Berkeley Winter School

Advanced Algorithmic Techniques for GPUs

Lecture 5: Advanced Data Optimizations

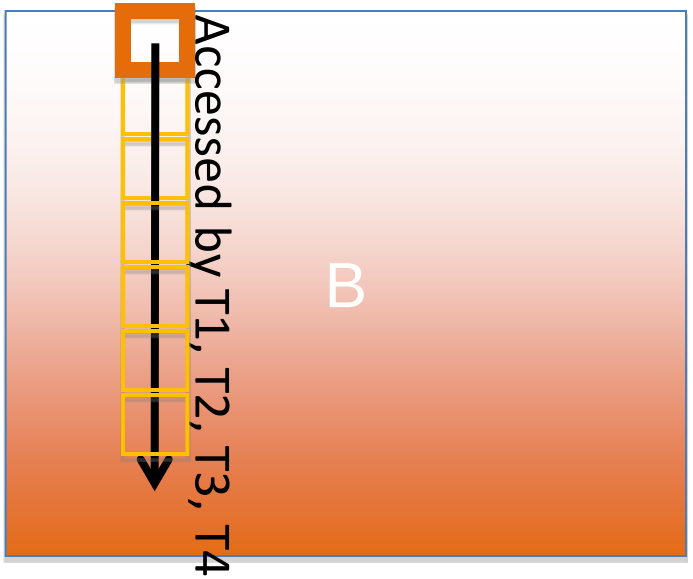
©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010

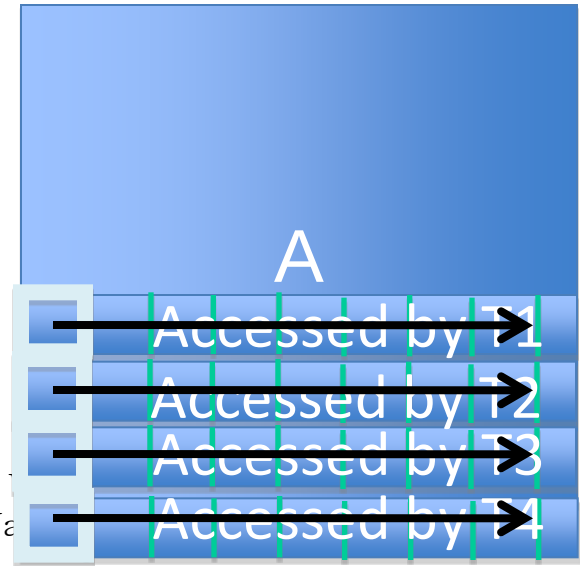
Objective

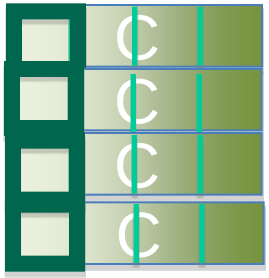
- Apply tiling, thread coarsening, and data layout transformations to one kernel
- Understand the practical use of these techniques

©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010

Data Reuse in Matrix-Matrix Multiplciation Revisited

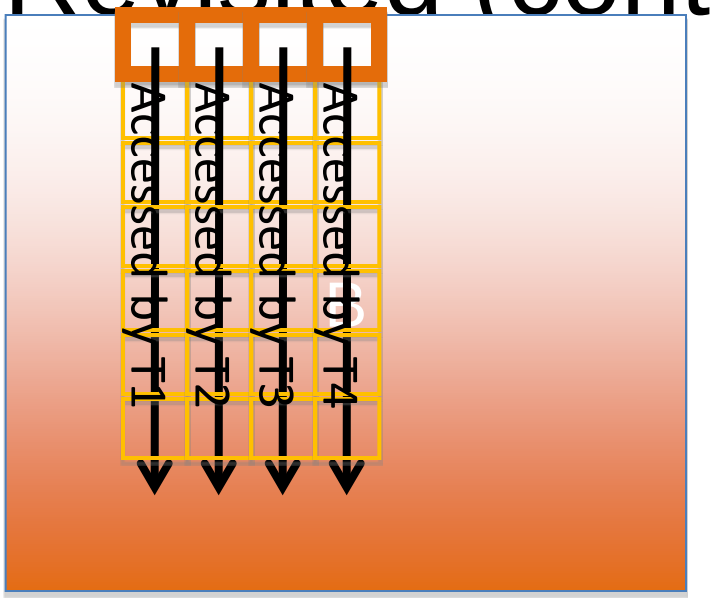


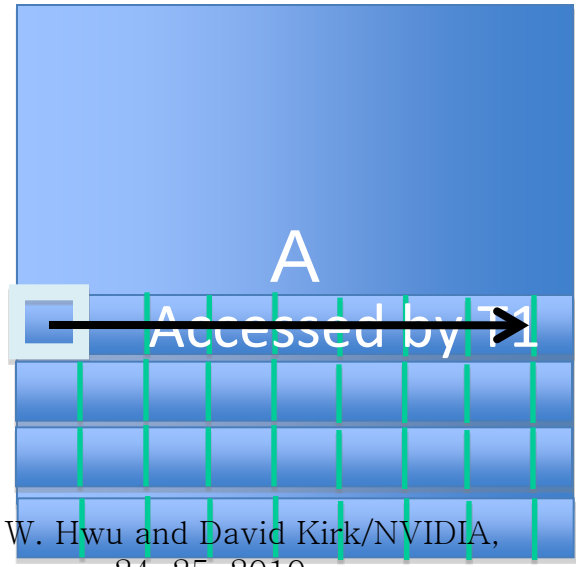


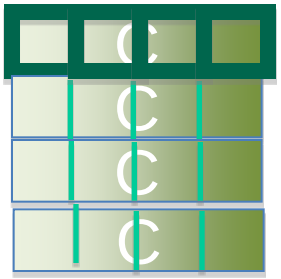


©Wen-mei
Berkeley, Ja

Data Reuse in Matrix-Matrix Multiplciation Revisited (cont.)



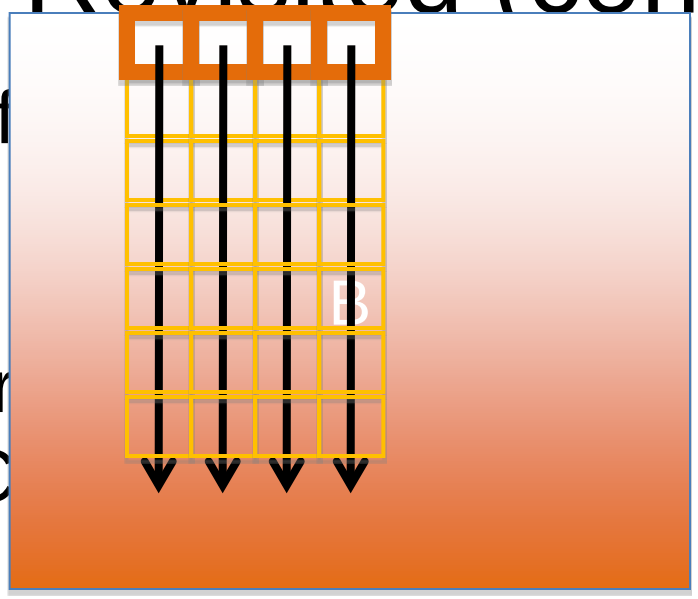


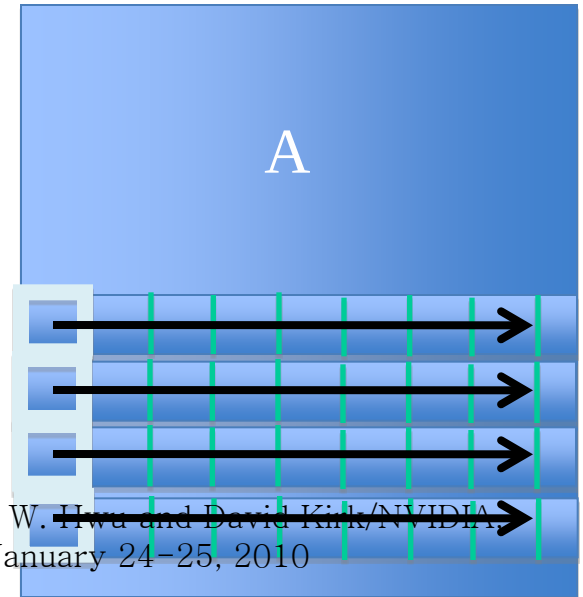


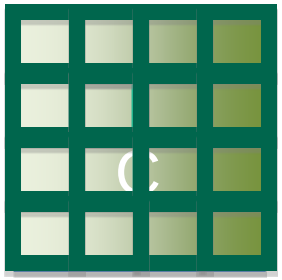
©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

Data Reuse in Matrix-Matrix Multiplciation Revisited (cont.)

- Only four elements of A and four elements of B is needed to calculate one step for a 16-element tile of C

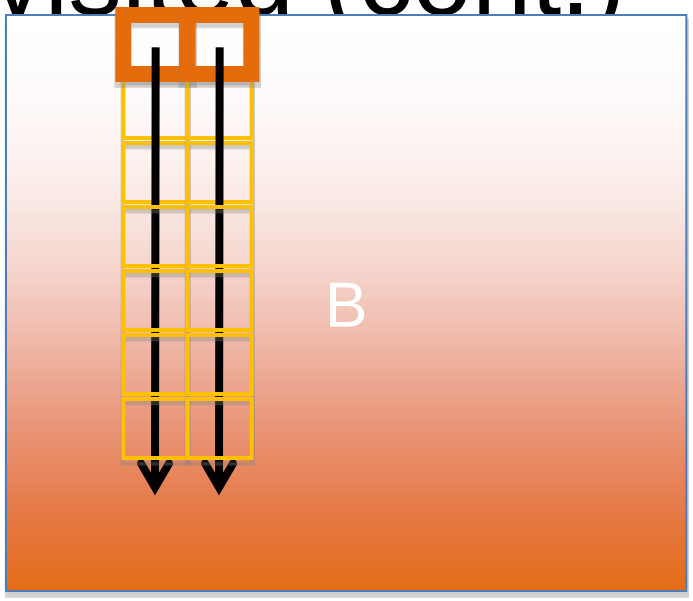


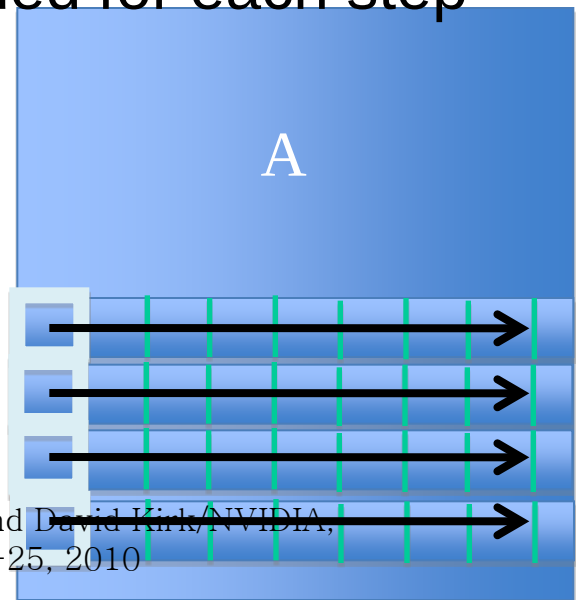


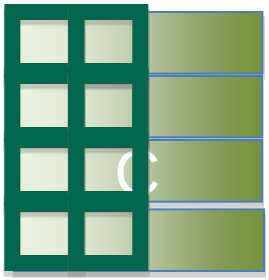


Data Reuse in Matrix-Matrix Multiplciation Revisited (cont.)

- The C tile does not need to be square
- This is a 4X2 tile
  - 4 elements of A and 2 elements of B are needed for each step



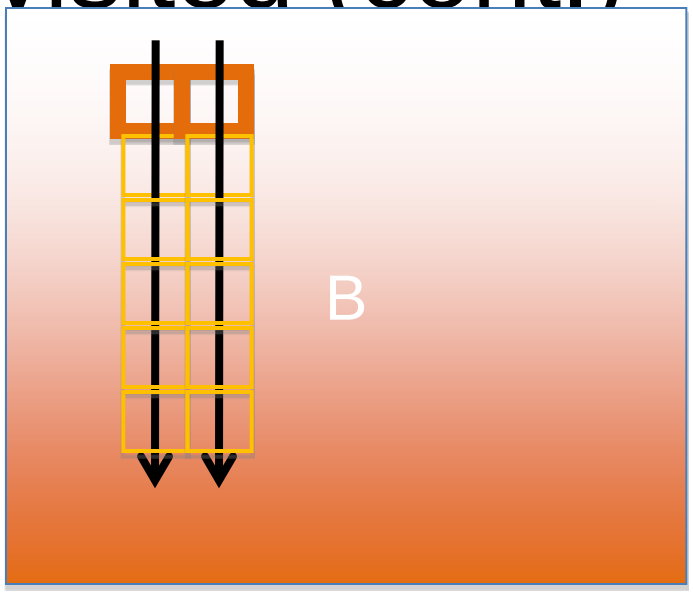


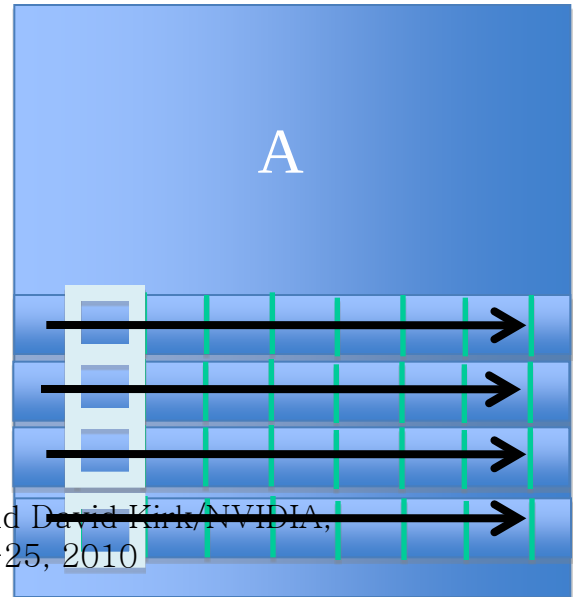


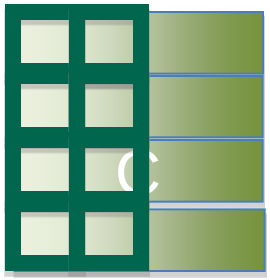
©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

Data Reuse in Matrix-Matrix Multiplciation Revisited (cont.)

- Step 2…



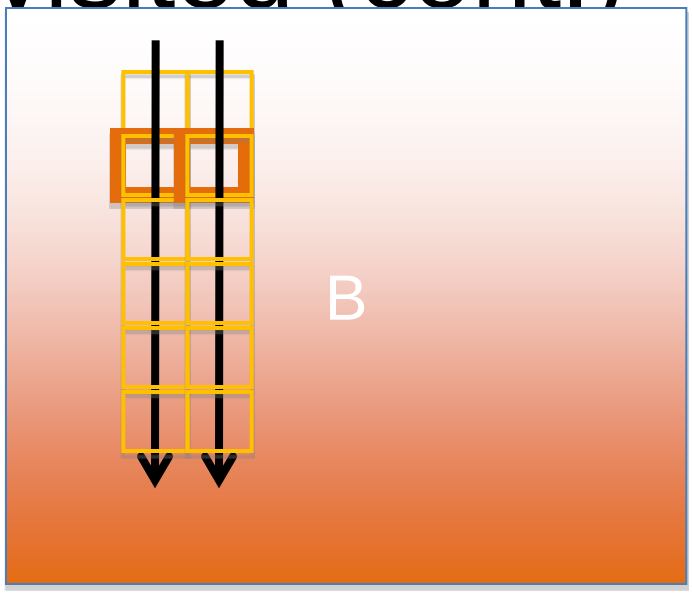


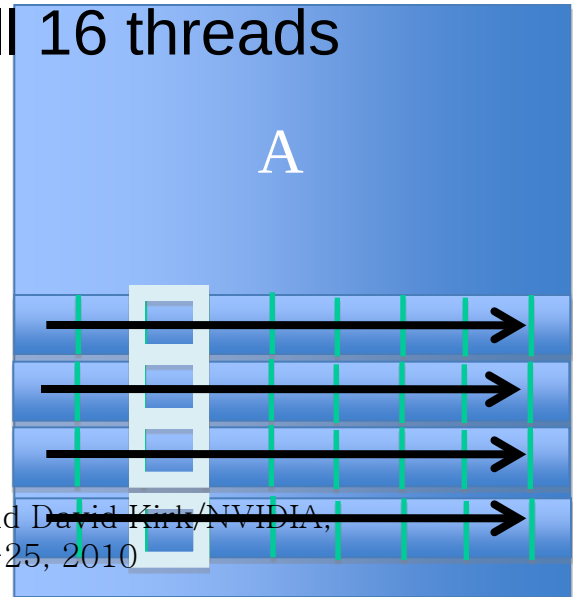


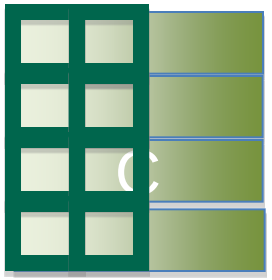
©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

Data Reuse in Matrix-Matrix Multiplciation Revisited (cont.)

- At each step
  - For 4X2 only 6 elements need to be loaded for all 8 threads to make progress
  - For 4X4, 8 elements for all 16 threads







©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

But, how about the kernel we saw.

```
__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
6.  int Col = bx
7.  float Pvalue
// Loop over the                    d to compute the Pd element
8.   for (int m = 0;     Width/TILE_WIDT
// Coolaborative loading of Md and Nd ti
9.     Mds[tx][ty] = Md[Row*Width + m*TI
10.    Nds[tx][ty] = Nd[(m*TILE_WIDTH +
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;
}
```

Each thread loads 1 element of A and 1 element of B

Each thread calculates TILE_WIDTH steps of a C element

In the kernel of the previous slide

- T^2 elements of A and T^2 element of B are loaded to calculate T steps for T^2 elements of C
- According to our analysis, we can use much smaller amount of shared memory by
  - Loading T element of A and T element of B to claculate 1 step for T^2 elements of C
  - Or loading TA elements of A and TB elements of B to calculate 1 step for TA*TB elements of C (rectangular matrix)
  - So, why didn't we do so?

©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010

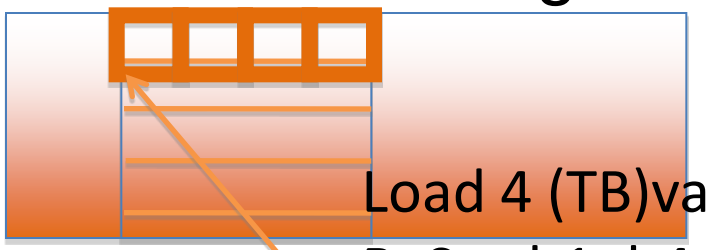
Synchronization Overhead

- We need to call __synchthreads() in the inner loop of each thread. In each iteration
  - only a subset of threads load A and B elements (divergence)
  - Call __synchthreads()
  - All threads calculate one step of the inner product
  - Call __synchtrheads()
  - Go to the next iteration
- Even though __synchtrheads() is a very efficient function, such intensive use is still going to hurt

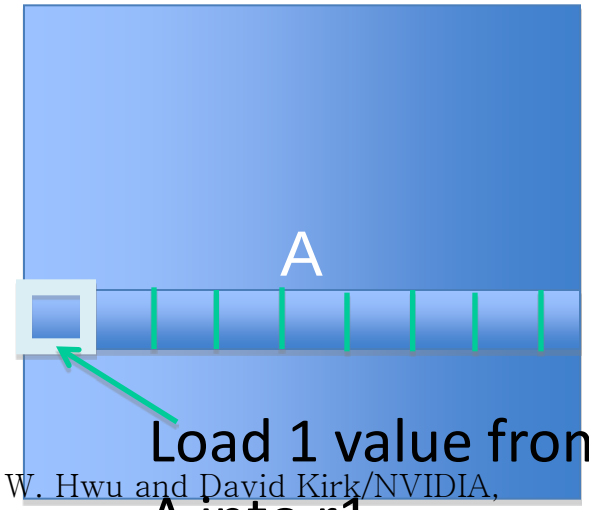
©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010

A somewhat different approach Optimization 1: thread coarsening

- Have each thread to calculate a horizontal subset of C elements
- Data loaded in A can be reused through registers
  - Register tiling



Load 4 (TB)values from B. Say b1~b4



Load 1 value from A into r1

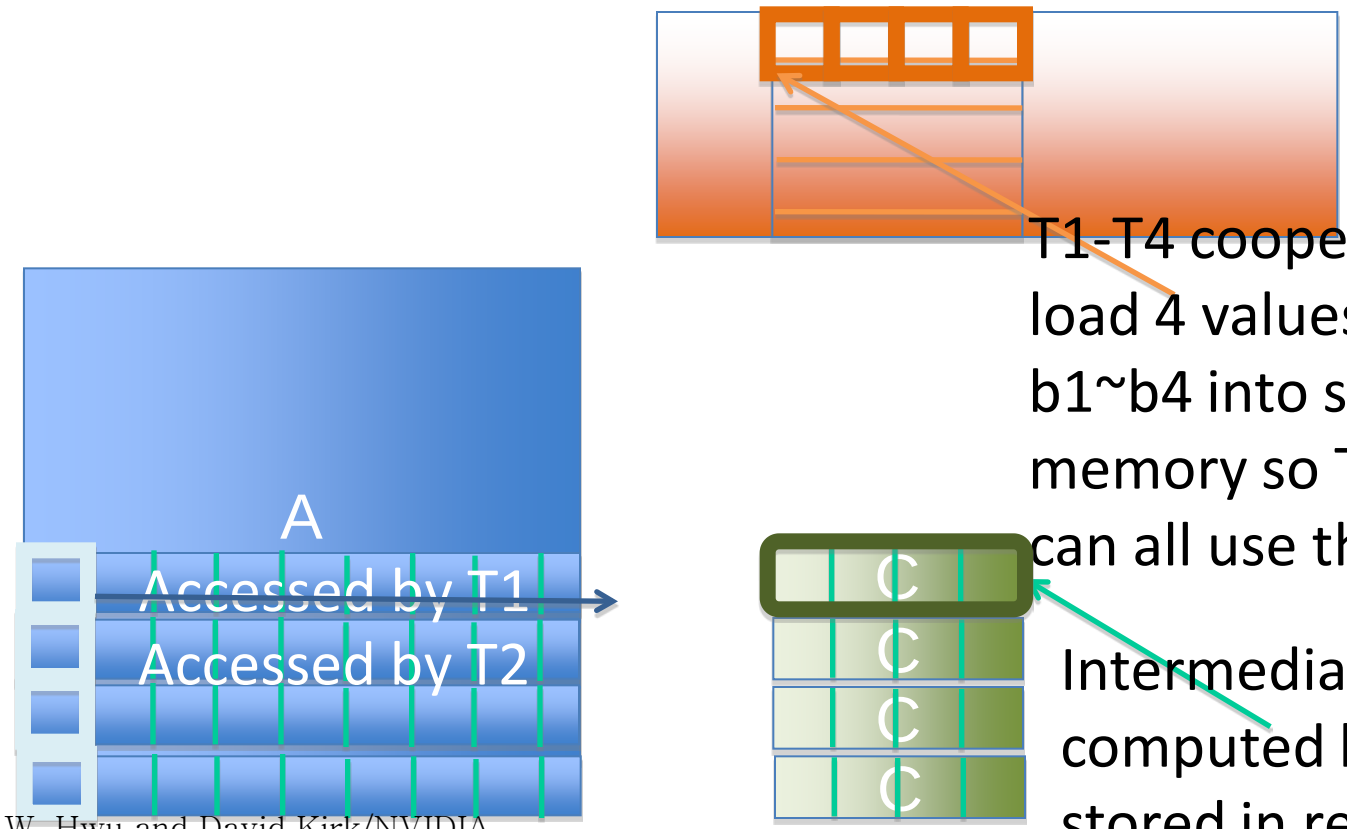
r1 can be reused to compute 4 intermediate results for C

```
C[0] += r1 * b1;
C[1] += r1*  b2;
C[2] += r1*  b3;
C[3] += r1*  b4;
```

©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

Optimization 2: Shared memory tiling

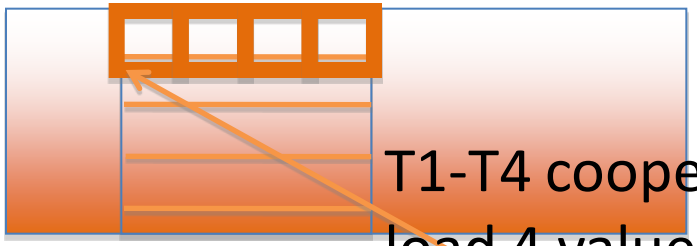
- Multiple threads collaborate to load TB B elements into shared memory



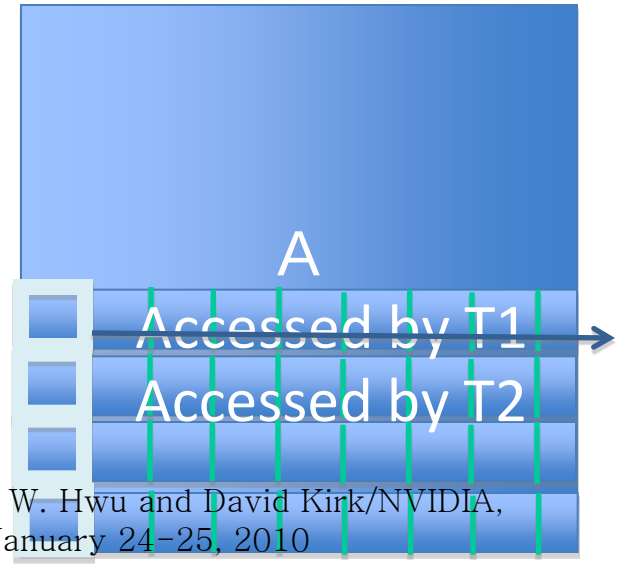
©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

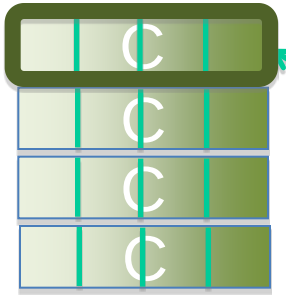
In one iteration, each thread

- loads one A element into register, accesses TB B elements from shared memory
  - Calculates one step for 1*TB C elements
  - TB ~16 in practice



T1-T4 cooperatively load 4 values from B, b1~b4 into shared memory so T1-T4 can all use them





Intermediate results computed by T1; stored in registers

©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

In one iteration, each block

- Loads TA A elements into registers, loads TB B elements into shared memory
  - TA is number of threads in thread block (64 or more in practice)
  - TB is number of threads folded into one thread in thread coarsening (16 or more in practice)
- However, loading of B will involve only a subset of threads (divergence)

©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010

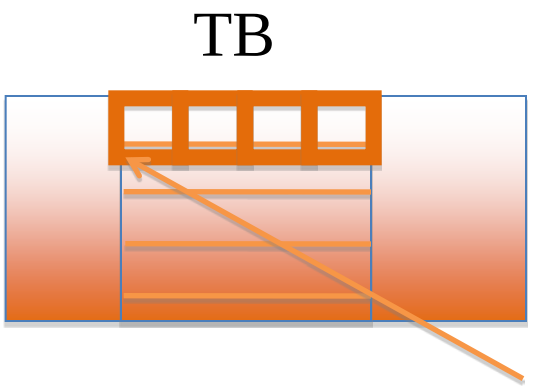
A more balanced approach, in each iteration

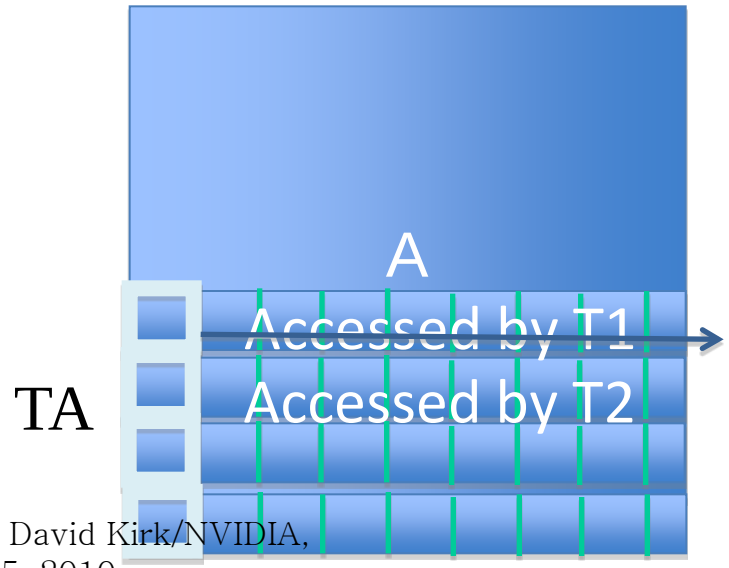
- All threads in a block collaborate to load a TBxK tile of B into shared memory
  - K is set so that TA = TB*K
  - Every thread loads one B element, no divergence
- Each thread loads K A elements into registers
- Each thread calculates K steps for TB C elements

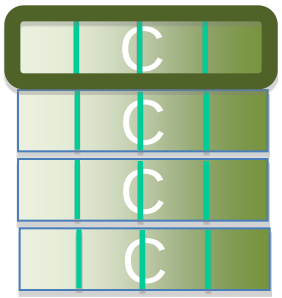
©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010

Summary

- Each block has TA threads
- Each thread coarsened by TB times
- Each thread loads
  - One B element
  - K =TA/TB A elements
  - To calculate K steps of TB C elements



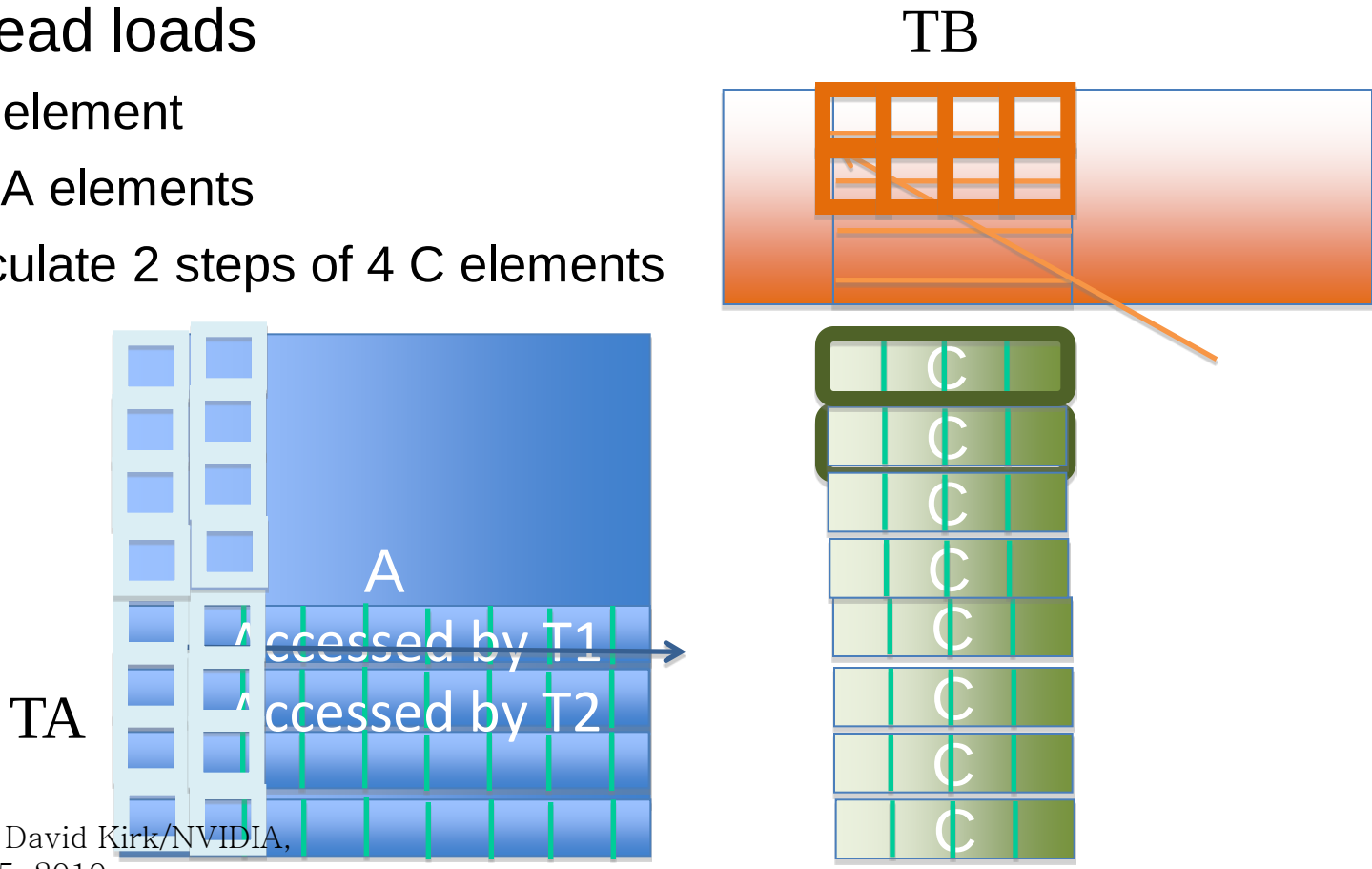




©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

For a toy example

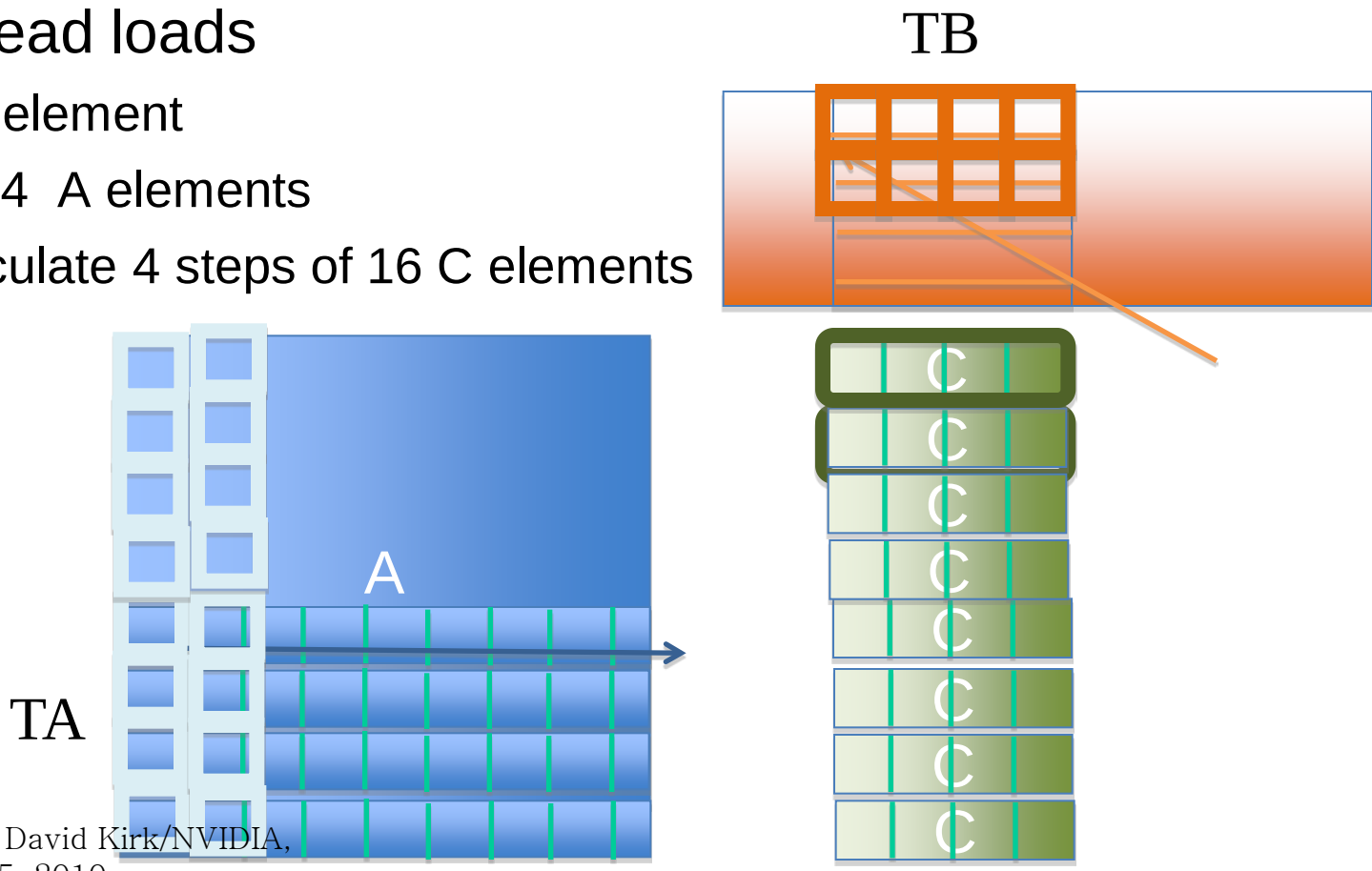
- Each block has 8 threads
- Each thread coarsened by 4 times
- Each thread loads
  - One B element
  - 8/4=2 A elements
  - To calculate 2 steps of 4 C elements



©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

For GTX280 (Volkov & Demmel)

- Each block has 64 threads
- Each thread coarsened by 16 times
- Each thread loads
  - One B element
  - 64/16=4 A elements
  - To calculate 4 steps of 16 C elements



©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

A Comparative Analysis

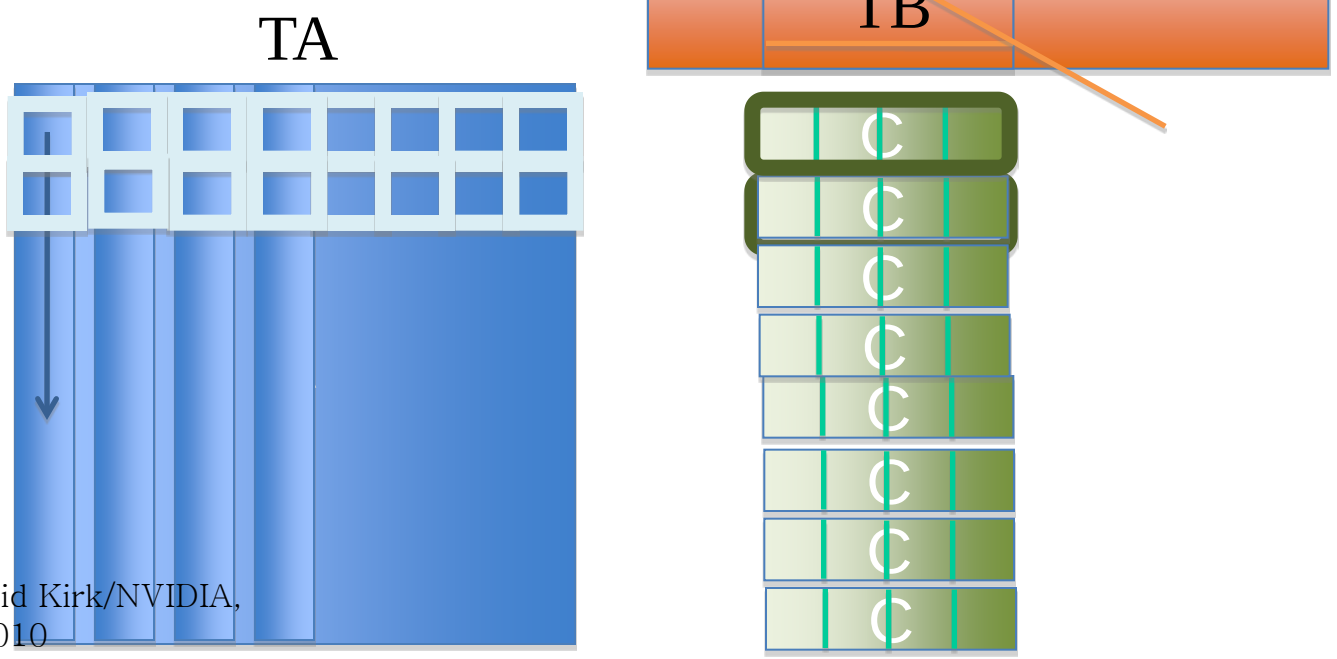
- Tiled MM introduced earlier:
  - Each thread block computes 32x32=1024 results
  - Use 9 KB on-chip memory (register + shared memory)
- Register tiled version of sgemm:
  - Each thread block computes 64x16=1024 results
  - Use only 4 ¼ KB on-chip memory
    - Similar degree of reuse; ~2X more efficient than tiled MM

Tiling algorithm	# of reuse per data in A	# of reuse per data in B	# of data computed per block in C	Shared memory usage per block	Register usage per TB	Performance on GTX280 in GFLOP/s
Register tiled MM	16	64	16x64	4x16x4 =256Bytes	64x16x4 =4KB	~430
Tiled MM	32	32	32x32	32x32x4x2 =8KBytes	32x32=1KB	<300

©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

Data Layout – For C (row major)

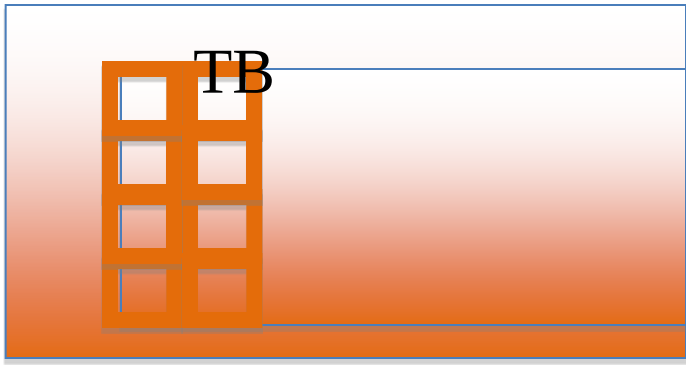
- Loading B into shared memory is easily coalesced with the 16X4 tile
- Loading A into registers in not coalesced
  - Transpose A for coalescing

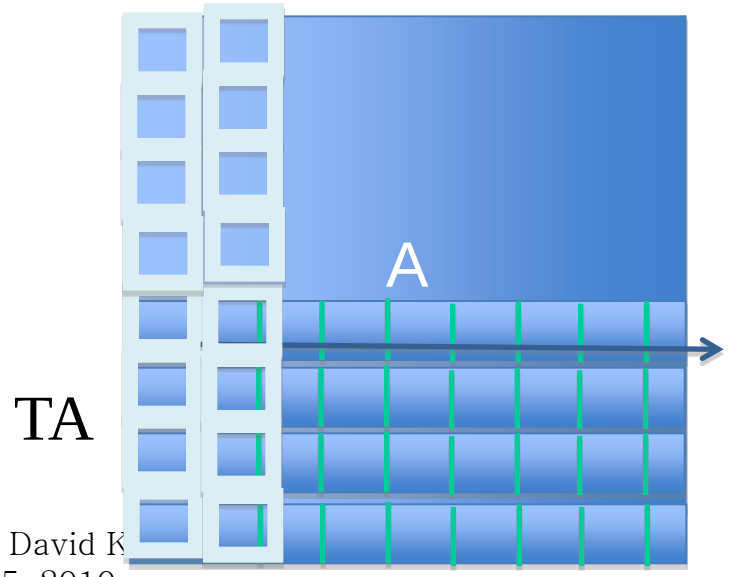


©Wen-mei W. Hwu and David Kirk/NVIDIA, Berkeley, January 24-25, 2010

Data Layout for FORTRAN

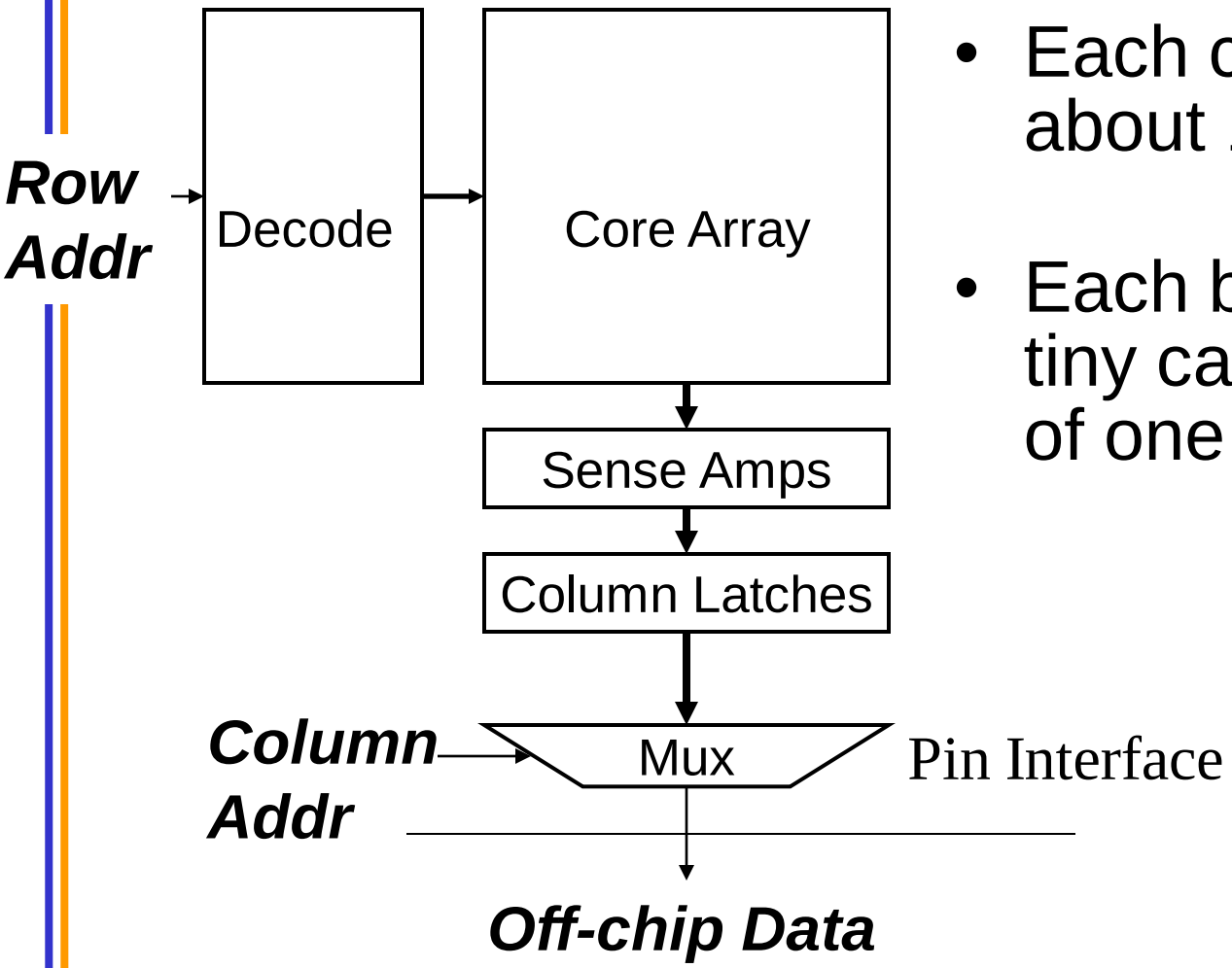
- Column major layout
- A accesses are coalesced
- B needs to be transposed
- C may need to be transposed





©Wen-mei W. Hwu and David K
Berkeley, January 24-25, 2010

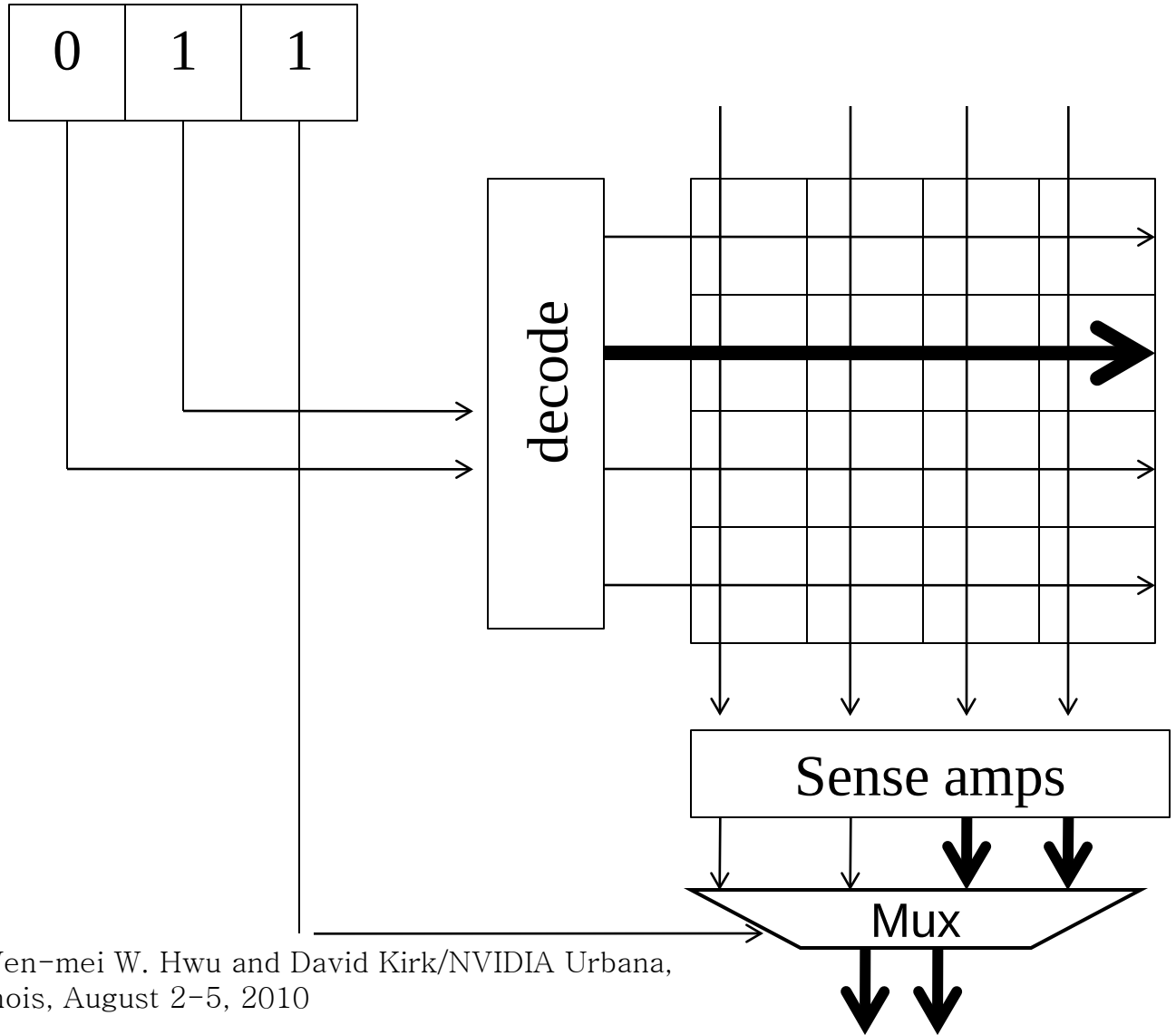
DRAM Bank Organization



- Each core array has about 1M bits
- Each bit is stored in a tiny capacitor, made of one transistor

© Wen-mei Hwu and S. J. Patel, 2005
ECE 511, University of Illinois

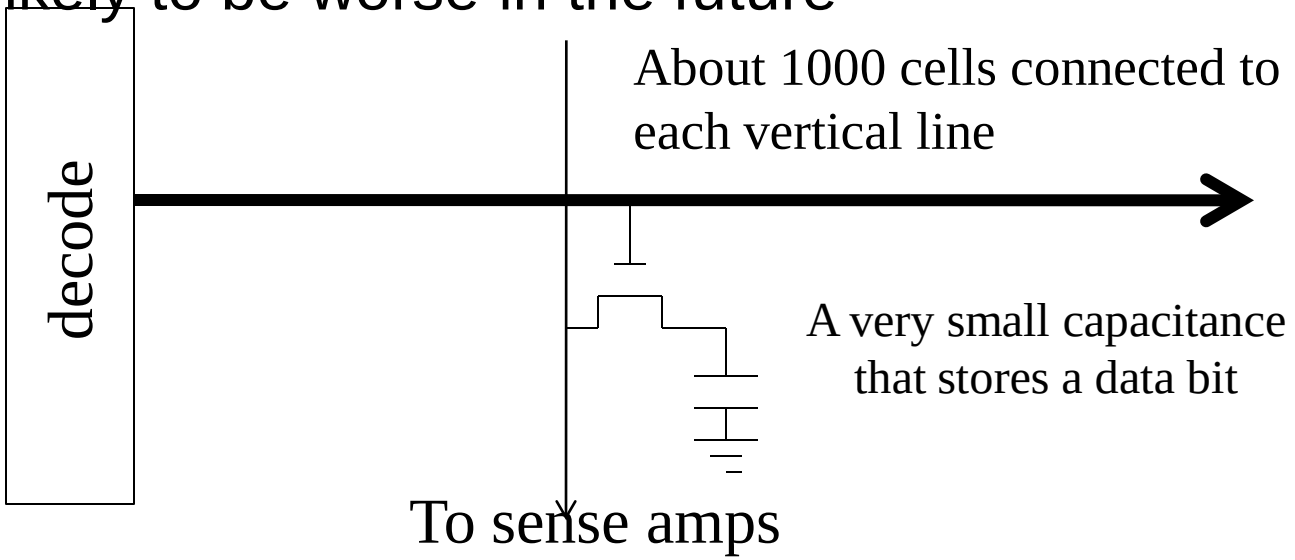
A very small (8x2 bit) DRAM Bank



©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

DRAM core arrays are slow.

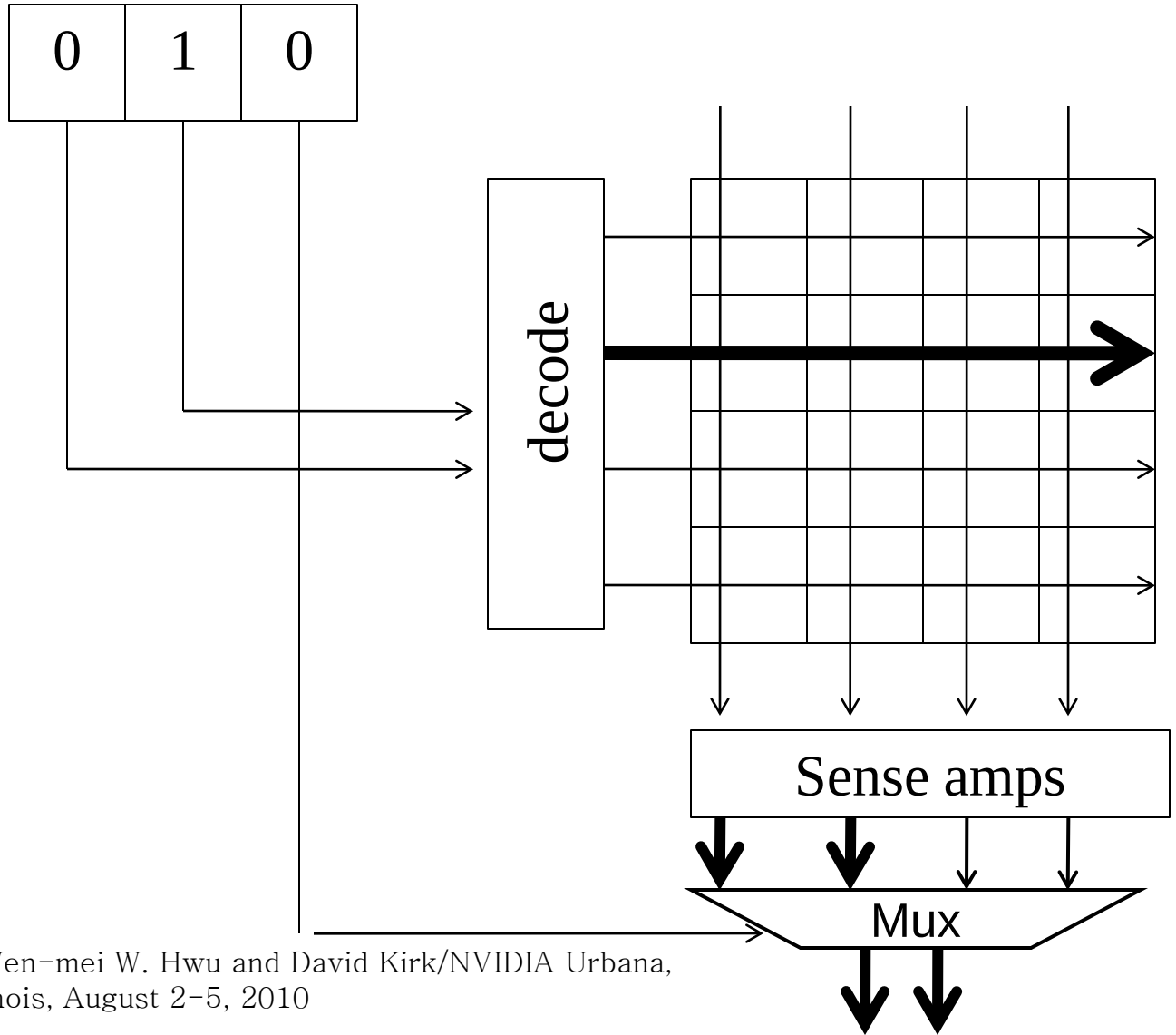
- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - … likely to be worse in the future



DRAM Bursting.

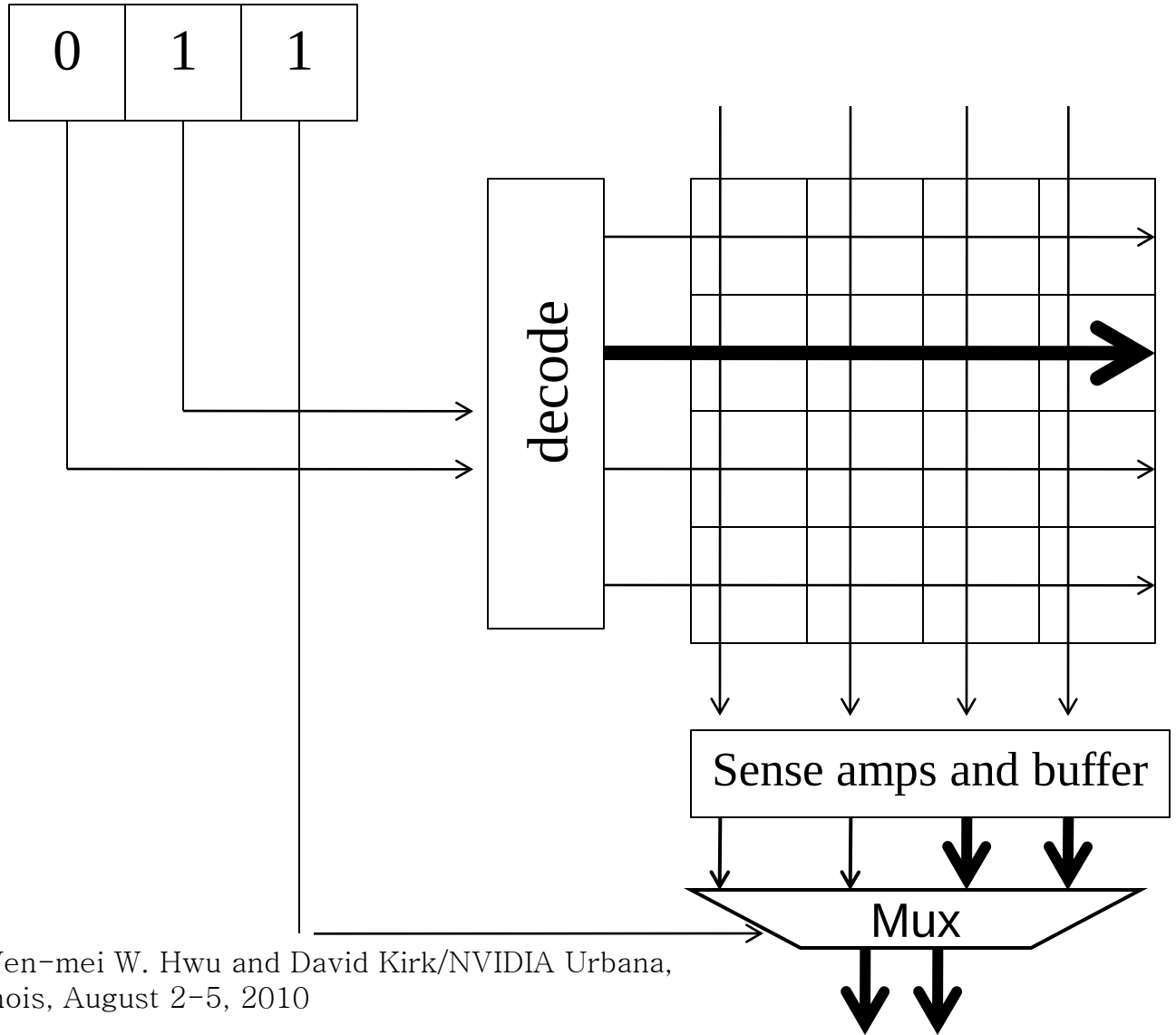
- For DDR{2,3} SDRAM cores clocked at 1/N speed of the interface:
  - Load (N × interface width) of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed
  - DDR2/GDDR3: buffer width = 4X interface width

DRAM Bursting



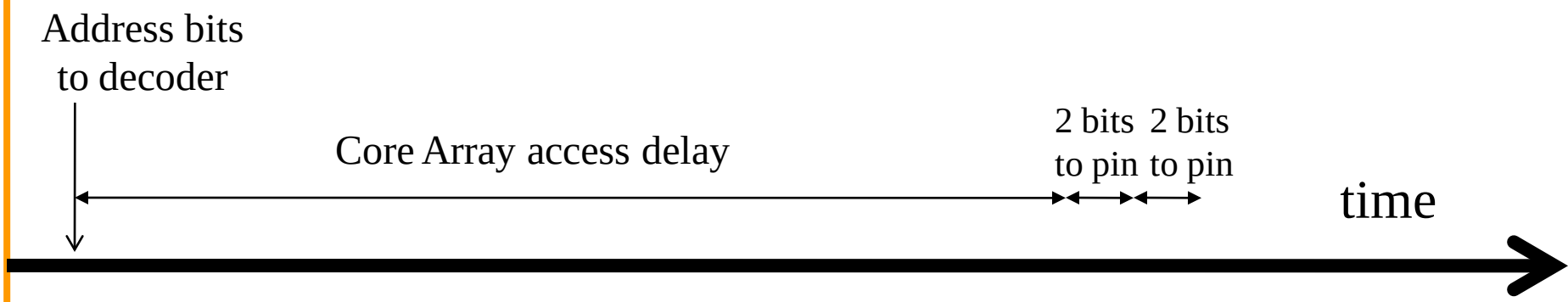
©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

DRAM Bursting



©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

DRAM Bursting for the 8x2 Bank



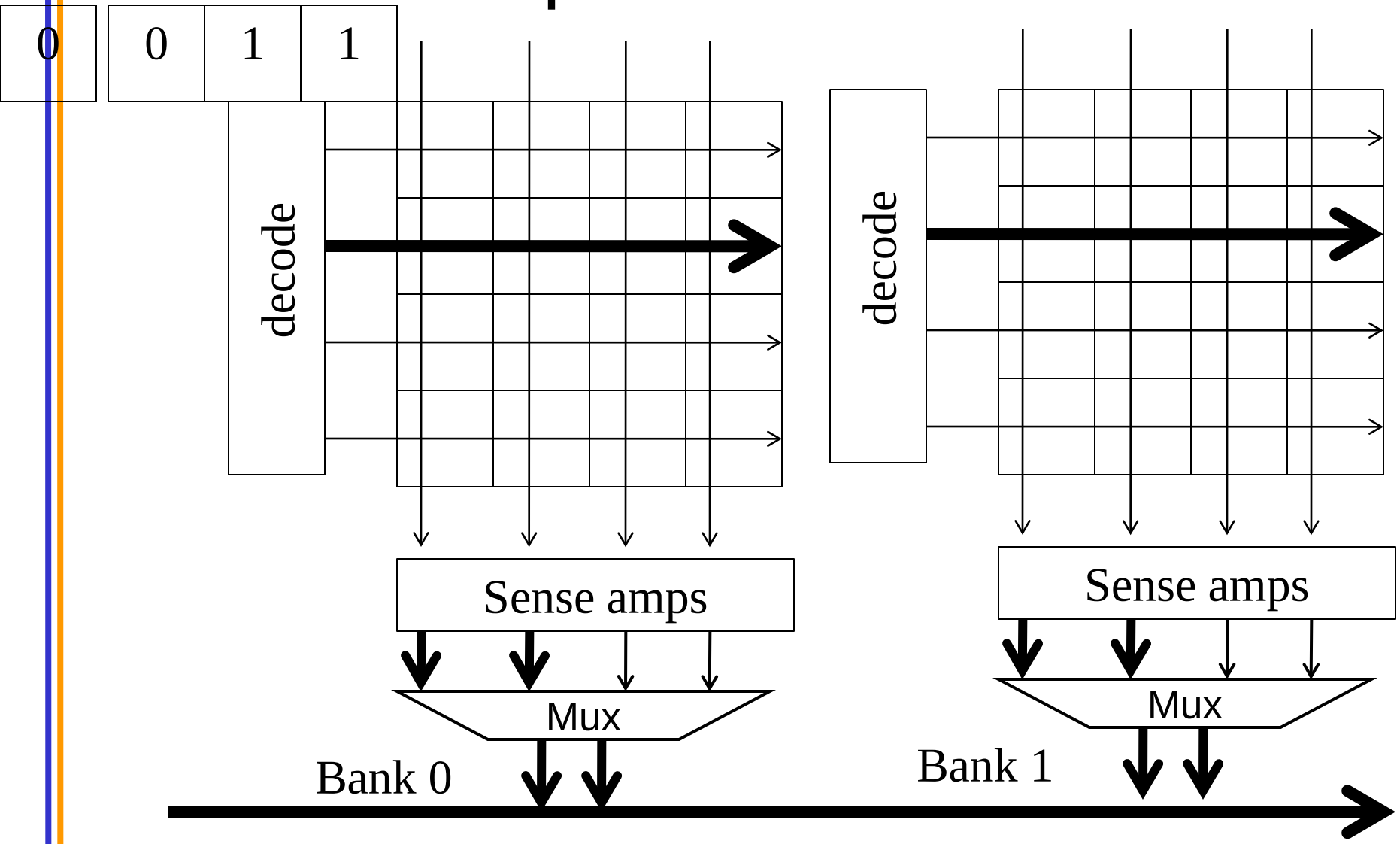
Non-burst timing

Burst timing

Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.

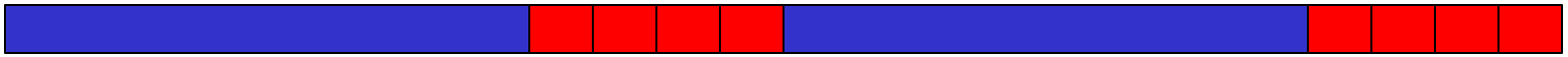
©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

Multiple DRAM Banks

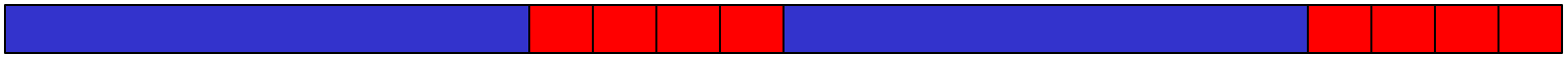


©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

DRAM Bursting for the 8x2 Bank



Single-Bank burst timing, dead time on interface



Multi-Bank burst timing, reduced dead time

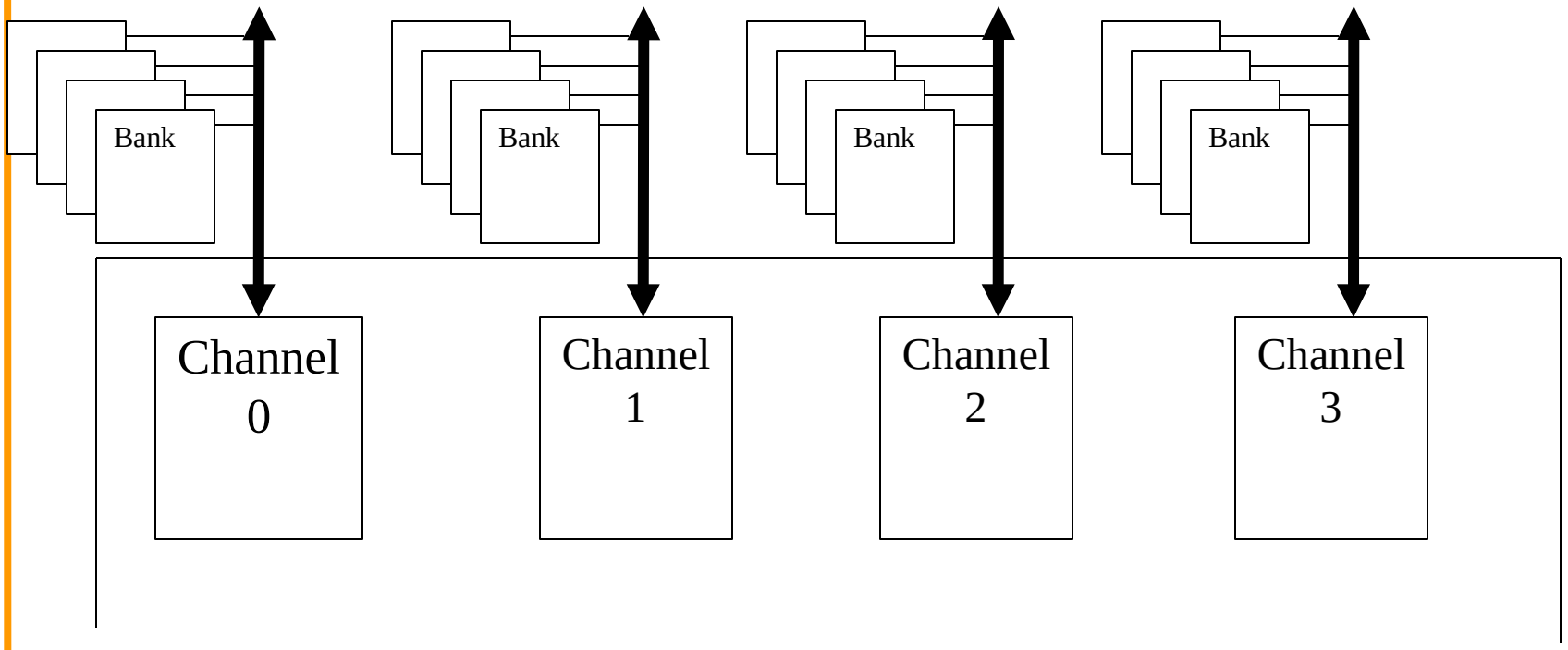
©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana,
Illinois, August 2-5, 2010

First-order Look at the GPU off-chip memory subsystem

- nVidia GTX280 GPU:
  - Peak global memory bandwidth = 141.7GB/s
- Global memory (GDDR3) interface @ 1.1GHz
  - (Core speed @ 276Mhz)
  - For a typical 64-bit interface, we can sustain only about 17.6 GB/s (Recall DDR - 2 transfers per clock)
  - We need a lot more bandwith (141.7 GB/s) – thus 8 memory channels

Multiple Memory Channels

- Divide the memory address space into N parts
  - N is number of memory channels
  - Assign each portion to a channel



©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

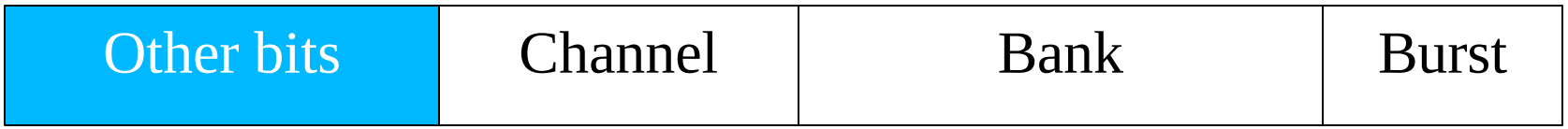
Memory Controller Organization of a Many-Core Processor

- GTX280: 30 Stream Multiprocessors (SM) connected to 8-channel DRAM controllers through interconnect
  - DRAM controllers are interleaved
  - Within DRAM controllers (channels), DRAM banks are interleaved for incoming memory requests
  - We approximate its DRAM channel/bank interleaving scheme through micro-benchmarking

©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

Back to the Big Picture

- Each global memory access is made to a memory location with an address
  - Some bits will determine the memory channel used
  - Some bits will determine the DRAM bank used
  - Some bits will determine the position within a burst



- When adjacent threads in a warp access words in a burst, the accesses are coalesced.

©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010

ANY MORE QUESTIONS?

©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2010