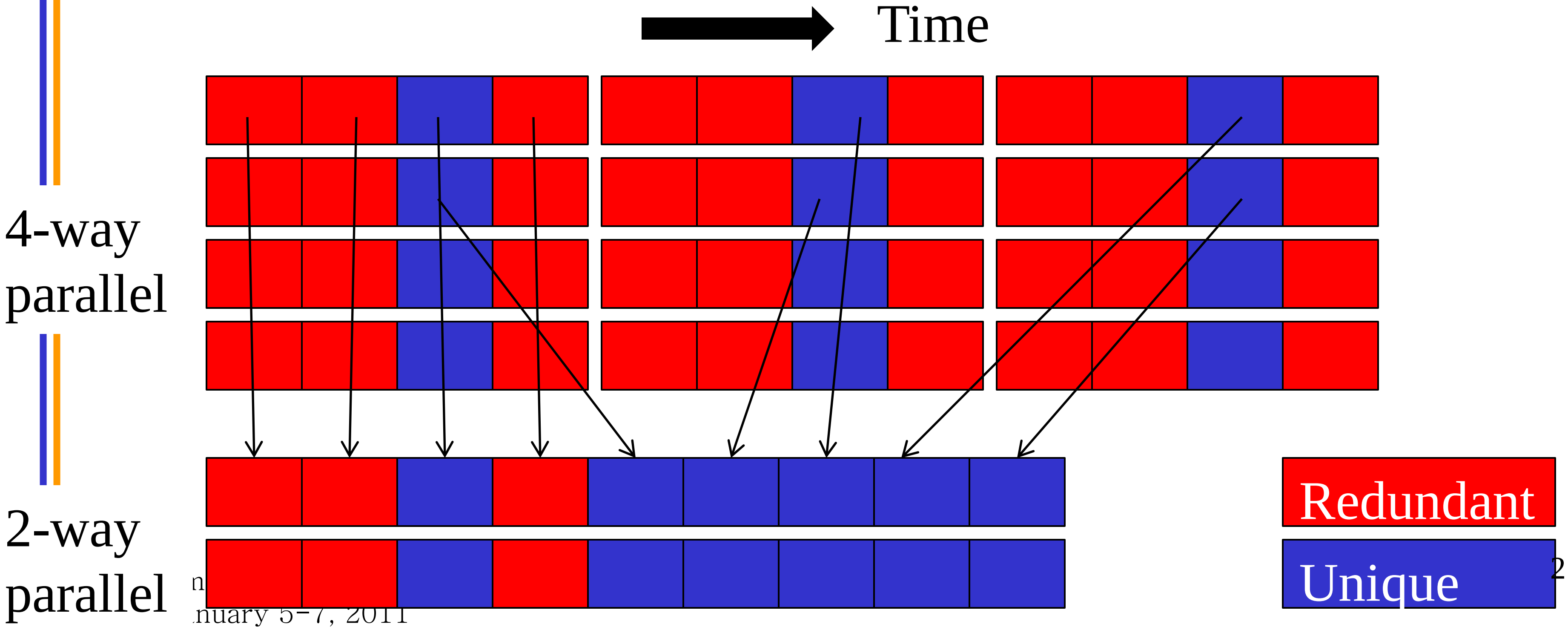
PASI Summer School

Advanced Algorithmic Techniques for GPUs

Lecture 4: Thread Coarsening and more on Tiling/Blocking

Thread Coarsening

- Parallel execution sometime 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 for calculating all 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

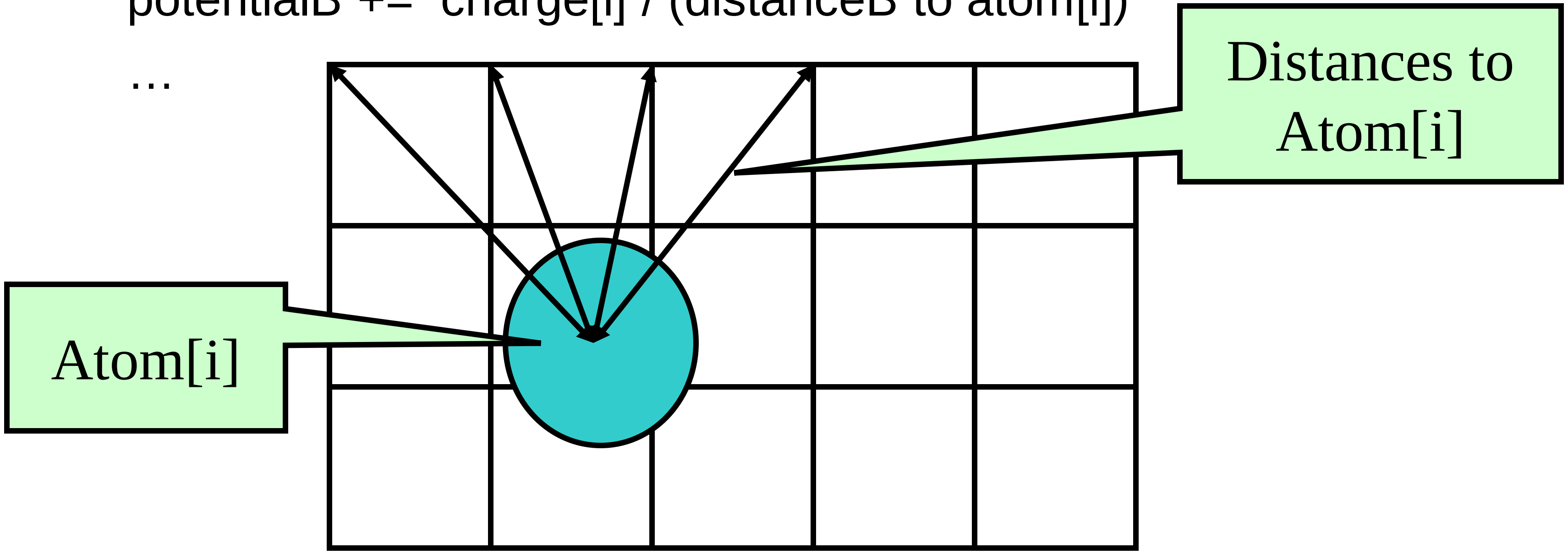
DCS Kernel with Register Tiling

- Add each atom's contribution to several lattice points at a time, where distances only differ in one (x) component:

potentialA += charge[i] / (distanceA to atom[i])

potentialB += charge[i] / (distanceB to atom[i])

…



DCS Coarsened Kernel Structure

- Example kernel processes up to 4 lattice points at a time in the inner loop
- Subsequent lattice points computed by each thread are offset by a half-warp to guarantee coalesced memory accesses
- Loads and increments 4 potential map lattice points from global memory at completion of the summation, mitigating register consumption

Coarsened Kernel Inner Loop Outline

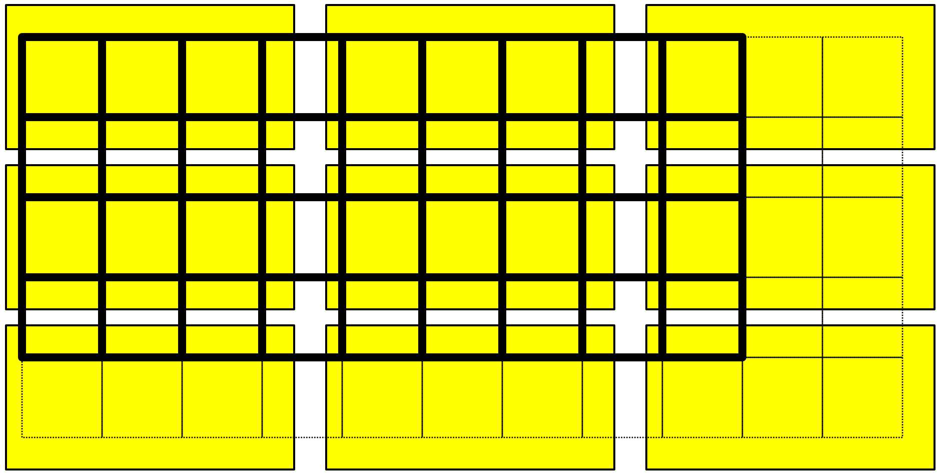
```
for (atomid=0; atomid<numatoms; atomid++) {
  float dy = coory - atominfo[atomid].y;
  float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
  float dx1 = coorx1 - atominfo[atomid].x;
  float dx2 = coorx2 - atominfo[atomid].x;
  float dx3 = coorx3 - atominfo[atomid].x;
  float dx4 = coorx4 - atominfo[atomid].x;
  energyvalx1 += atominfo[atomid].w * (1.0f / sqrtf(dx1*dx1 + dysqpdzsq));
  energyvalx2 += atominfo[atomid].w * (1.0f / sqrtf(dx2*dx2 + dysqpdzsq));
  energyvalx3 += atominfo[atomid].w * (1.0f / sqrtf(dx3*dx3 + dysqpdzsq));
  energyvalx4 += atominfo[atomid].w * (1.0f / sqrtf(dx4*dx4 + dysqpdzsq));
}
…
```

More Comments on Coarsened Kernel

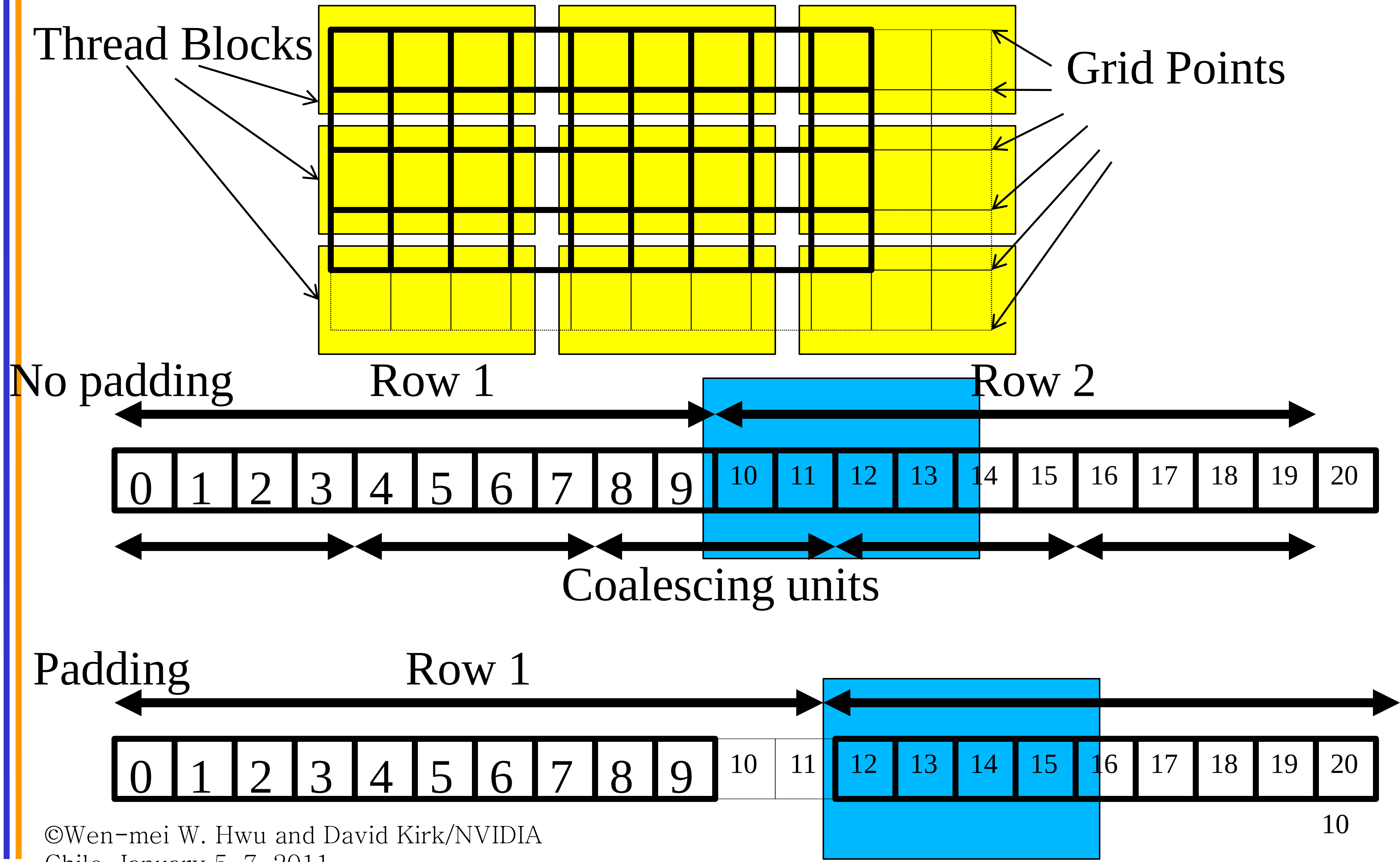
- Pros:
  - We can reduce the number of loads by reusing atom coordinate values for multiple voxels, by storing in regs
  - By merging multiple points into each thread, we can compute dy^2+dz^2 once and use it multiple times, much like the fast CPU version of the code
  - A good balance between efficiency, locality and parallelism
- Cons:
  - Uses more registers, one of several limited resources
  - Increases effective tile size, or decreases thread count in a block, though not a problem at this level

Basic DCS Kernel

- Each thread calculates value for one grid point
- A small toy example. ASSUME
  - Each thread block consists of 8 threads
  - Each warp consists of 4 threads, 16-byte coalescing
  - a 10X5 potential grid map
  - Padding - 2 points in x dim and 1 point in y dim
    - No boundary tests
    - Coalescing
  - 44% overhead



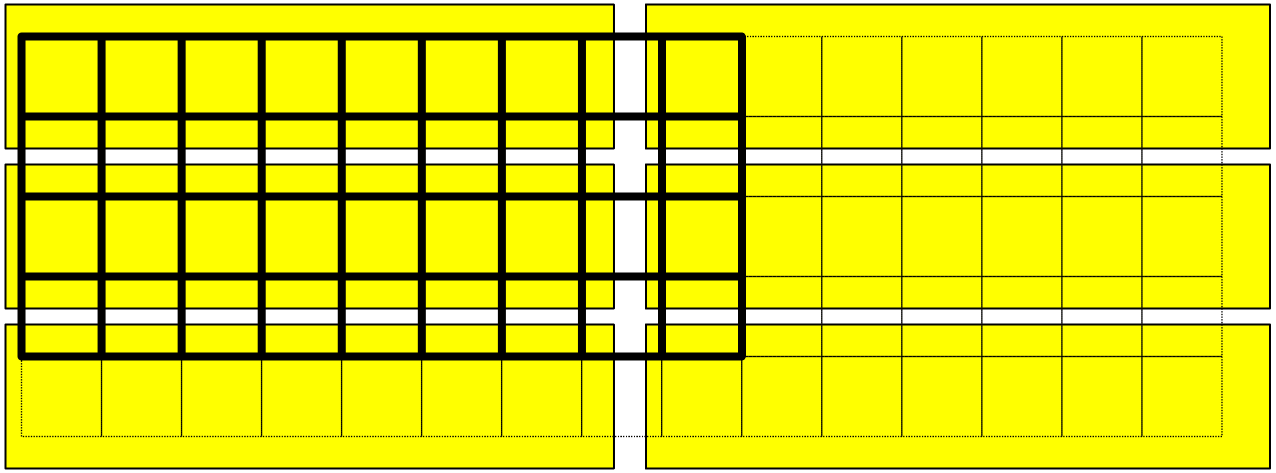
DCS Memory Coalescing



Coarsened DCS Kernel

- Merge threads to calculate more than one lattice point per thread, resulting in larger computational tiles:
  - Thread count per block may need to be be decreased to reduce computational tile size as per thread work is increased
  - Otherwise, tile size gets bigger as threads do more than one lattice point evaluation, resulting on a significant increase in padding and wasted computations at edges

Simple Thread Coarsening



- Each thread processes two grid points
  - Increased padding overhead (92%)
  - Classic quantization effect
- Can be mitigated by reducing the number of threads in each block, X-dim in particular

A Simple Quiz

- Assume
  - 1000X1000 energy grid
  - 16X16 thread block
  - 64-byte coalesing units

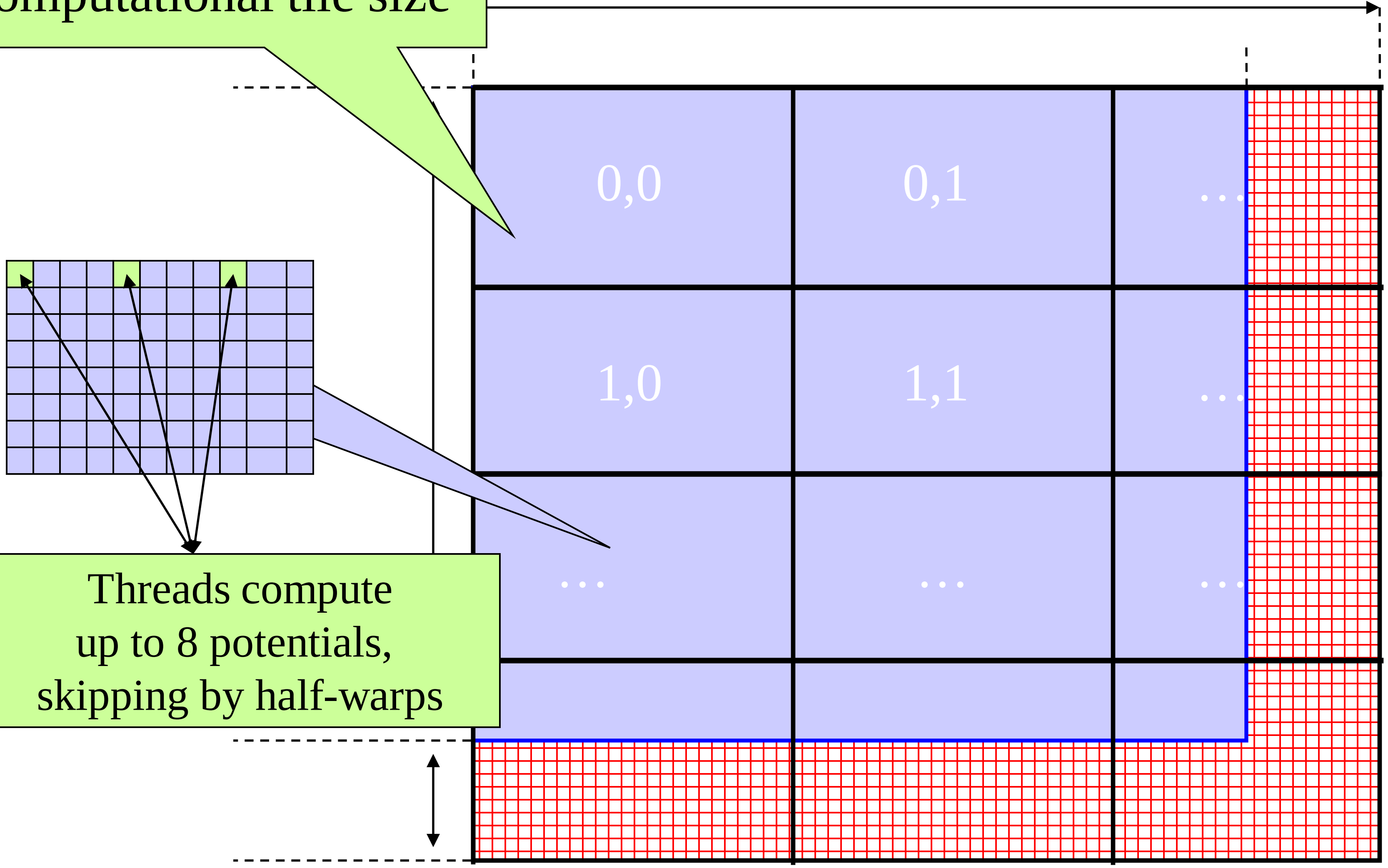
- What is the padding overhead if each thread processes one grid point?

- What is the padding overhead if each thread processes four grid points?

DCS CUDA Block/Grid Decomposition

(Coarsened, coalesced)

Coarsening increases computational tile size

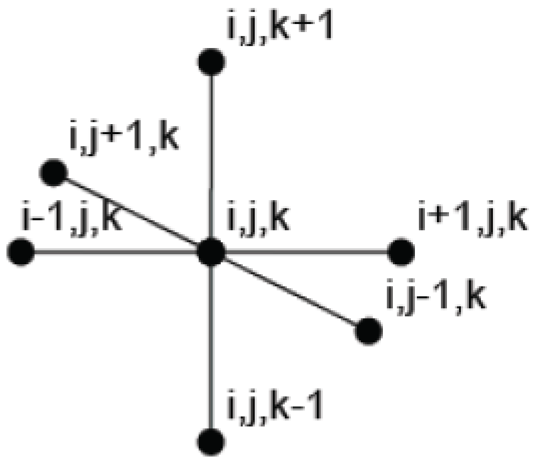


Threads compute up to 8 potentials, skipping by half-warps

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

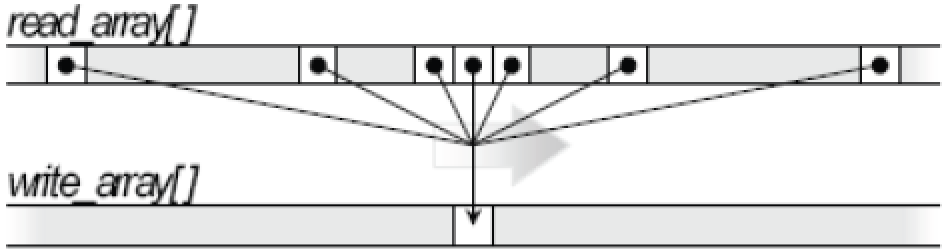
- High parallelism: Conceptually, all points in the grid can be updated in parallel.
- Each computation performs a global sweep through the data structure.
- Low computational intensity: High memory traffic for very few computations.
- Base case: one thread calculates one point
- Challenge: Exploit parallelism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

Memory Access Details

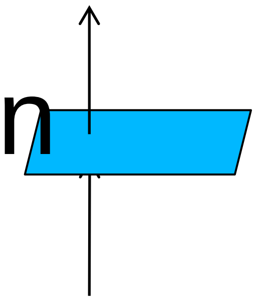
- General Equation:

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

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

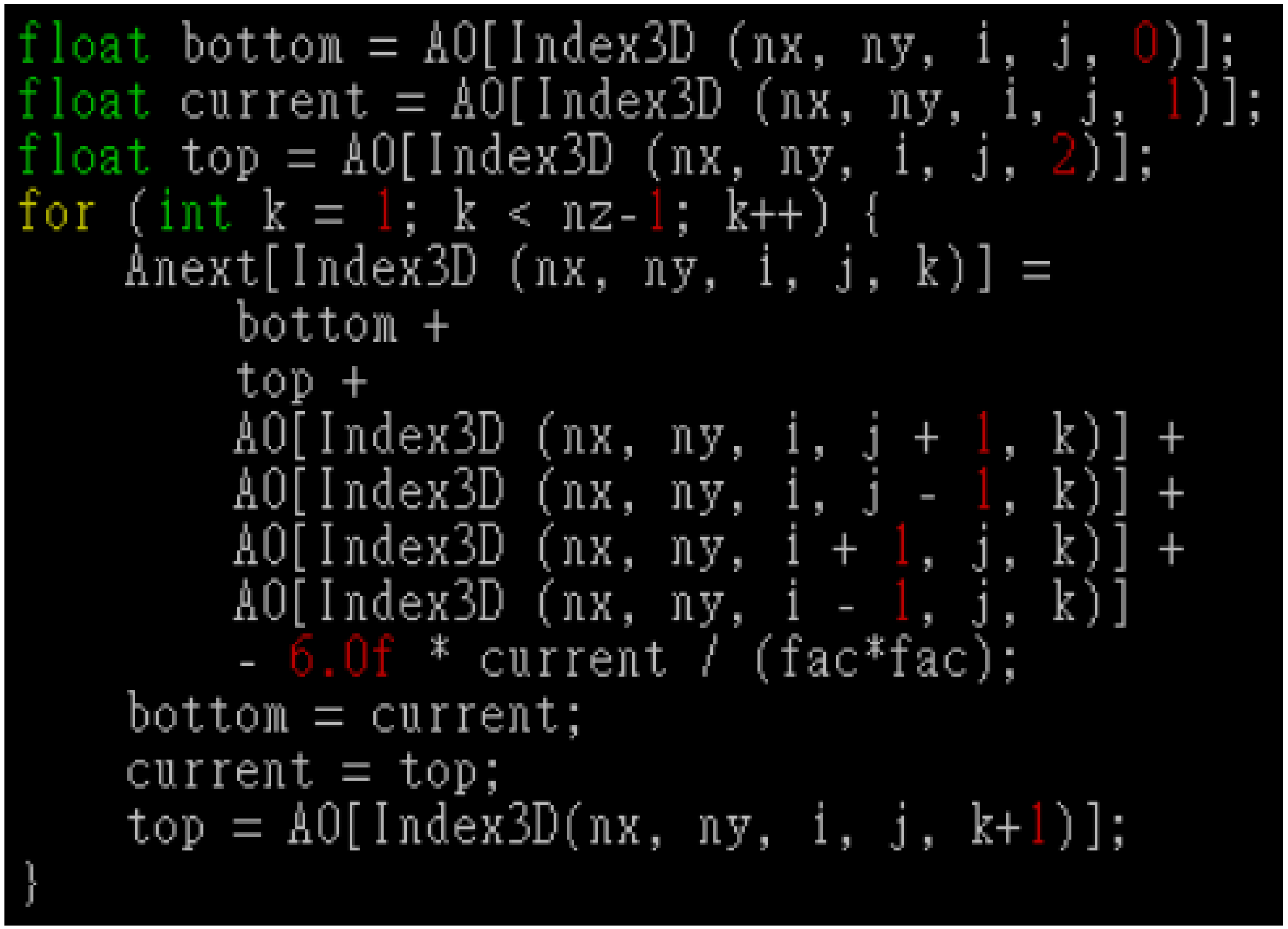


Coarsened implementation



- 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 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 Coarsened Kernel Code



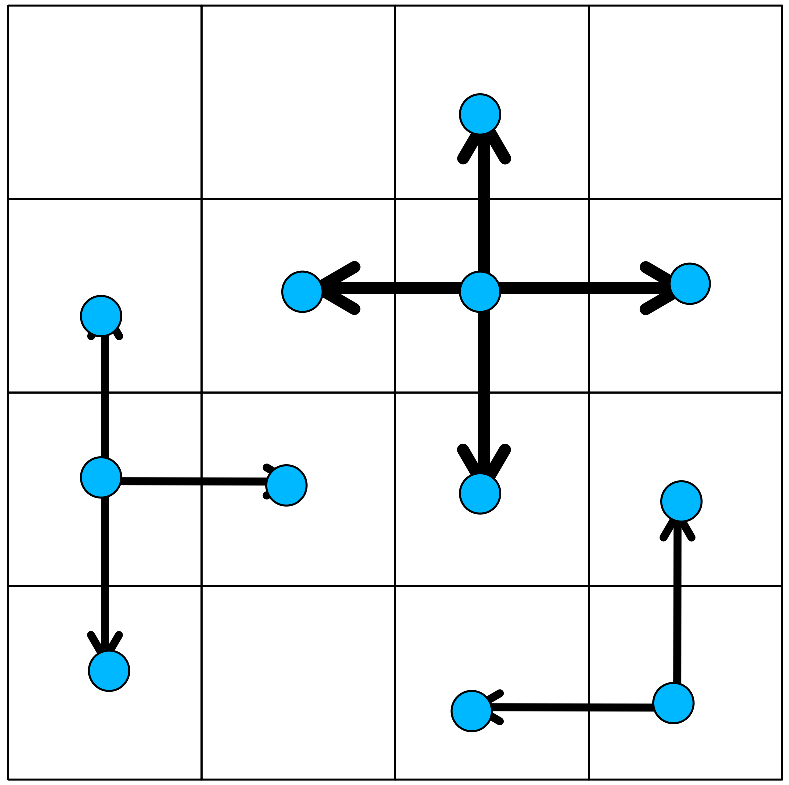
```
float bottom = A0[Index3D (nx, ny, i, j, 0)];
float current = A0[Index3D (nx, ny, i, j, 1)];
float top = A0[Index3D (nx, ny, i, j, 2)];
for (int k = 1; k < nz-1; k++) {
    Anext[Index3D (nx, ny, i, j, k)] =
        bottom +
        top +
        A0[Index3D (nx, ny, i, j + 1, k)] +
        A0[Index3D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

Loads in the 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

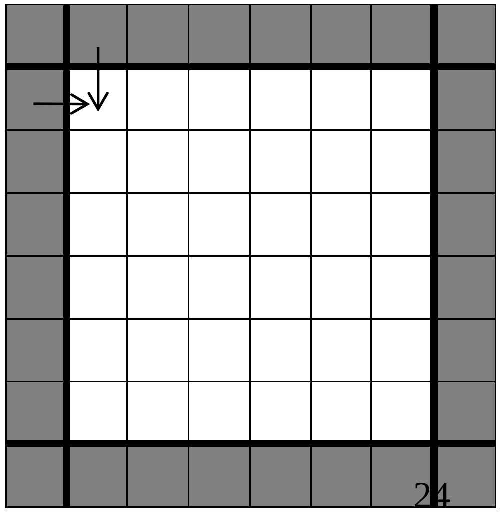


Improving Locality: 2D Tiling

- 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 deal with halo data
  - 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..



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) =18*10=180 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 640/180 = 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

In Fermi

- It is often better not to load halo elements into shared memory.
- Rather, just put in a test and load the halo to from the global memory for the boundary elements

ANY MORE QUESTIONS?