VSCSE Summer School

Proven Algorithmic Techniques for Many-core Processors

Lecture 3: Blocking/Tiling for Locality

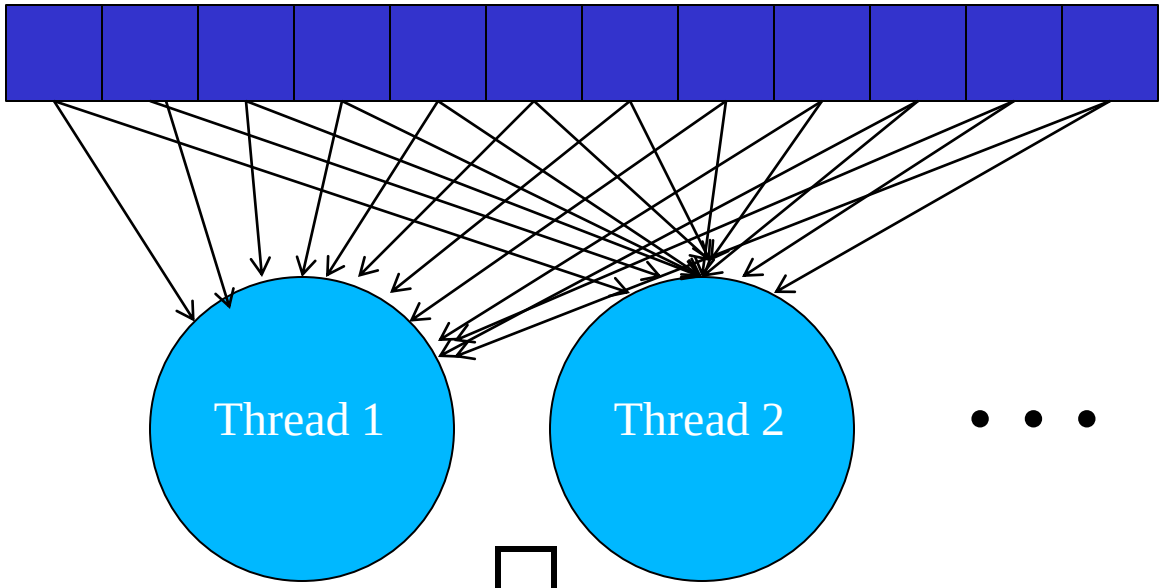
Objective

- Reuse each data accessed from the global memory multiple times
  - Across threads – shared memory blocking
  - With a thread - register tiling
- Register tiling is also often used to re-use computation results for increased efficiency.

Basic Idea

Global Memory

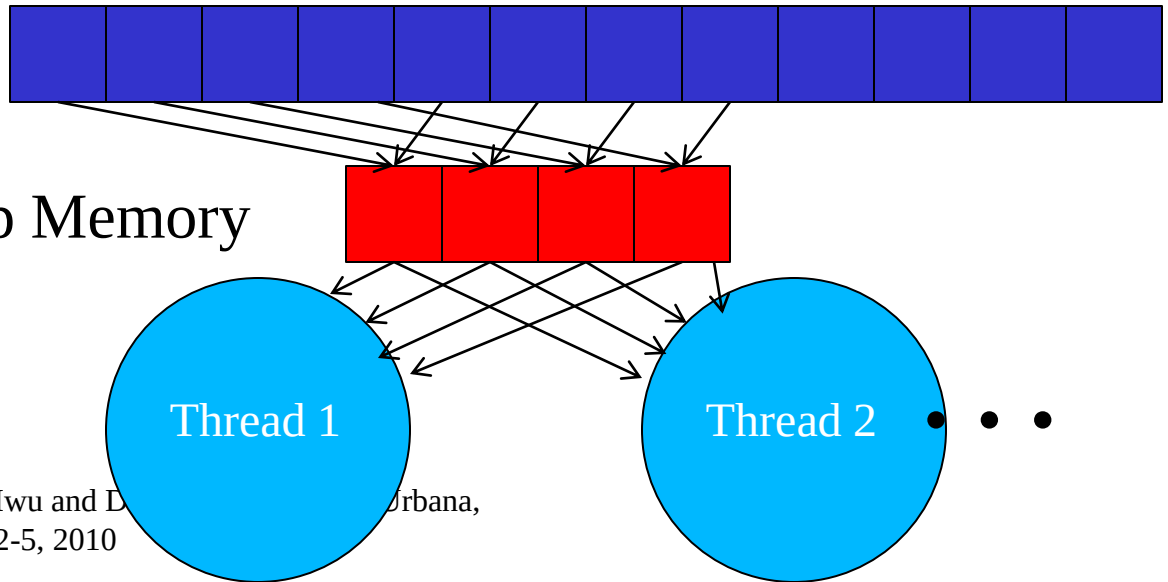
in



Global Memory

in

On-chip Memory



Basic Concept of Blocking/Tiling

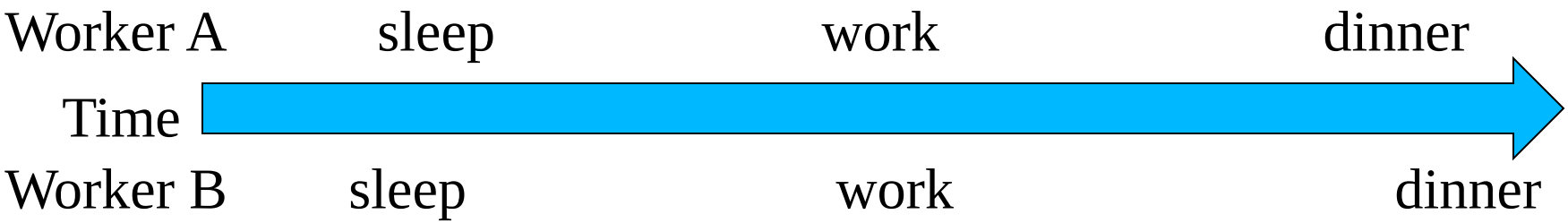
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commutes
  - Blocking/Tiling for global memory accesses

Some computations are more challenging to block/tile than others.

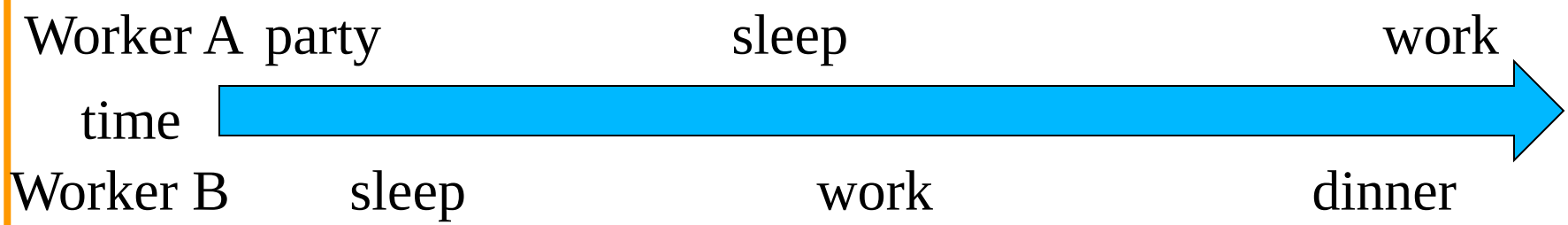
- Some carpools may be easier than others
  - More efficient if neighbors are also classmates or co-workers
  - Some vehicles may be more suitable for carpooling
- Similar variations exist in blocking/tiling

Carpools need synchronization.

- Good – when people have similar schedule

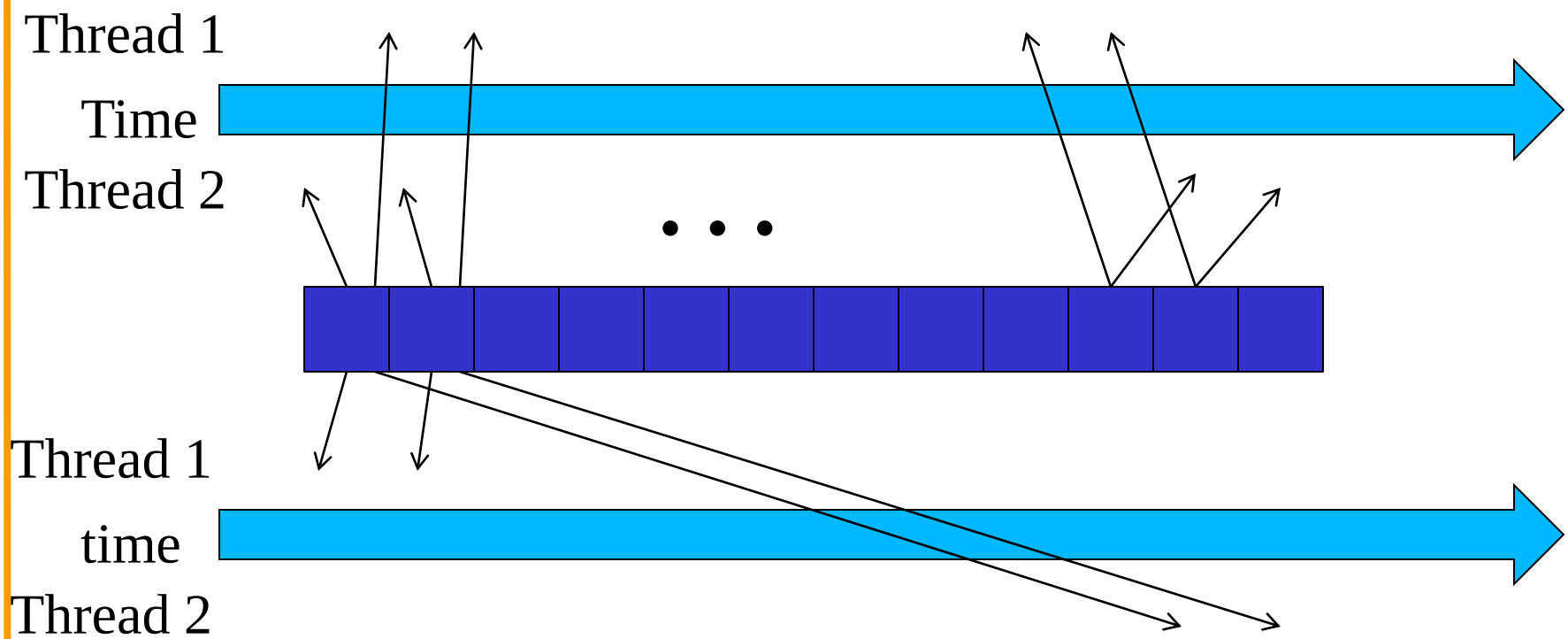


- Bad – when people have very different schedule



Same with Blocking/Tiling

- Good – when threads have similar access timing



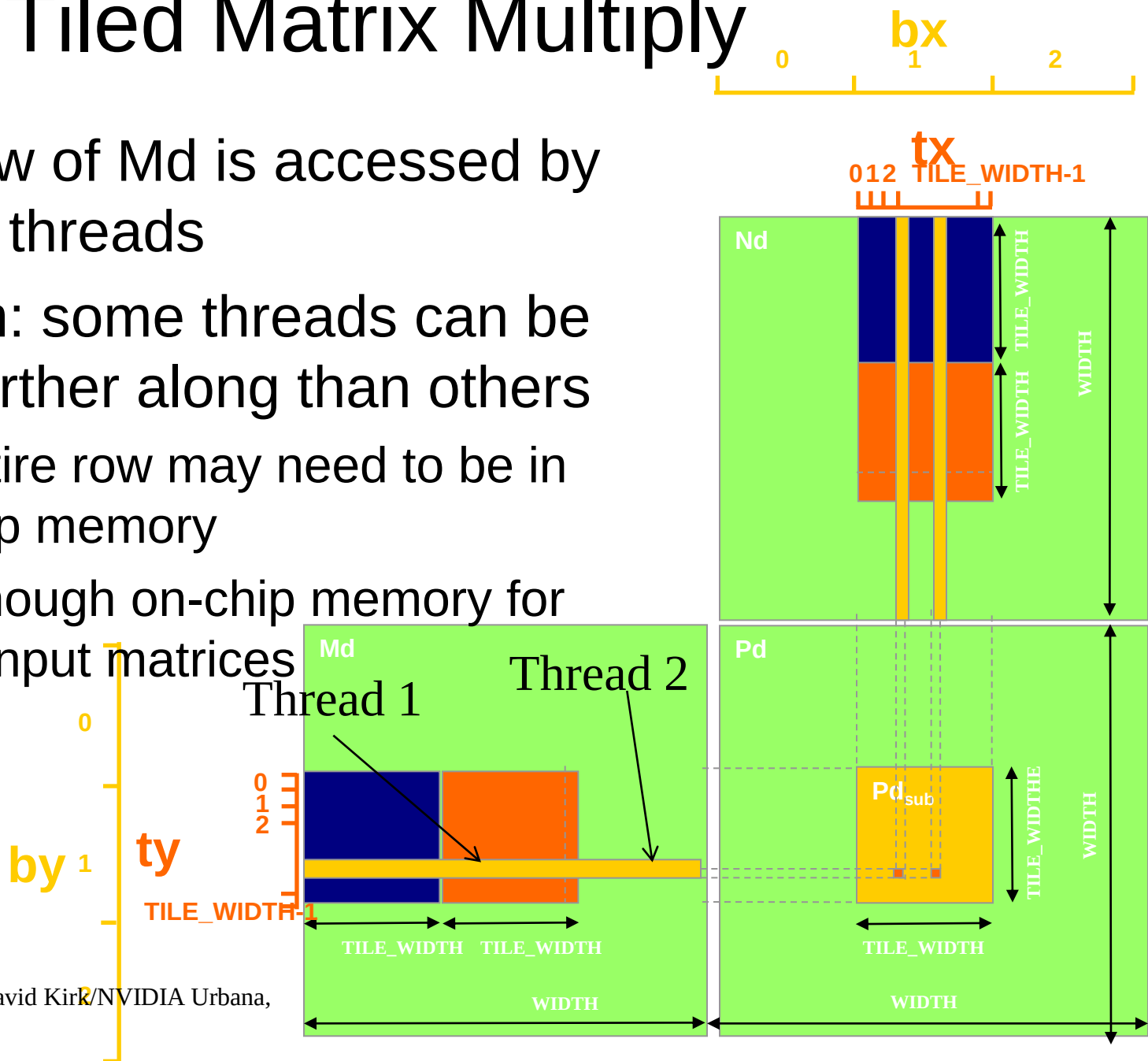
- Bad – when threads have very different timing

Outline of Technique

- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to get their data from the on-chip memory
- Move on to the next block/tile

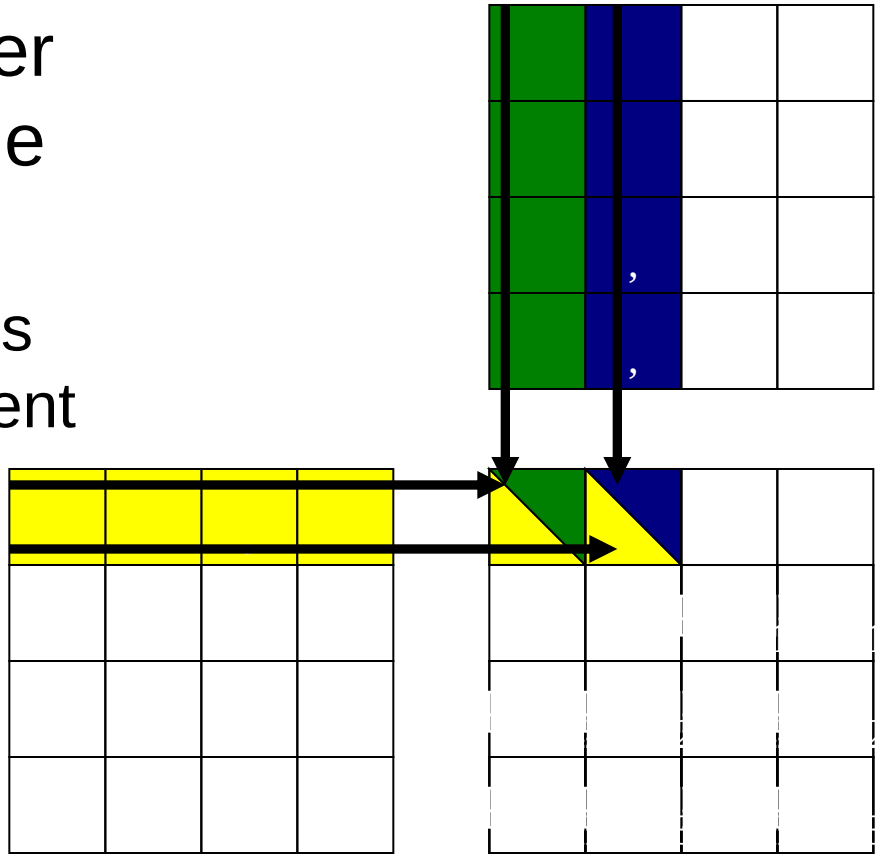
Tiled Matrix Multiply

- Each row of Md is accessed by multiple threads
- Problem: some threads can be much further along than others
  - An entire row may need to be in on-chip memory
  - Not enough on-chip memory for large input matrices

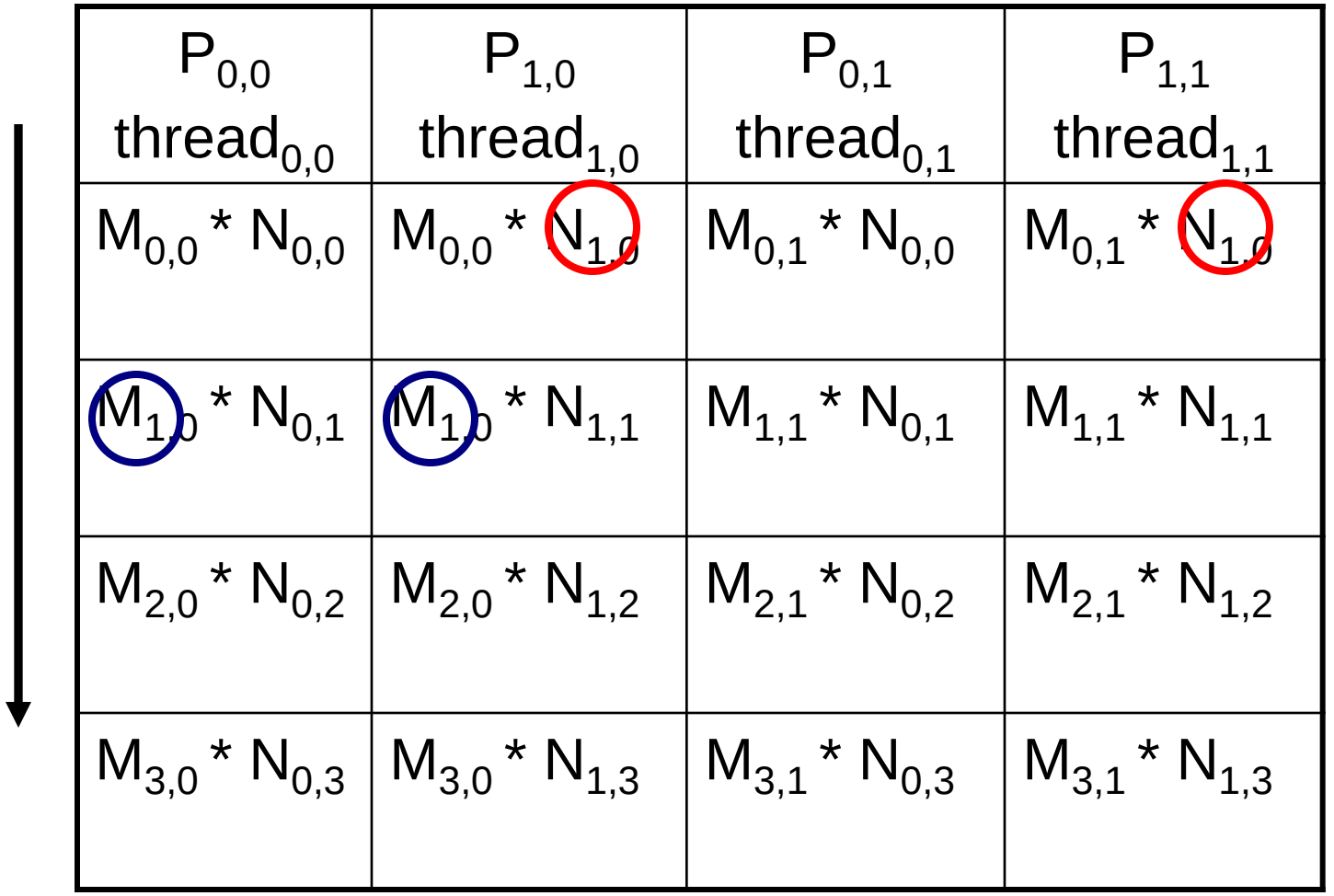


A Small Example

- Can we use two on-chip memory locations to reduce the number of M accesses by the two threads?
  - Not if the two threads can have very different timing!

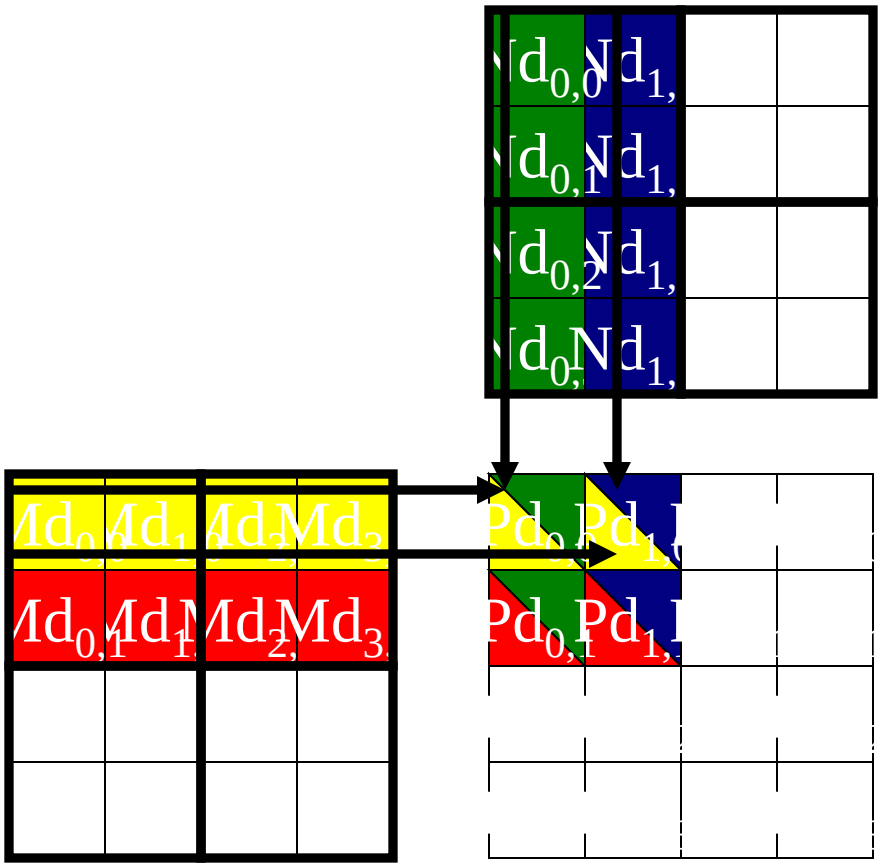


Every M and N Element is used exactly twice in generating a 2X2 tile of P



$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Breaking Md and Nd into Tiles



Each phase uses one tile from Md and one from Nd

$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

First-order Size Considerations

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor

CUDA Code – Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

Tiled Matrix Multiplication Kernel

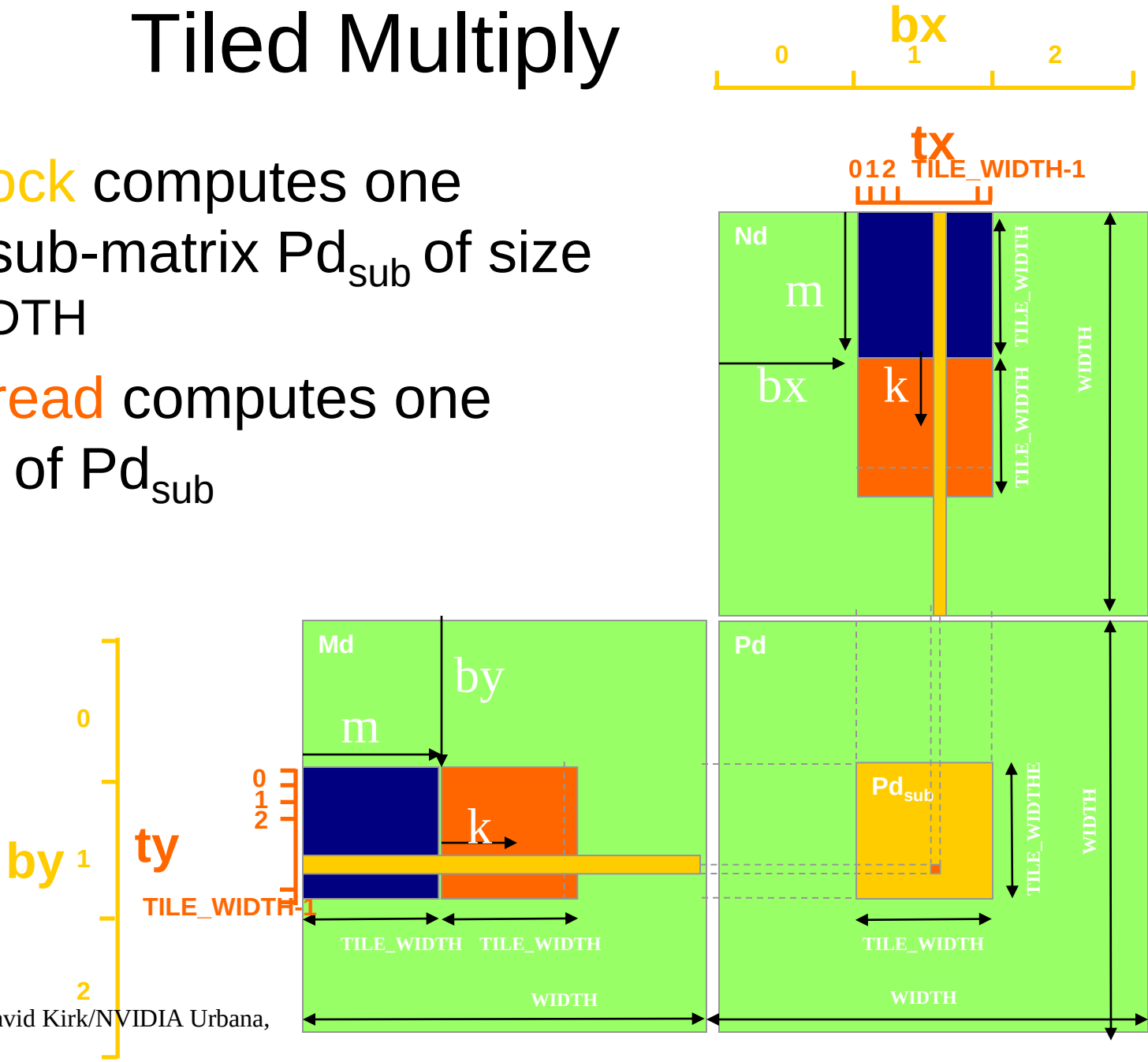
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
9.     Mds[tx][ty] = Md[(m*TILE_WIDTH + tx)*Width+Row];
10.    Nds[tx][ty] = Nd[Col*Width+(m*TILE_WIDTH + ty)];
11.    __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.      Pvalue += Mds[tx][k] * Nds[k][ty];
14.      synchthreads();
15.  }
16.  Pd[Row*Width+Col] = Pvalue;
```

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



Shared Memory and Threading

- Each SM in Fermi has 64KB on-chip SRAM, partitioned into 48KB L1 cache and 16KB shared memory, or vice versa
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing 2 or 6 thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - A 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!

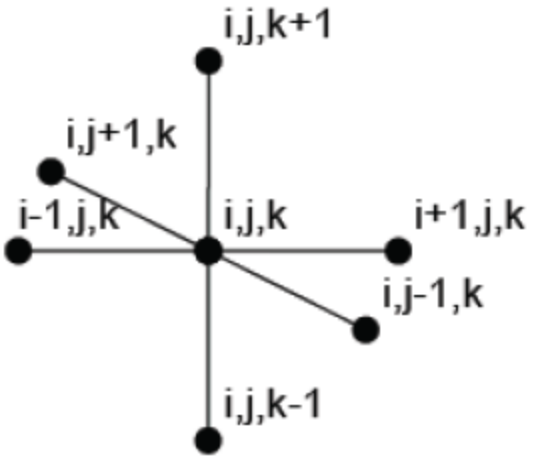
More Difficult Blocking/Tiling Cases

- Some applications do not access input data in a uniform way
  - Convolution
  - PDE solvers

STENCIL CODE EXAMPLE

Stencil Computation

- Describes the class of nearest neighbour computations on structured grids.
- Each point in the grid is a weighted linear combination of a subset of neighbouring values.



Overview

- Application: 7-pt Stencil Computation/ Stencil Probe.
- Optimizations and concepts covered : Improving locality and Data Reuse
  - 2D Tiling in Shared Memory
  - Register Tiling

Stencil Computation

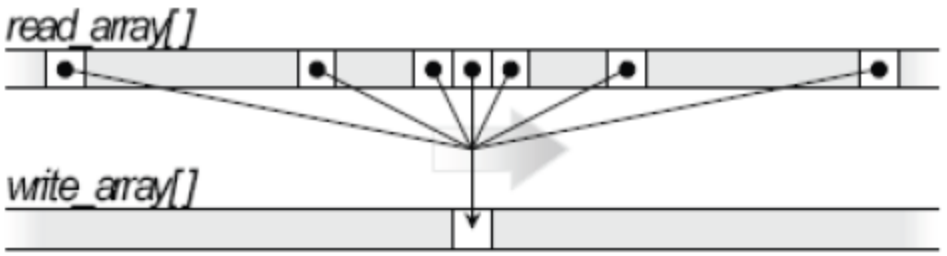
- High parallelism: Conceptually, all points in the grid can be updated in parallel.
- Each computation performs a global sweep through the data structure.
- Low computational intensity: High memory traffic for very few computations.
- Challenge: Exploit parallelism without overusing memory bandwidth

Implementation Details

- General Equation:

$$\begin{aligned} B[i,j,k] &= C_0 A[i,j,k] + C_1(\\ &+ A[i-1,j,k] + A[i,j-1,k] + A[i,j,k-1] \\ &+ A[i+1,j,k] + A[i,j+1,k] + A[i,j,k+1]) \end{aligned}$$

- Separate read and write arrays.
- Mapping of arrays from 3D space to linear array space.



Naïve implementation

- Each thread calculates a one-element thin column along the z-dimension
  - Each block computes a rectangular column along the z-dimension
- Each thread loads its input elements from global memory, independently of other threads
  - High read redundancy, heavy global memory traffic
- Optimization – each thread can reuse data along the z-dimension
  - The current center input becomes the bottom input
  - The current top input becomes the center input

Sample Naïve Kernel Code

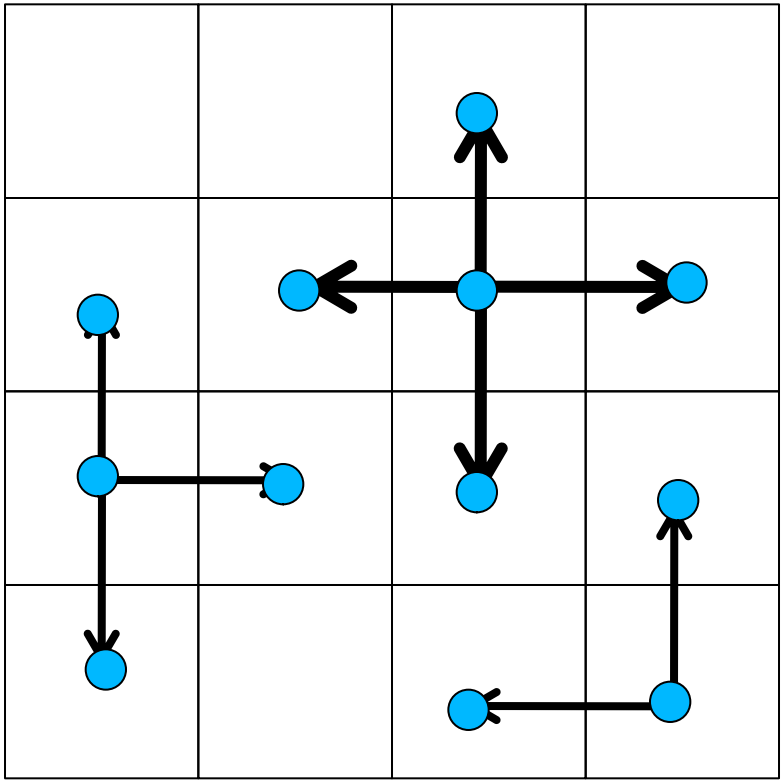
```
float bottom = A0[Index3D (nx, ny, i, j, 0)];
float current = A0[Index3D (nx, ny, i, j, 1)];
float top = A0[Index3D (nx, ny, i, j, 2)];
for (int k = 1; k < nz-1; k++) {
    Anext[Index3D (nx, ny, i, j, k)] =
        bottom +
        top +
        A0[Index3D (nx, ny, i, j + 1, k)] +
        A0[Index3D (nx, ny, i, j - 1, k)] +
        A0[Index3D (nx, ny, i + 1, j, k)] +
        A0[Index3D (nx, ny, i - 1, j, k)]
        - 6.0f * current / (fac*fac);
    bottom = current;
    current = top;
    top = A0[Index3D(nx, ny, i, j, k+1)];
}
```

Memory Loads in the Naïve Kernel

- Assume no data reuse along the z-direction within each thread,
  - A thread loads 7 input elements for each output element.
- With data reuse within each thread,
  - A thread loads 5 input elements for each output

Data Reuse

- Each internal point is used to calculate seven output values
  - self, 4 planar neighbors, top and bottom neighbors
- Surface, edge, and corner points are used for fewer output values

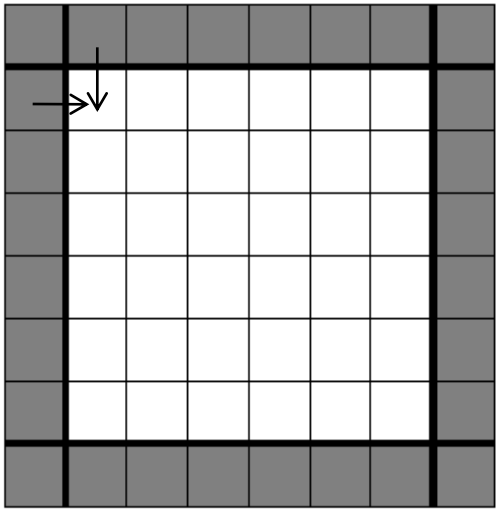


Improving Locality: 2D Tiling

- Assume that all threads of a block march up the z-direction in synchronized phases
- In each phase, all threads calculate a 2-D slide of the rectangular output column
- For each phase, maintain three slices of relevant input data in the on-chip memories
  - One top and one bottom element in each thread's private registers
  - All current elements also in shared memory

Improving Locality: 2D Tiling (cont.)

- From one phase to next, the kernel code
  - Moves current element to register for lower element
  - Moves top element from top register to current register and shared memory
  - Load new top element from Global Memory to register
- Need to load halo data as well
  - Needed to calculate edge elements of the column
  - For each 3D nxmxp output block to be computed, we need to load (n+2)x(m+2)x(p+2) inputs..



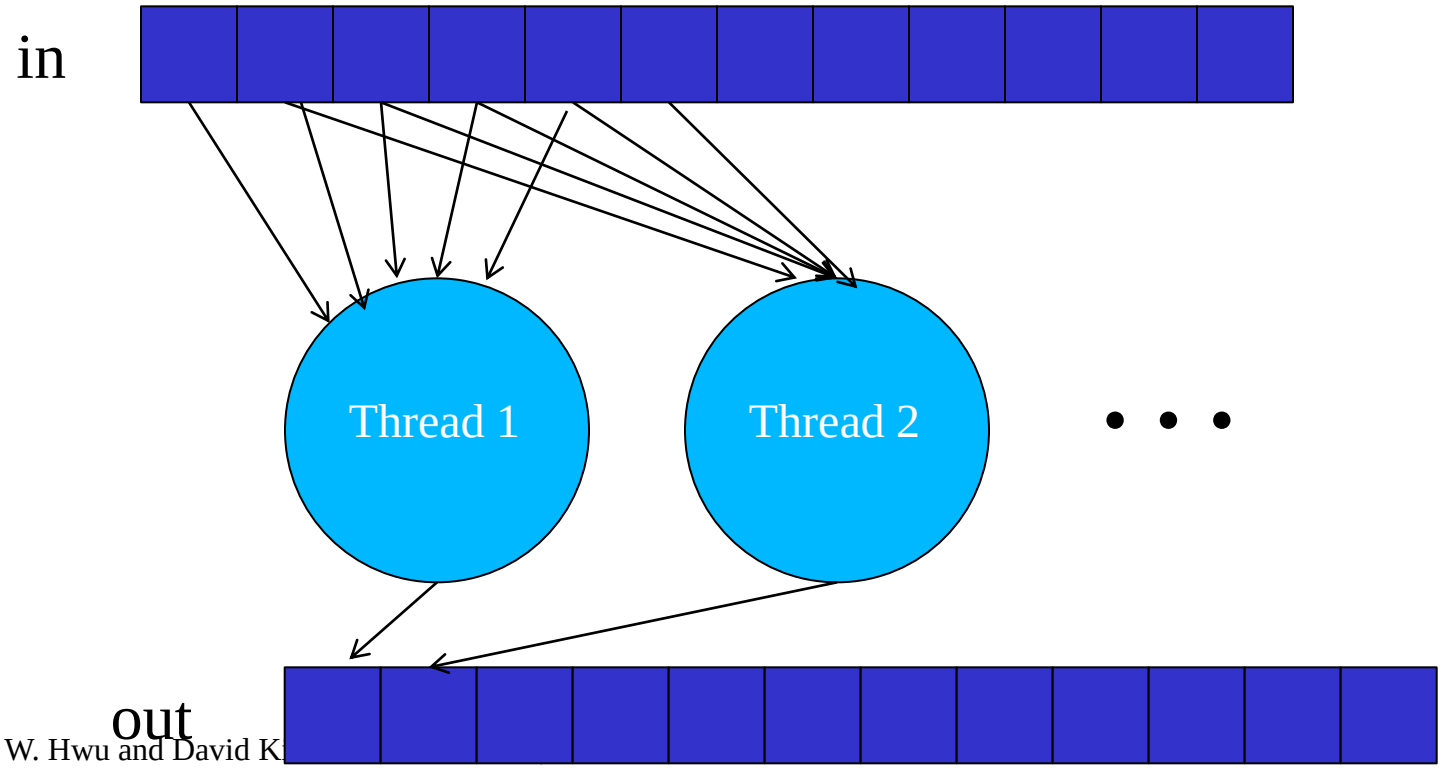
Halo overhead can hurt.

- For small n and m, the halo overhead can be very significant
  - If n=16 and m = 8, each slice calculates 16*8=128 output elements in each slice and needs to load 18*10=180 input elements
  - In optimized naive code, each output element needs 5 loads from global memory, a total of 5*128=640 loads
  - The total ratio of improvement is 640/180 = 3.5, rather than 5 times
  - The value of n and m are limited by the amount of registers and shared memory in each SM

CONVOLUTION EXAMPLE

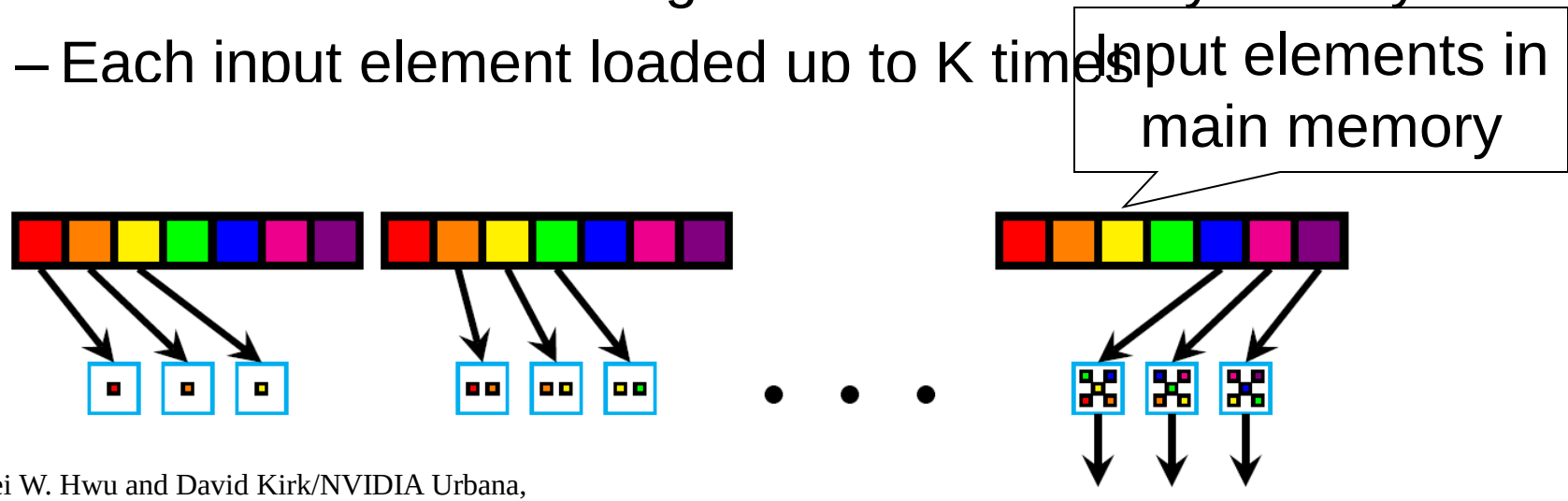
Convolution

- A convolution kernel is used to change an input element to the weighed linear combination of its neighbors.



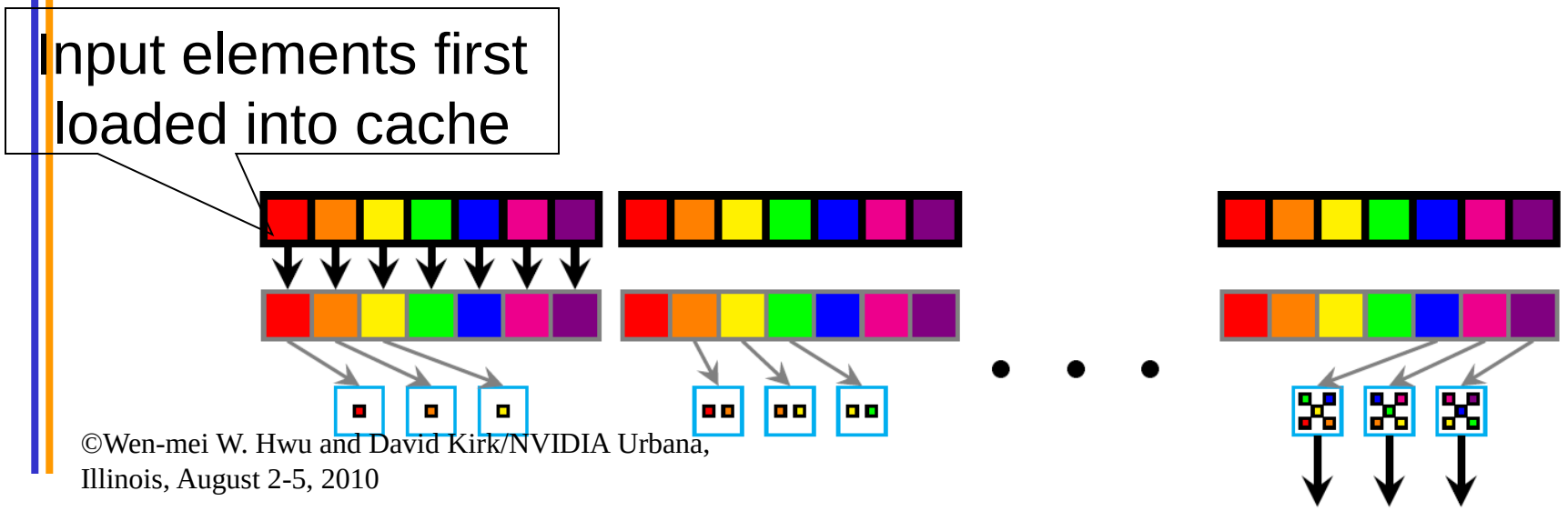
Example: Convolution – Base Parallel Code

- Each parallel task calculates an output element
- Figure shows
  - 1D convolution with K=5 kernel
  - Calculation of 3 output elements
- Highly parallel but memory bandwidth inefficient
  - Uses massive threading to tolerate memory latency
  - Each input element loaded up to K times



Example: convolution using on-chip caching

- Output elements calculated from cache contents
  - Each input element loaded only once
  - Cache pressure – (K-1+N) input elements needed for N output elements
    - 7/3 = 2.3, $7^2/3^2 = 5.4$, $7^3 / 3^3 = 12$
    - For small caches, the benefit can be significantly reduced due to the high-ratio of additional elements loaded.



Tiling for convolution is both easy and hard

- Convolution is conceptually easy to block
  - All threads that calculate a tile of output can share a tile of input
- High-performance tiling can be hard to achieve
  - Larger convolution kernels increase data sharing but also increase halo overhead
  - Many 3D convolution kernels cannot be done by traversing along the z-dimension sequentially
  - Higher-dimension convolution quickly run out of on-chip memory space

ANY MORE QUESTIONS?