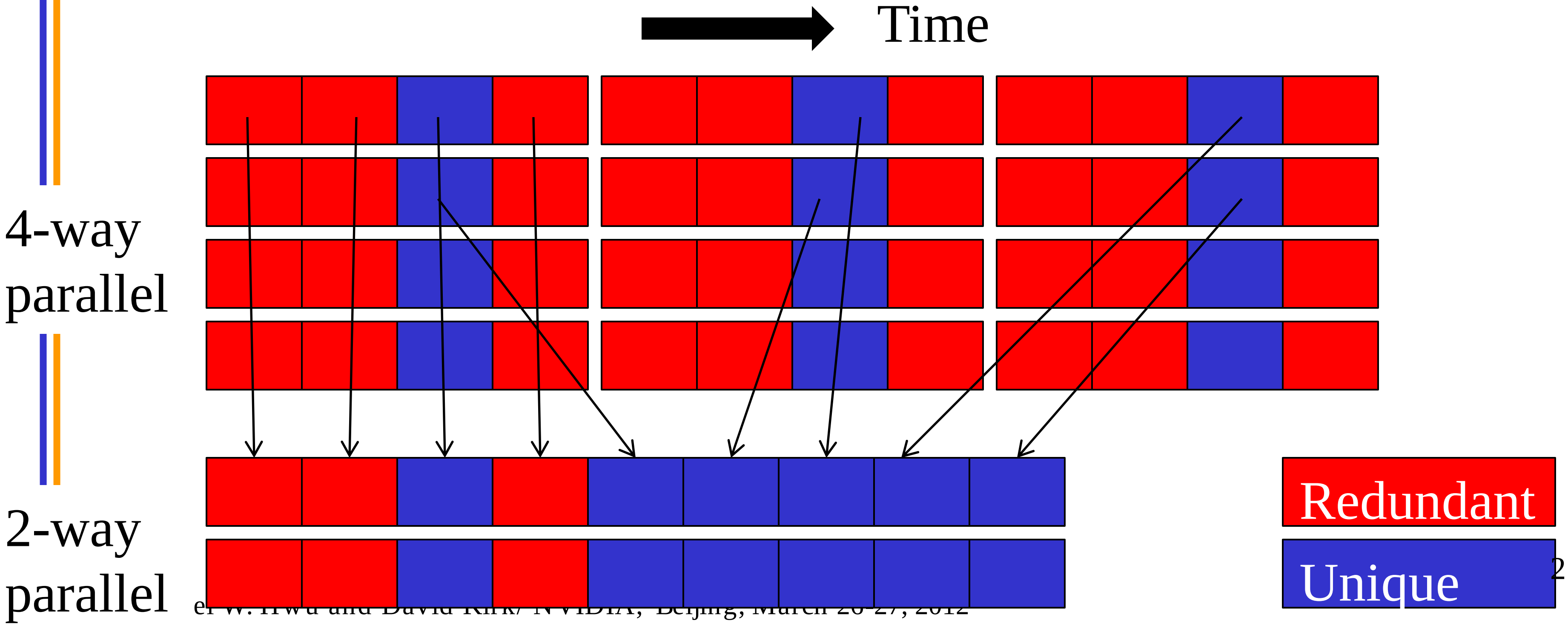
ICCS School

Advanced GPU Programming for Science

Lecture 3: Thread Coarsening and More on Tiling/Blocking

Thread Coarsening

- Parallel execution sometimes requires doing redundant memory accesses and/or calculations
  – Merging multiple threads into one allows re-use of result, avoiding redundant work



Outline of Technique

- Merge multiple threads so each resulting thread calculates multiple output elements
  - Perform the redundant work once and save result into registers
  - Use register result to calculate multiple output elements
- Merged kernel code will use more registers
  - May reduce the number of threads allowed on an SM
  - Increased efficiency may outweigh reduced parallelism, especially if ample for given hardware

Register Tiling

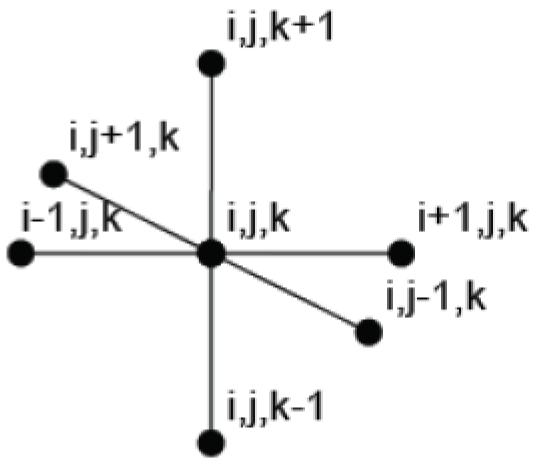
- Registers
  - extremely fast (short latency)
  - do not require memory access instructions (high throughput)
  - But – private to each thread
  - Threads cannot share computation results or loaded memory data through registers

- With thread coarsening
  - The computation from merged threads can now share registers

STENCIL CODE EXAMPLE

Stencil Computation

- Describes the class of nearest neighbor computations on structured grids.
- Each point in the grid is a weighted linear combination of a subset of neighboring values.
- Optimizations and concepts covered : Improving locality and Data Reuse
  - 2D Tiling in Shared Memory
  - Coarsening and Register Tiling



Stencil Computation

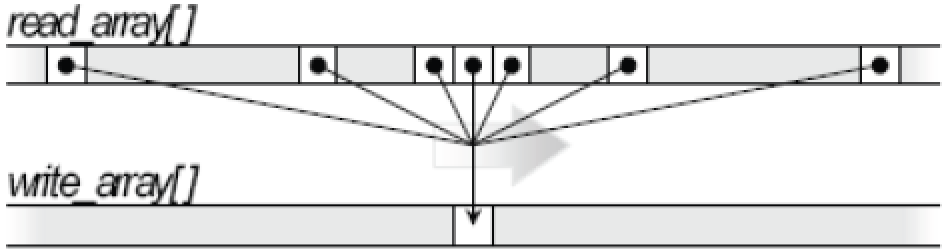
- High parallelism: Conceptually, all points in the grid can be updated in parallel.
- Each computation performs a global sweep through the data structure.
- Low computational intensity: High memory traffic for very few computations.
- Base case: one thread calculates one point
- Challenge: Exploit parallelism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

Memory Access Details

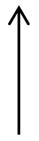
- General Equation:

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

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

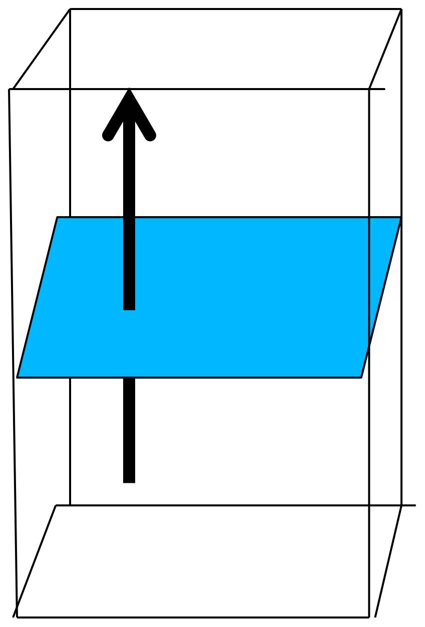


Coarsened implementation



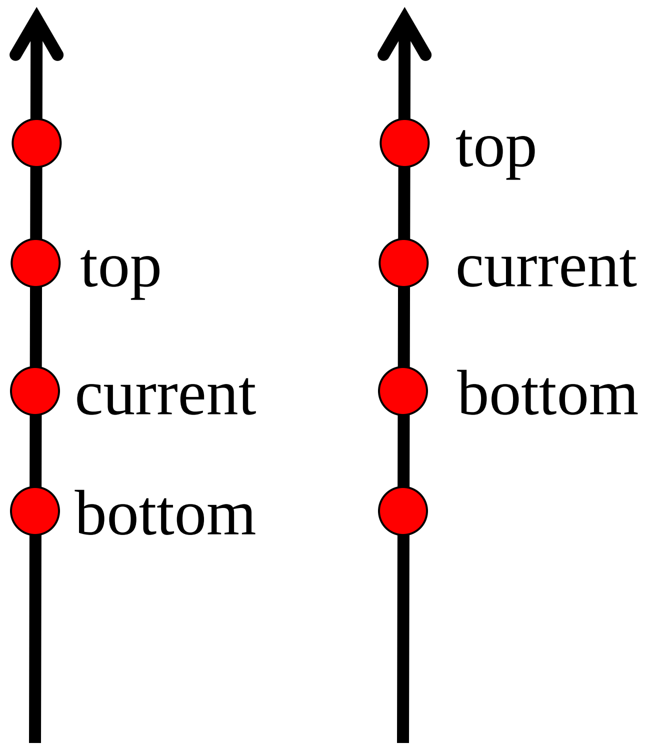
- Each thread calculates a one-element thin column along the z-dimension
  - Each block computes a rectangular column along the z-dimension

- Each thread loads all its input elements for an output point from global memory, independently of other threads
  - High read redundancy, heavy global memory traffic



Register Tiling

- 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

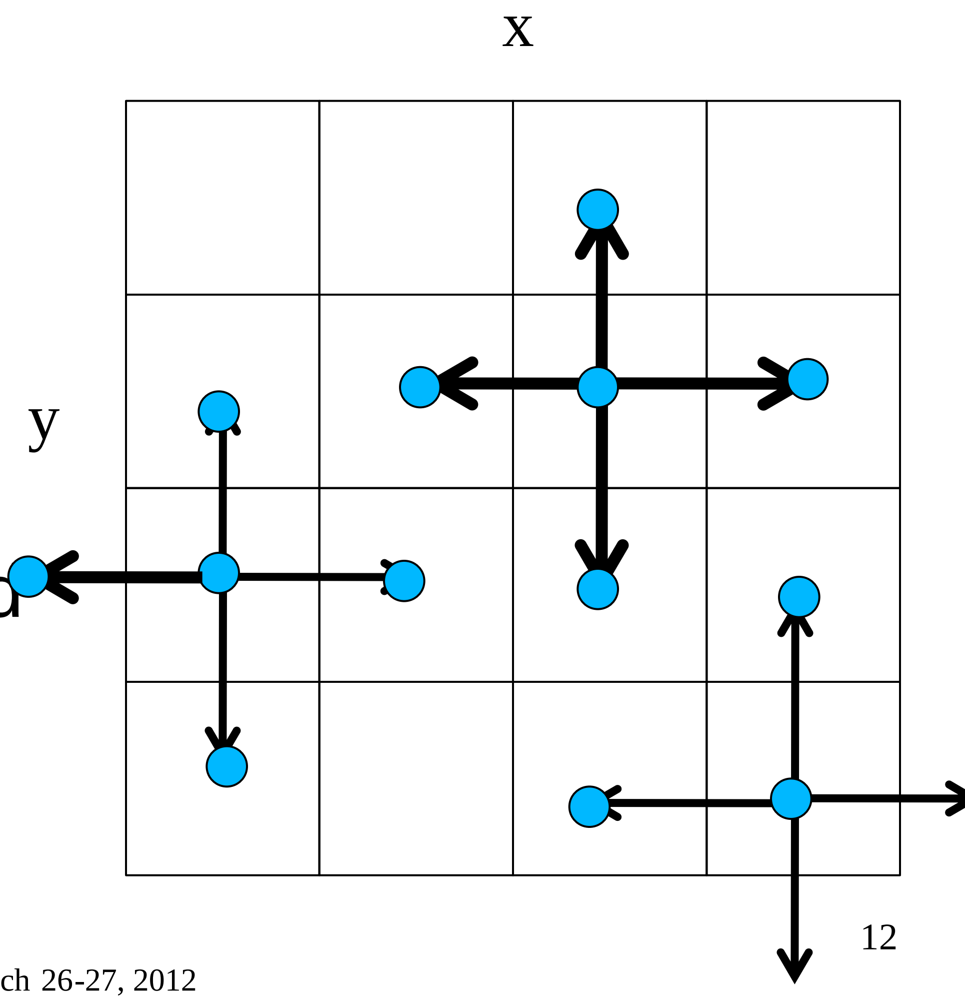


Savings of Coarsened Kernel

- 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

Cross-Thread 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



Sample Coarsened Kernel Code

```
i = bx * dx + tx;
j = by * dy + ty;
float bottom = Aorig[Index3D(Nx, Ny, i, j, 0)];
float current = Aorig[Index3D(Nx, Ny, i, j, 1)];
float top     = Aorig[Index3D(Nx, Ny, i, j, 2)];
/* Nx and Ny: width of the grid in x and y directions, given as kernel arguments */
for (k=1, k < Nz-1, k++)  {
    Anext[Index3D(Nx,Ny,i,j,0)] = bottom + top +
                    (i==0)? 0: Aorig[Index3D(Nx, Ny, i-1, j, k) +
                    (i==Nx-1)? 0: Aorig[Index3D(Nx, Ny, i+1, j, k) +
                    (j==0)? 0: Aorig[Index3D(Nx, Ny, i, j-1, k) +
                    (j==Ny-1)? 0: Aorig[Index3D(Nx, Ny, i, j+1, k) –
                    6 * current / (fac * fac);
    bottom = current;
    current = top;
}
```

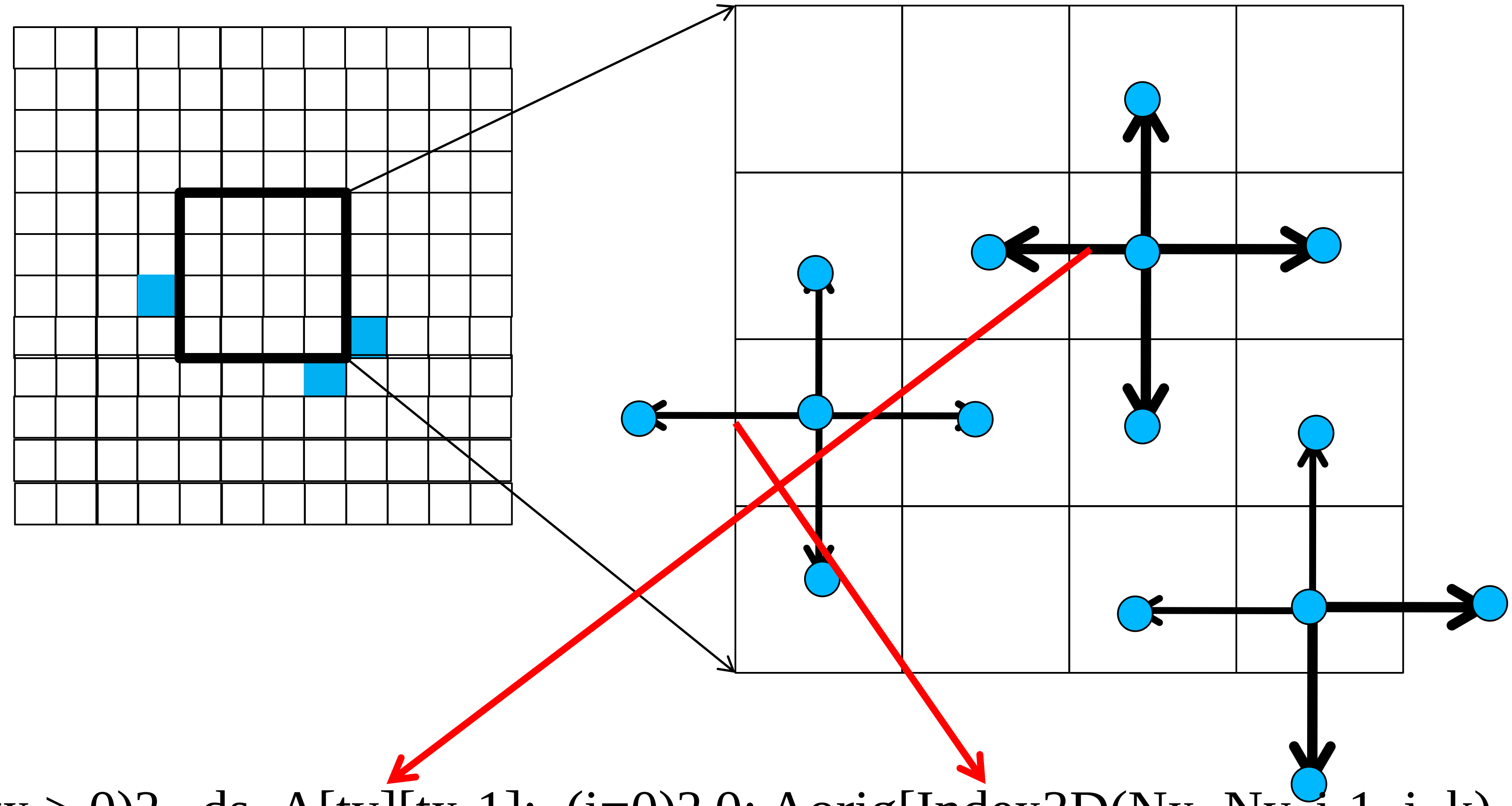
Improving Locality: 2D Tiling

- Assume that all threads of a block march up the z-direction in synchronized phases
- In each phase, all threads calculate a 2-D slice 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**

Sample Coarsened Kernel Code

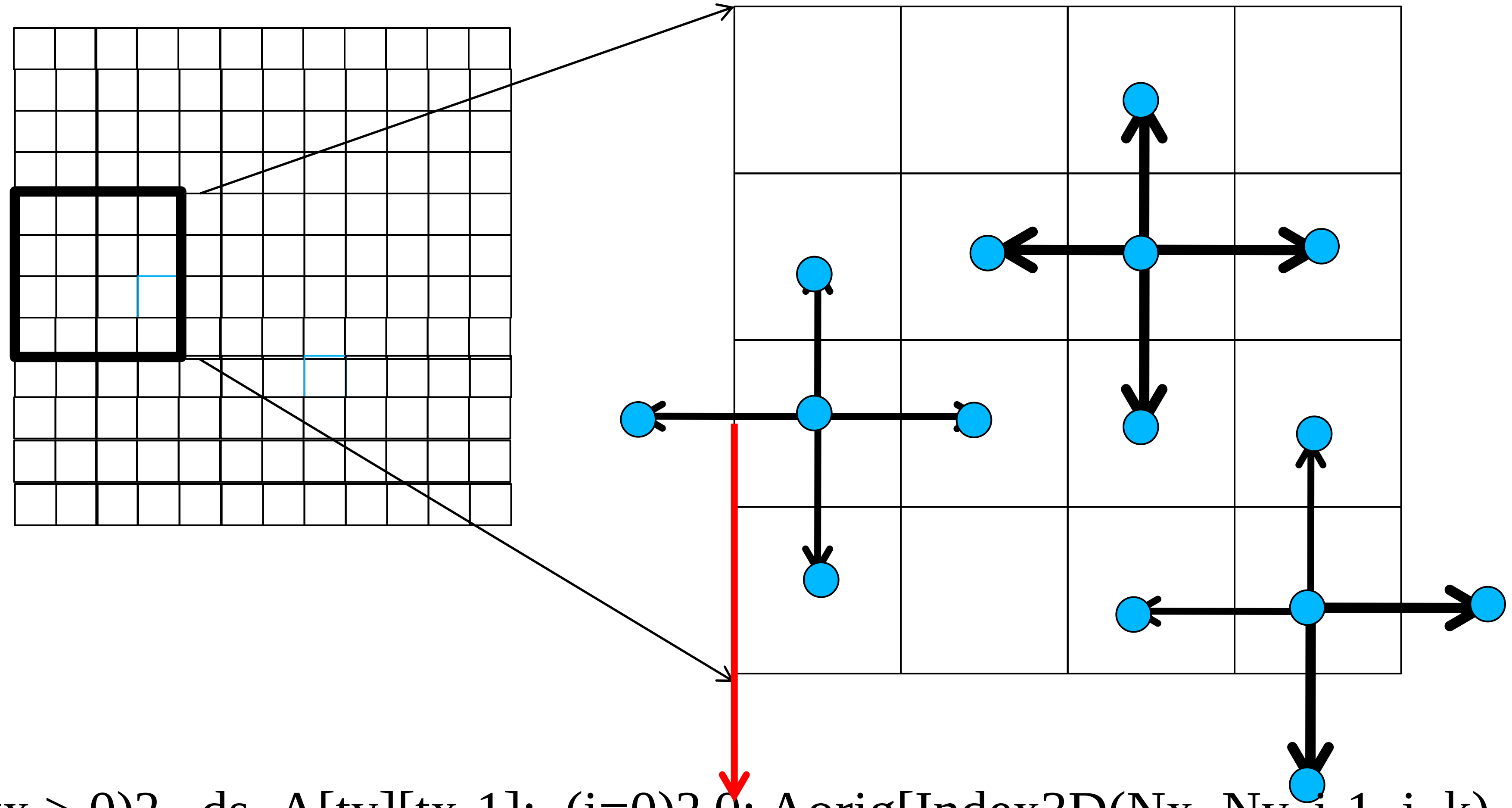
```
__shared__   float  ds_A[TILE_SIZE][TILE_SIZE];
float bottom = Aorig[Index3D(Nx, Ny, i, j, 0)];
float current = Aorig[Index3D(Nx, Ny, i, j, 1)];
ds_A[threadIdx.y][threadIdx.x] = current;
float top       = Aorig[Index3D(Nx, Ny, i, j, 2)];

for (k=1, k < Nz-1, k++)  {
     Anext[Index3D(Nx,Ny,i,j,0)] = bottom + top +





          __synchthreads();
          bottom = current;     ds_A[ty][tx] = current; current = top;
          __synchthreads()
}
```

Not all neighbors will be in the shared memory



(tx > 0)?   ds_A[ty][tx-1]: (i=0)? 0: Aorig[Index3D(Nx, Ny, i-1, j, k)

Not all neighbors will be in the shared memory



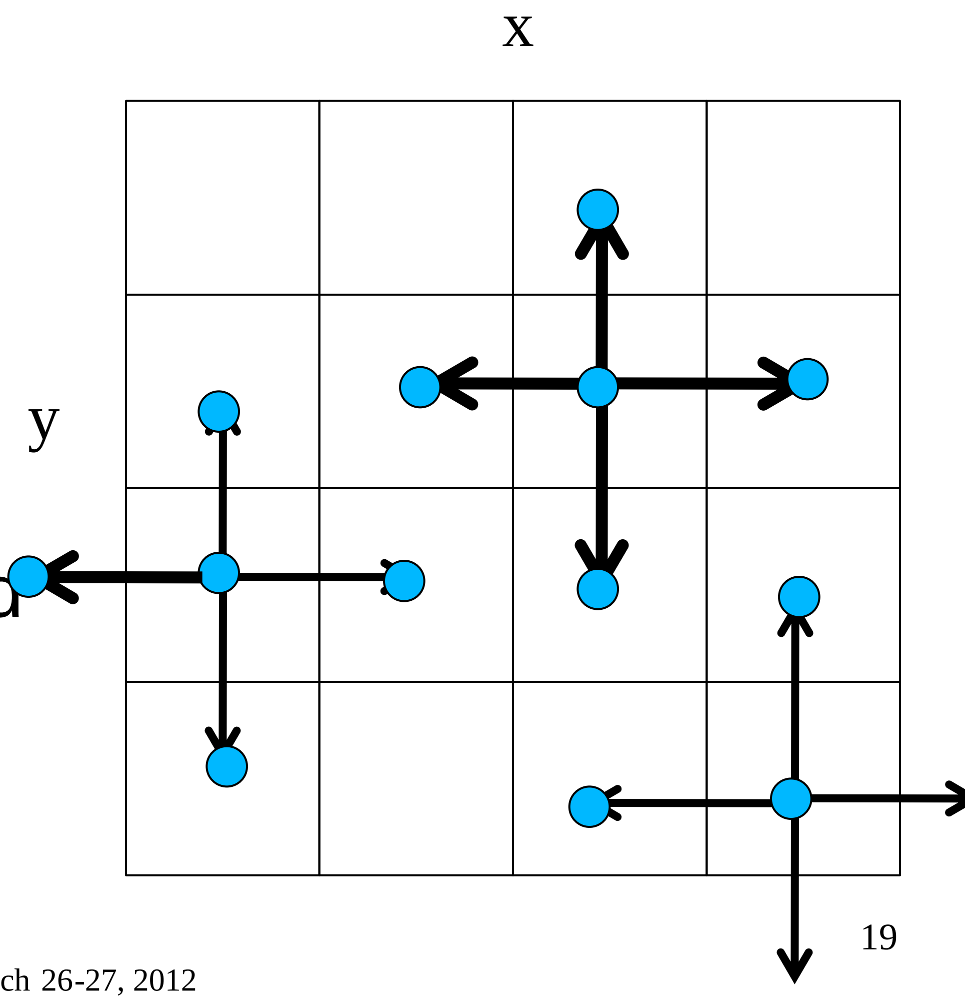
(tx > 0)?   ds_A[ty][tx-1]: (i=0)? 0: Aorig[Index3D(Nx, Ny, i-1, j, k)

Sample Coarsened Kernel Code

```
__shared__   float  ds_A[TILE_SIZE][TILE_SIZE];
float bottom = Aorig[Index3D(Nx, Ny, i, j, 0)];
float current = Aorig[Index3D(Nx, Ny, i, j, 1)];
ds_A[threadIdx.y][threadIdx.x] = current;
float top       = Aorig[Index3D(Nx, Ny, i, j, 2)];

for (k=1, k < Nz-1, k++)  {
      Anext[Index3D(Nx,Ny,i,j,0)] = bottom + top +
        (tx > 0)?   ds_A[ty][tx-1]:  (i=0)? 0: Aorig[Index3D(Nx, Ny, i-1, j, k) +
        (tx < dx)? ds_A[ty][tx+1]: (i=Nx-1)? 0: Aorig[Index3D(Nx, Ny, i+1, j, k) +
        (ty > 0)?   ds_A[ty-1][tx]:  (j=0)? 0: Aorig[Index3D(Nx, Ny, i, j-1, k) +
        (ty < dx)? ds_A[ty+1][tx]: (j=Ny-1)? 0: Aorig[Index3D(Nx, Ny, i, j+1, k) –
                        6 * current / (fac * fac);
      bottom = current;     ds_A[ty][tx] = current;
      current = top;
}
```

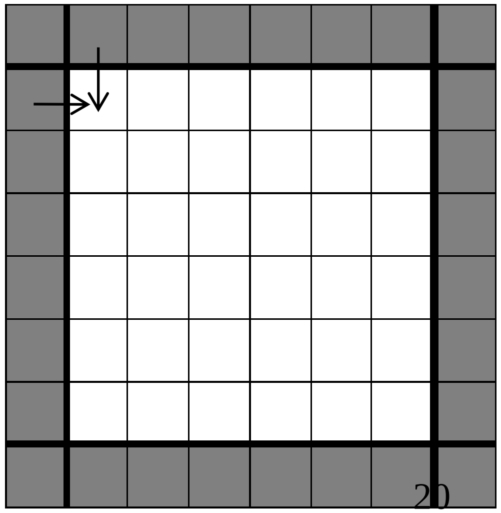
Cross-Thread Data Reuse

- 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 (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 deal with halo data
  - Needed to calculate edge elements of the column
  - For each 2D nxm tile slice to be computed, we need to load (n+2)x(m+2) - 4 inputs..



Loading halo elements 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 (16+2)*(8+2) -4 =18*10=176 elements
  - In coarsened code, each output element needs 5 loads from global memory, a total of 5*128=640 loads
  - The total ratio of improvement is 640/176 = 3.6, rather than 5 times
  - The value of n and m are limited by the amount of registers and shared memory in each SM

Another Approach

- One can load all halo cells into the shared memory as well
- This would reduce the control divergence and non-coalesced accesses at the calculation of Anext.
- However, loading the halo cell will involve control divergence and non-coalesced accesses.
- The net effect is usually not better (left as homework).

In Fermi

- It is actually better not to load halo elements into shared memory.
  - The halo cells are most likely already loaded by neighboring thread blocks as their core cells.
- The halo cells are often available in the L2 cache anyway

ANY MORE QUESTIONS?