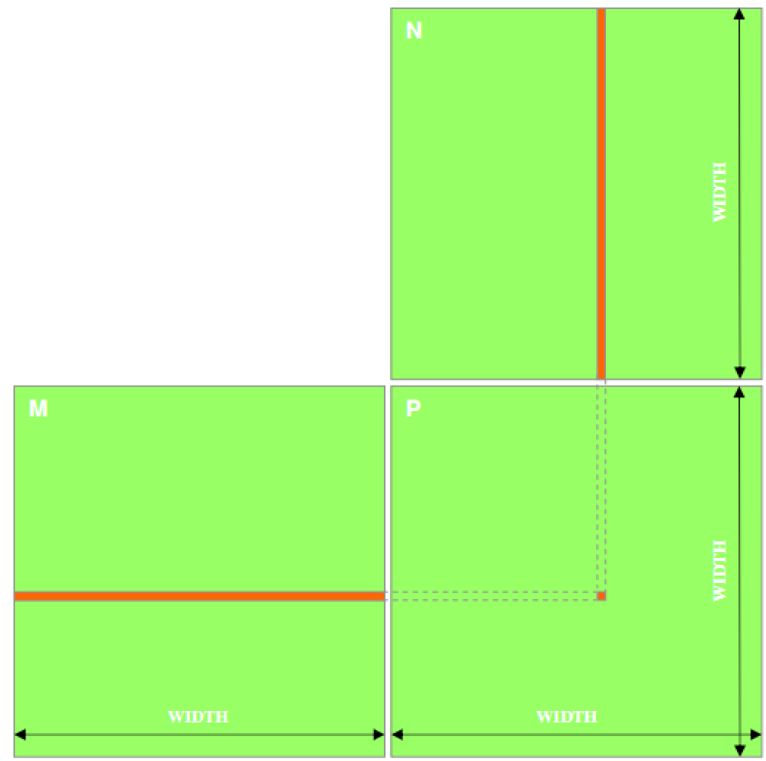
# CUDA编程(2)

周　斌 @NVIDIA & USTC 2015年7月

# 我们来重新分析 matrix multiply

# Matrix Multiply: CPU 实现

```
void MatrixMulOnHost(float* M, float* N, float* P, int width)
{
  for (int i = 0; i < width; ++i)
    for (int j = 0; j < width; ++j)
    {
      float sum = 0;
      for (int k = 0; k < width; ++k
      {
        float a = M[i * width + k];
        float b = N[k * width + j];
        sum += a * b;
      }
      P[i * width + j] = sum;
    }
}
```






Code from: http://courses.engr.illinois.edu/ece498/al/lectures/lecture3%20cuda%20threads%20spring%202010.ppt

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Md.width + k];
     float Ndelement = Nd[k * Nd.width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```

访问矩阵，所以用二维block



Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

   for (int k = 0; k < Width; ++k)
   {
      float Mdelement = Md[ty * Md.width + k];
      float Ndelement = Nd[k * Nd.width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

每个kernel计算一个输出结果



Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

CPU 版本的2个外层循环去哪儿了 ？



Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

     for (int k = 0; k < Width; ++k)
     {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
      }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

不用锁或同步，为什么？



Code from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

- 问题
  - 矩阵长度限制
    - 仅用一个block
    - G80 和 GT200 – 最多512 个线程/block
  - 很多 global memory 读写访问

- 去除长度限制
  - 将Pd 矩阵拆成tile小块
  - 把一个tile 布置到一个block
  - 通过 threadIdx 和 blockIdx 索引

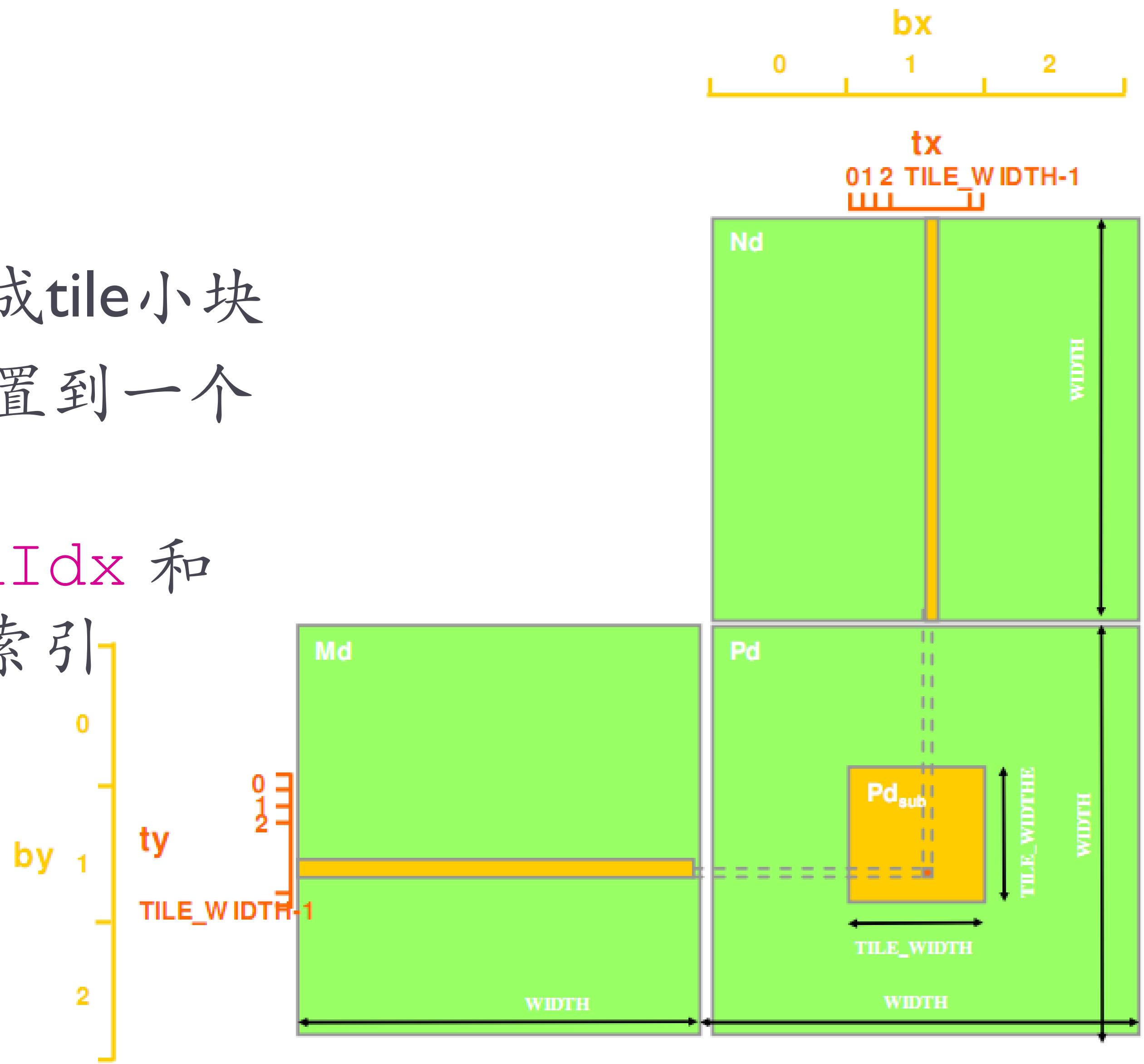





Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

- 例如
  - 矩阵: 4x4
  - `TILE_WIDTH` = 2
  - Block 尺寸: 2x2

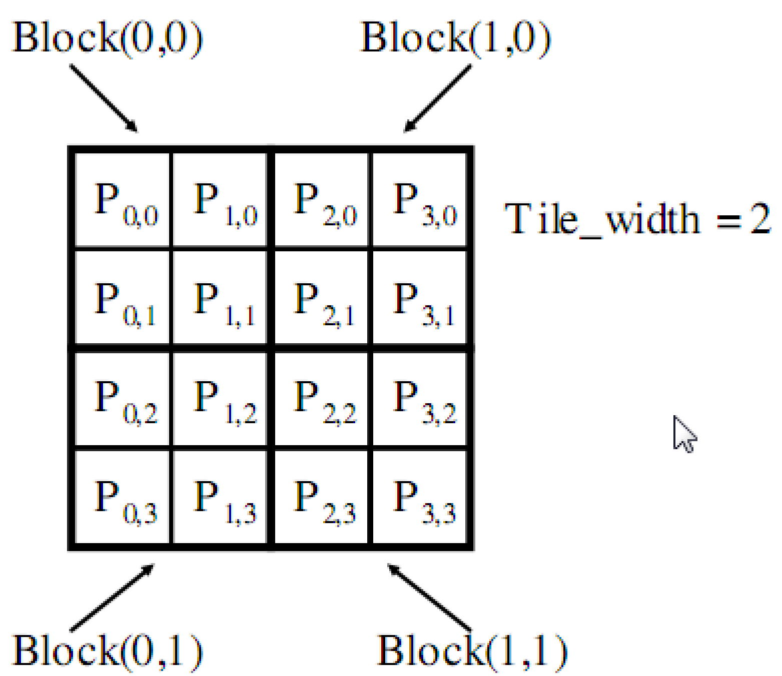





Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

- 例如
- 矩阵: 4x4
- TILE_WIDTH = 2
- Block 尺寸: 2x2

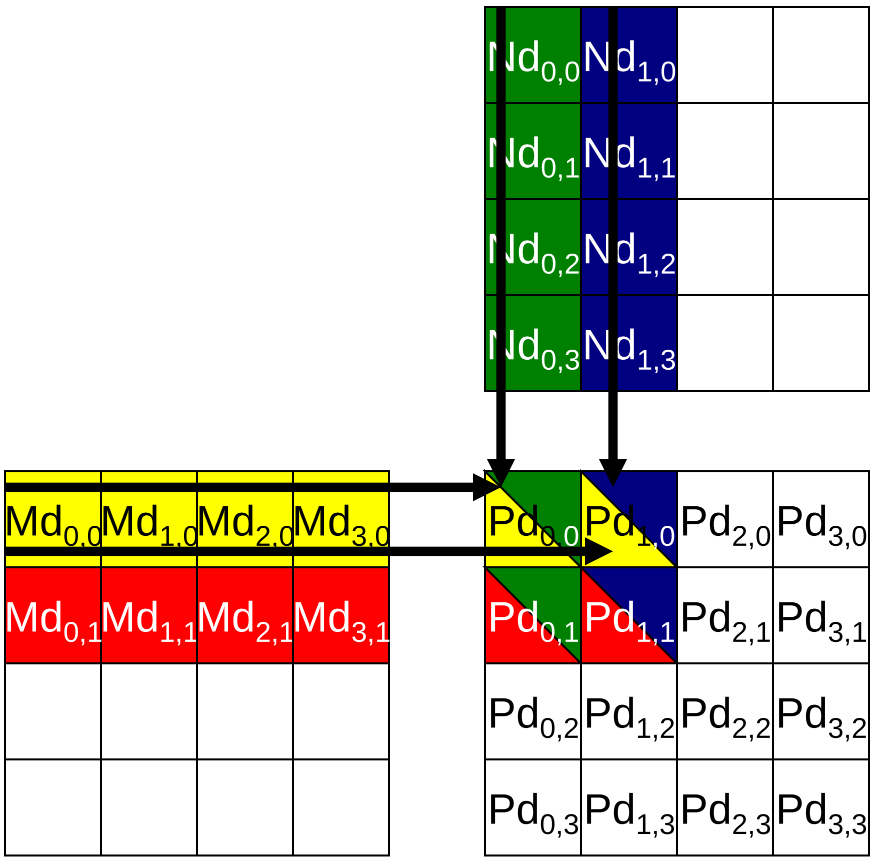





Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  int Col = blockIdx.x * blockDim.x + threadIdx.x;

  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row * Width + k] * Nd[k * Width +
   Col];

  Pd[Row * Width + Col] = Pvalue;
}
```



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

计算矩阵Pd 和M的行索引

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  int Row = blockIdx.x * blockDim.x + threadIdx.x;
  int Col = blockIdx.y * blockDim.y + threadIdx.y;

  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
    Pvalue += Md[Col * Width + k] * Nd[k * Width + Row];

  Pd[Row * Width + Col] = Pvalue;
}
```



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

计算矩阵Pd 和N的列索引

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  int Col = blockIdx.x * blockDim.x + threadIdx.x;

  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

  Pd[Row * Width + Col] = Pvalue;
}
```



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

每个线程计算块内子矩阵的一个元素

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  int Col = blockIdx.x * blockDim.x + threadIdx.x;

  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

  Pd[Row * Width + Col] = Pvalue;
}
```



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

- 调用 kernel:

```
dim3 dimGrid(Width / TILE_WIDTH, Height / TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

MatrixMulKernel<<<dimGrid, dimBlock>>>(
  Md, Nd, Pd, TILE_WIDTH);
```

# global memory 读写怎么办?

- 受限于global memory 带宽
  - G80 峰值GFLOPS: 346.5
  - 需要 1386 GB/s 的带宽来达到
  - G80 存储器实际带宽: 86.4 GB/s
    - 限制代码 21.6 GFLOPS
    - 实际上，代码运行速度是 15 GFLOPS
  - 必须大幅减少对 global memory 的访问

- 每个输入元素被Width 个线程读取
- 使用 shared memory 来减少 global memory 带宽需求

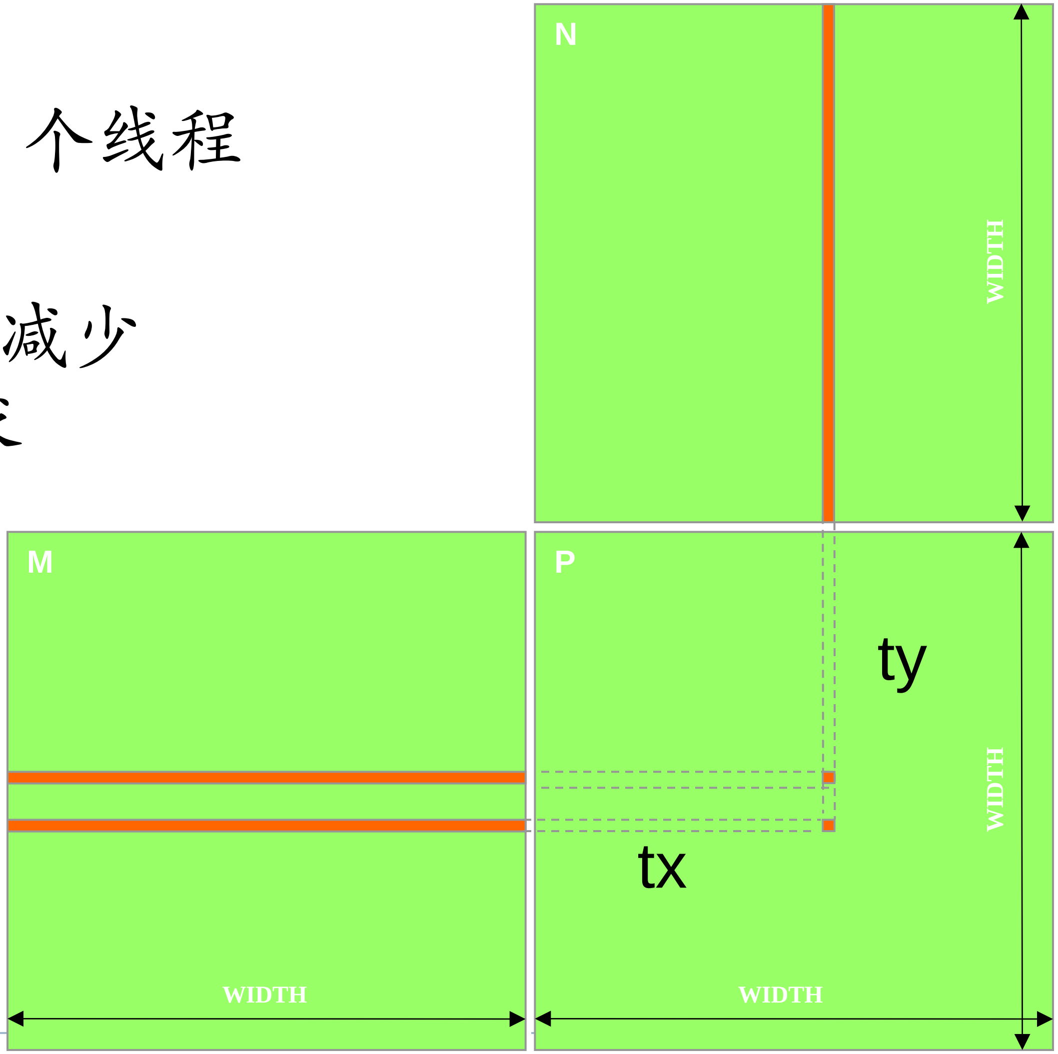



Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

- 把 kernel 拆分成多个阶段
  - 每个阶段用 Md 和 Nd 的子集累加Pd
  - 每个阶段有很好的数据局部性

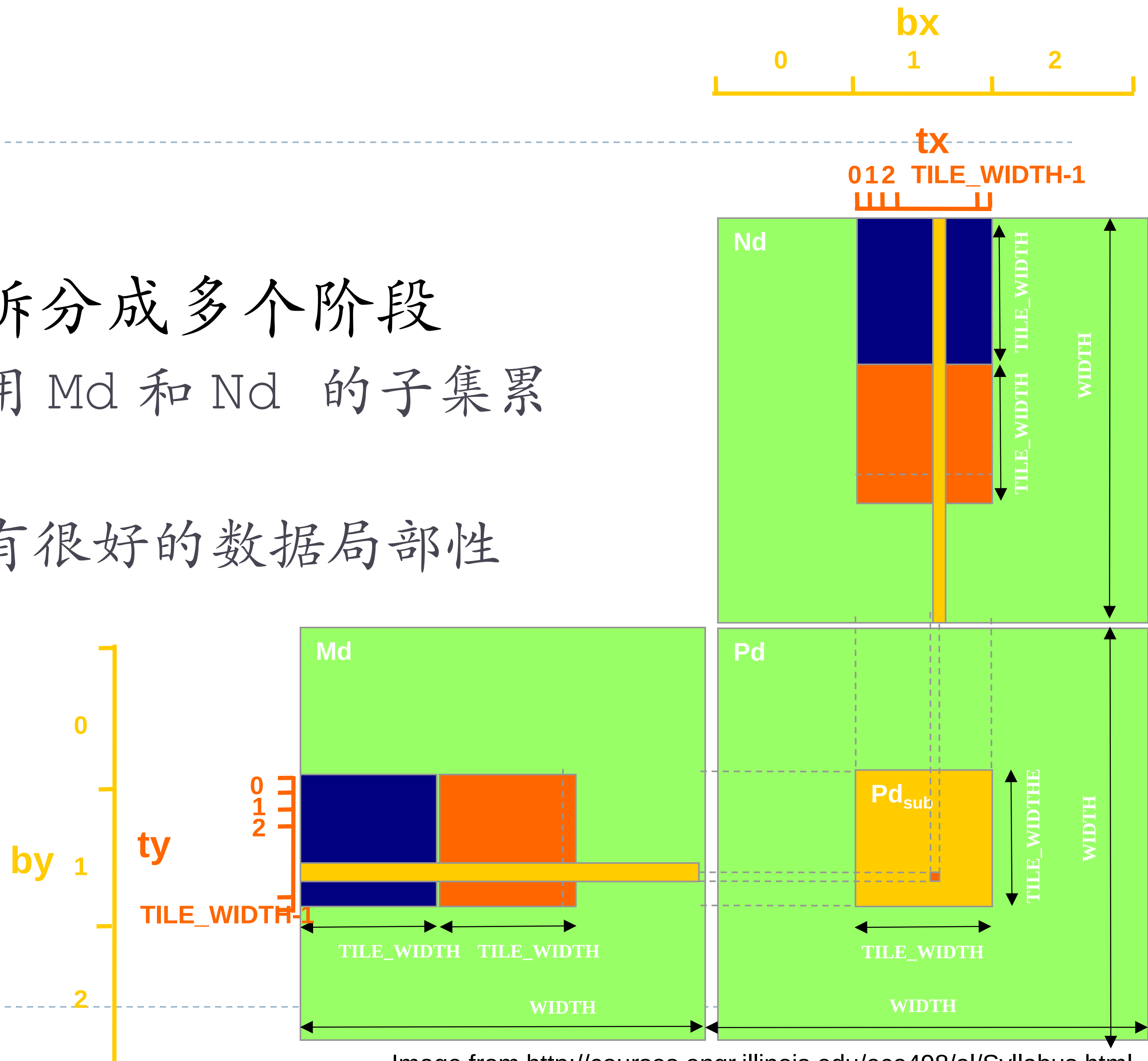
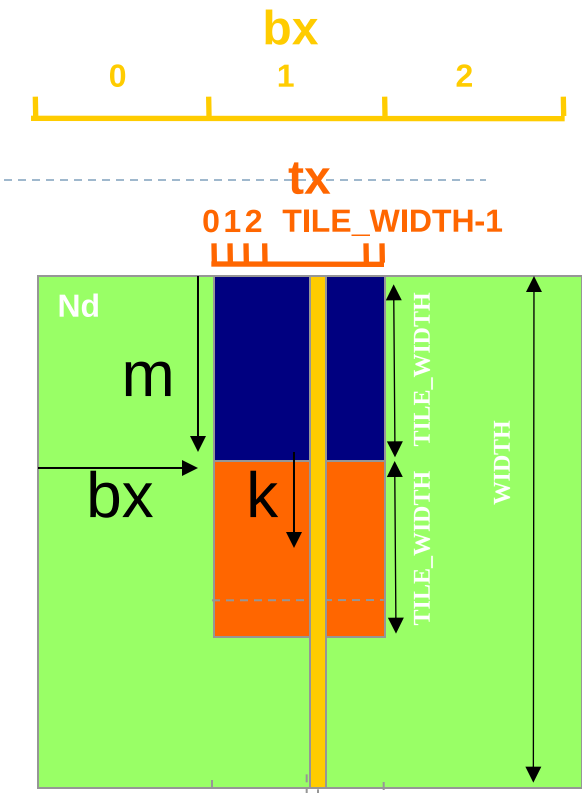


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

- 每个线程
  - 读入瓦片内 Md 和 Nd 的一个元素存入 shared memory



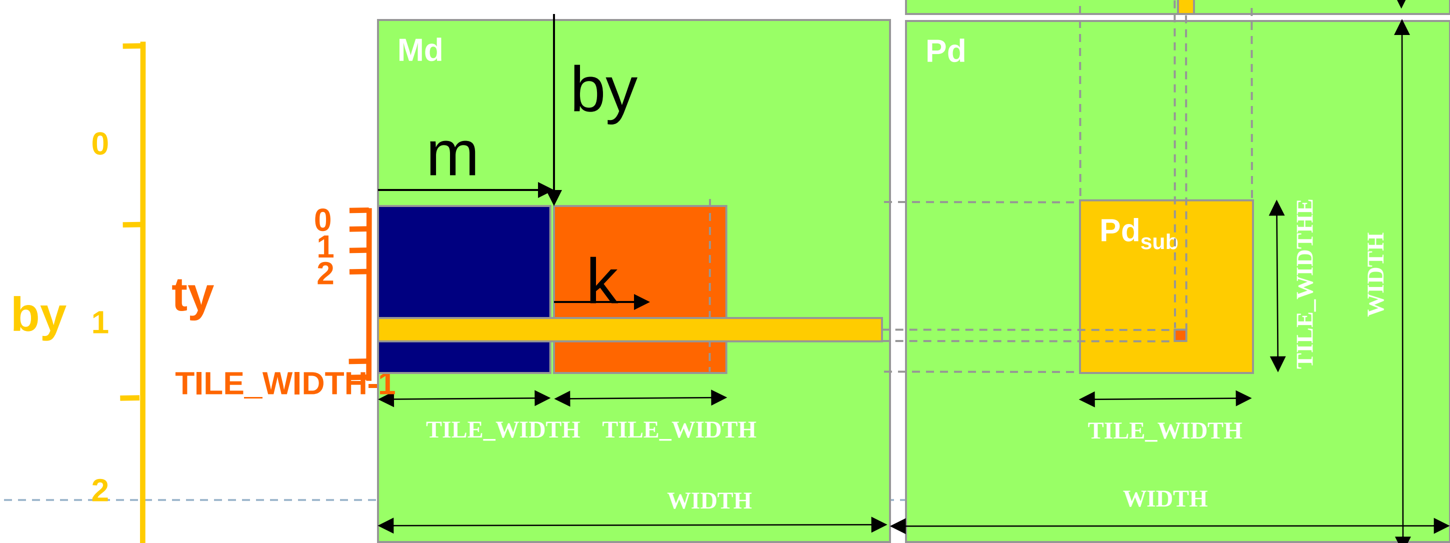




Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __synchthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

Shared memory 存储 Md 和 Nd 的子集



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m)  {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

Width/TILE_WIDTH
- 阶段数目

m
- 当前阶段的索引



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

从Md 和 Nd 各取一个元素存入 shared memory



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __synchthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

等待block内所有线程, 即, 等到整个瓦片存入 shared memory



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

累加点乘的子集



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

为什么？



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMulKernel(
  float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __synchthreads();
  }
  Pd[Row*Width+Col] = Pvalue;
}
```

把最终结果写入
global memory



Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

# Matrix Multiply

- 如何选取 TILE_WIDTH的数值?
  - 如果太大的话会怎样?

- 如何选取 TILE_WIDTH的数值?
  - 如果太大的话会怎样?
    - 超出一个块允许的最大线程数
      - G80 and GT200 – 512
      - Fermi – 1024
      - Kerpler – 1024
      - 可疑的，依据不同的计算能力，请查表

- 如何选取 TILE_WIDTH的数值?
  - 如果太大的话会怎样?
    - 超出一个块允许的最大线程数
      - → G80 and GT200 – 512
      - → Fermi – 1024
    - 超出shared memory 极限
      - → G80: 16KB / SM 并且 8 blocks / SM
        - → 2 KB / block
        - → 1 KB 给 Nds , 1 KB 给 Mds (16 * 16 * 4)
        - → TILE_WIDTH = 16
        - → 更大的 TILE_WIDTH 将导致更少的块数

- Shared memory 瓦片化的好处
  - global memory 访问次数减少TILE_WIDTH倍
    - 16x16 瓦片 减少16倍
  - G80
    - 现在 global memory 支持 345.6 GFLOPS
    - 接近峰值 346.5 GFLOPS

# G80 线程尺寸的考虑

- 每个 thread block 有许多个线程
  - TILE_WIDTH 为 16 时： 16*16 = 256 个线程
- 需要许多个 thread blocks
  - 一个 1024*1024 Pd 需要： 64*64 = 4K Thread Blocks
- 每个 thread block 执行 2*256 = 512 次global memory 的float 读入， 为了供应 256 * (2*16) = 8K mul/add 操作.
  - 存储带宽不再是限制因素



Slide from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

# Atomic Functions原子函数

- 许多原子操作:

```
// 算术运算
atomicAdd()
atomicSub()
atomicExch()
atomicMin()
atomicMax()
atomicAdd()
atomicDec()
atomicCAS()
```

```
// 位运算
atomicAnd()
atomicOr()
atomicXor()
```



See B.10 in the NVIDIA CUDA C Programming Guide

# Atomic Functions原子函数

- 不同块里面的线程如何协作？
- 尽量少用原子操作，为什么？