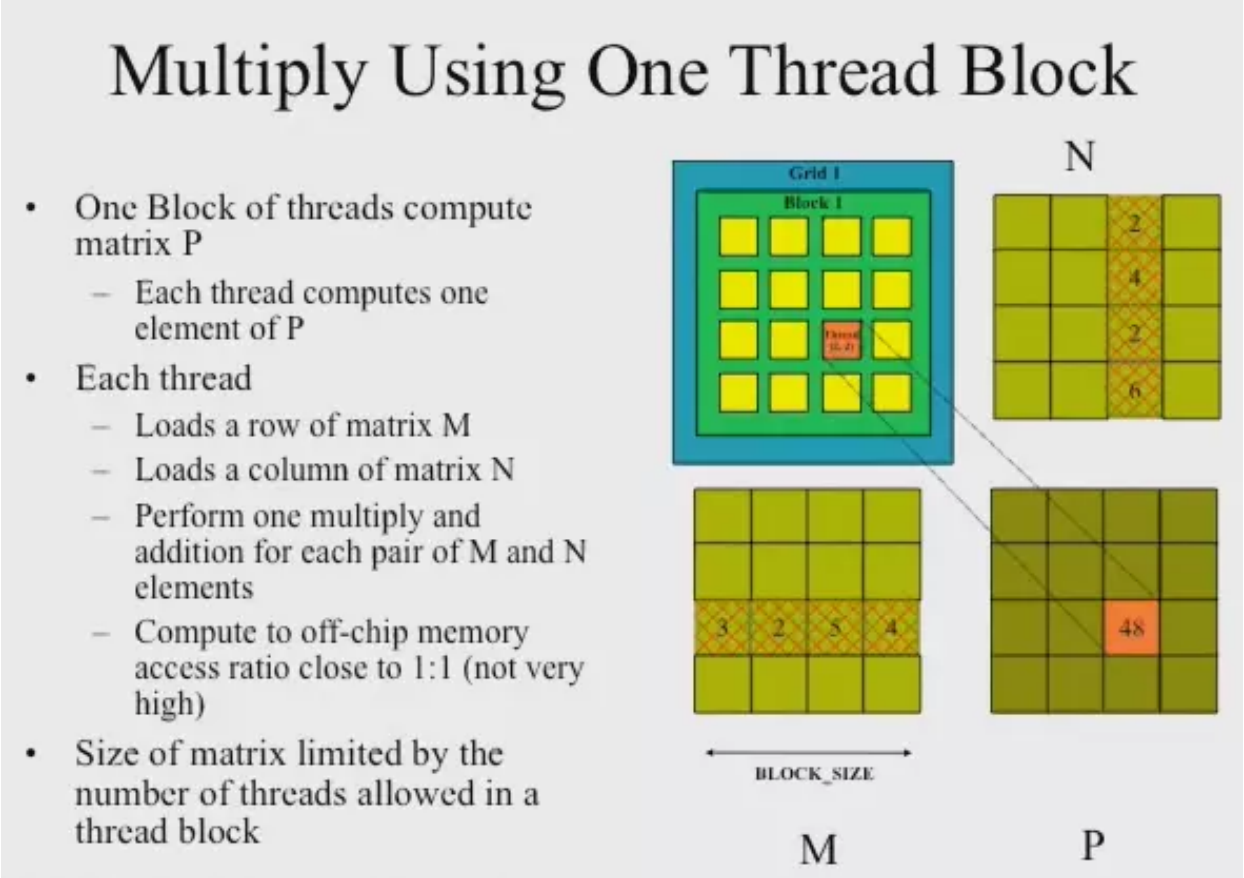
思想： 为了引入共享内存的概念降低GPU带宽使用，把要计算的两个矩阵A B 先分解成BLOCK_SIZE=16大小的submatrix，每一个block结构运算一个submatrix乘法，这样在一个block中所有的线程是共享参数的，不用每次计算都从global memory中重新加载。

```
template <int BLOCK_SIZE> __global__ voidmatrixMulCUDA(float *C, float *A, float *B, int wA, int wB){ // Thread 所在 block 的 location int bx = blockIdx.x; int by = blockIdx.y; // Thread 的 location int tx = threadIdx.x; int ty = threadIdx.y; // A矩阵16 * 16 子矩阵的起始下标int aBegin = wA * BLOCK_SIZE * by; // A矩阵16 * 16 子矩阵的终止下标（就是A矩阵一次运算一行，对应着B 矩阵一次运算一列）int aEnd = aBegin + wA - 1; // A矩阵下标一次移动的步长， 子矩阵是16 * 16，一次处理一个子矩阵，那么步长显然就是16了int aStep = BLOCK_SIZE; // B 矩阵子矩阵对应的起始下标int bBegin = BLOCK_SIZE * bx; // B 矩阵子矩阵对应的步长，一次移动16*widthB，同样也是隔出16*16的子矩阵出来int bStep = BLOCK_SIZE * wB; // 累加，得到行 * 列的值float Csub = 0; // 循环次数等于widthA / 16，把长向量点积运算转化为两个短向量点积后的和for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) { // 定义A的共享子矩阵变量,因为__shared__ 声明，所以同一个block中的所有threads都可见，//每个thread填充一个元素，并计算一个行列乘积，减小带宽使用 __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
```

```
// 定义A的共享子矩阵变量 __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
// 每个block包含16 * 16 个线程，所以每个线程负责一个矩阵元素的拷贝（注意同步）
As[ty][tx] = A[a + wA * ty + tx]; Bs[ty][tx] = B[b + wB * ty + tx];
// Synchronize to make sure the matrices are loaded
__syncthreads(); // 每个线程计算 子矩阵的行列乘积，大循环外边还有累加，累加的是不同子矩阵点积和for (int k = 0; k < BLOCK_SIZE; ++k) { Csub += As[ty][k] * Bs[k][tx]; } // 再次同步 __syncthreads(); } // 把结果赋值到C矩阵，计算结果对应C下边的过程int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx; C[c + wB * ty + tx] = Csub;}
```

只看源代码很难理解矩阵加速真正的原理，这是一个坑，还有是输入矩阵的尺寸大小，只能是BLOCK_SIZE=16的整数倍，不然会出错（实验结果也表明确实出错了，又是一个坑）。

为什么采用Tiled 算法呢？主要是不这么做GPU从global memory读取数据的代价太大了。



How about performance?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code should run at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS