ECE 508
Manycore Parallel Algorithms

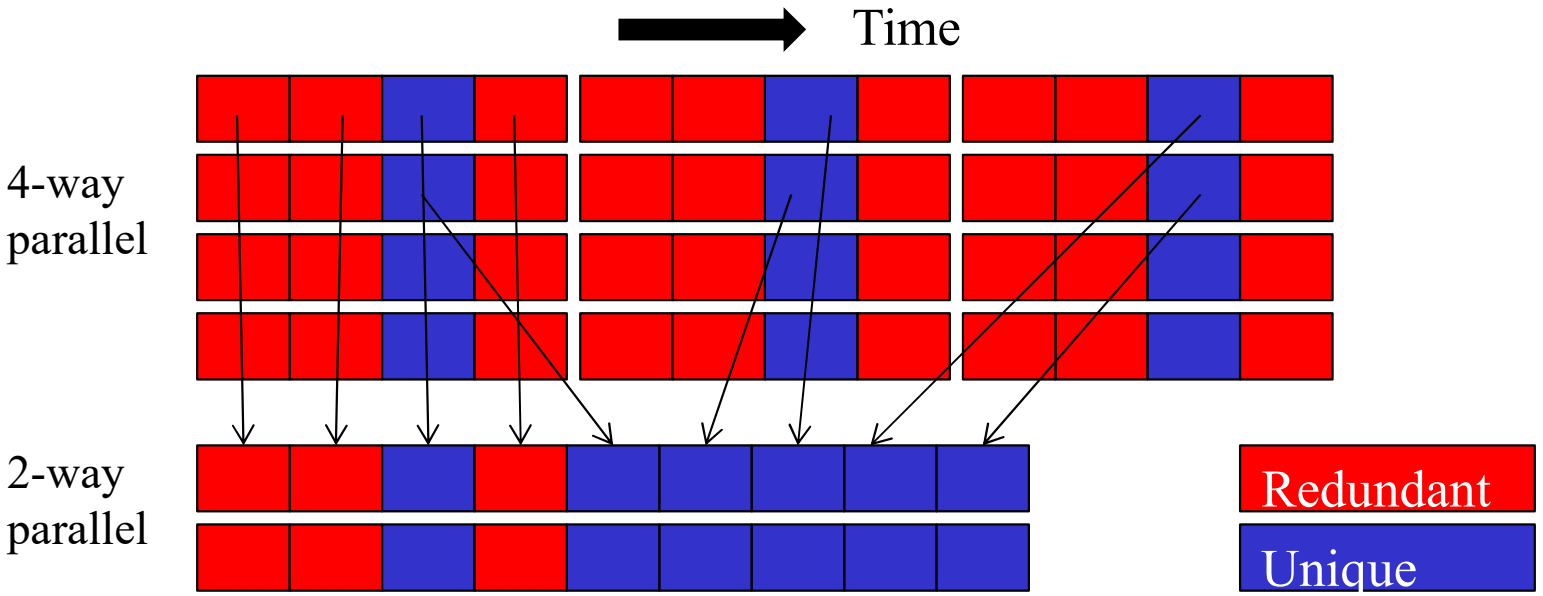
Lecture 3: Thread Coarsening and Register Tiling

Objective

- To learn thread coarsening and register tiling,
  - two important, closely related techniques for trading reduced parallelism in return for increased memory and compute efficiency.
  - These are especially important when an application is memory-bound or compute-bound.

Merge Threads to Reduce Redundancy

- Parallel execution implies redundant work.
- **Merging** several **threads allows** re-use of results, **reducing redundant work**.



Thread Coarsening: More Registers, Less Parallelism

(Context: parallelization over outputs.)

- Instead of one, a **thread calculates several outputs**.
  - **Save** thread **index calculations into registers**.
  - Use the registers for each output element.
- Coarsened **kernel requires more registers**,
  - which may limit threads executed in parallel.
  - Increased efficiency may outweigh reduced parallelism.
  - See Kirk & Hwu, 3rd ed., Section 5.5.

Coarsening Can Limit Performance in Several Ways

What drawbacks can arise from reduction of thread count and increased register use?

- Not enough thread blocks to keep SMs busy.
- Not enough thread blocks to balance across SMs.
- Not enough thread blocks / threads to hide latency.

Provides a Tuning Method for GPU Codes

Parallelization granularity is a tuning knob.

- Many choices possible between two extremes.
  - One extreme: each thread computes one output.
  - The other: one thread computes all outputs.
- Getting **tradeoff** right **not too hard for one data set on one GPU**.
- Harder to consider variable data size and changing GPU parameters over generations.

Use Two Examples to Illustrate Techniques

- To illustrate the techniques,
  we examine two case studies.
  - First, the **DCS Gather kernel**.
  - Second, a **stencil computation**.

Coarsening Produces Larger Computational Tiles

- Starting with Gather DCS kernel,
  - merge threads so that **each thread**
  - **calculates several** energy **grid points**.

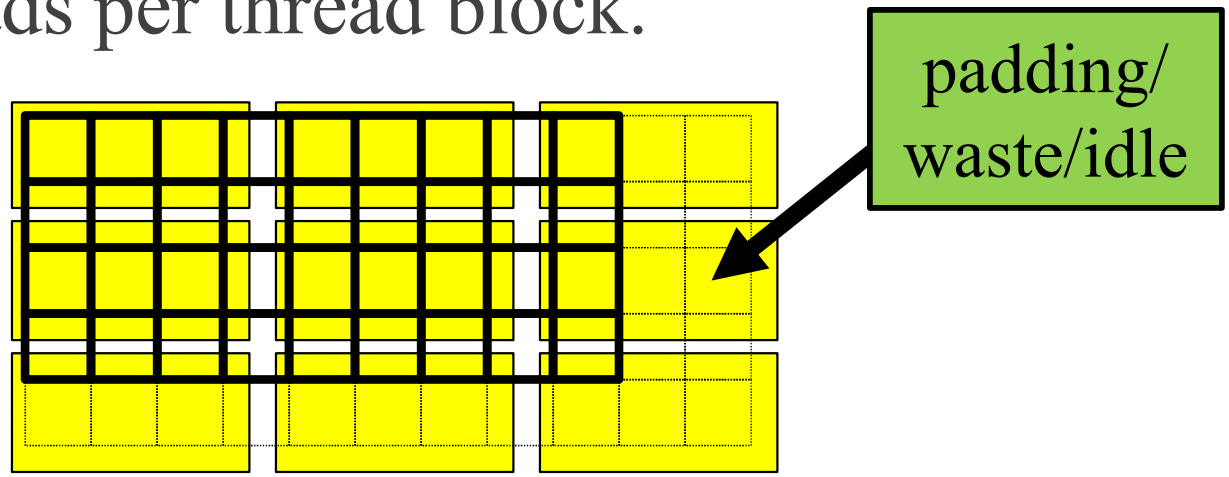
Doing so **increases computational tile size**
(size of computation performed by a thread block),

- which implies **more padding and wasted computation** at boundaries of output array.
- **May need to reduce thread count per block** to avoid this issue.

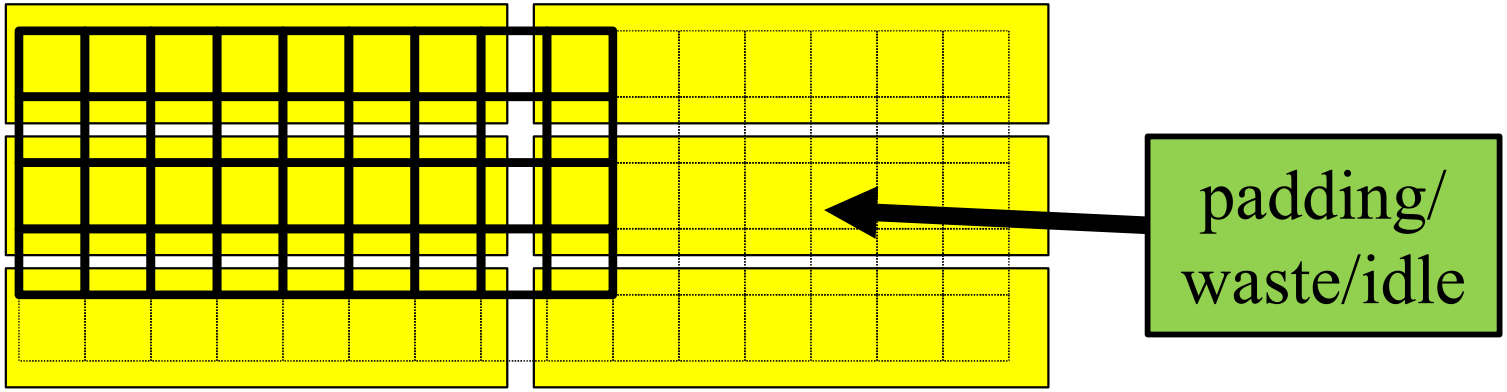
Gather DCS on a Toy Example

Let's **consider an example**:

- **one output per thread** (orig. DCS Gather kernel),
- a **10×5** potential grid map, and
- **2×4 = 8** threads per thread block.

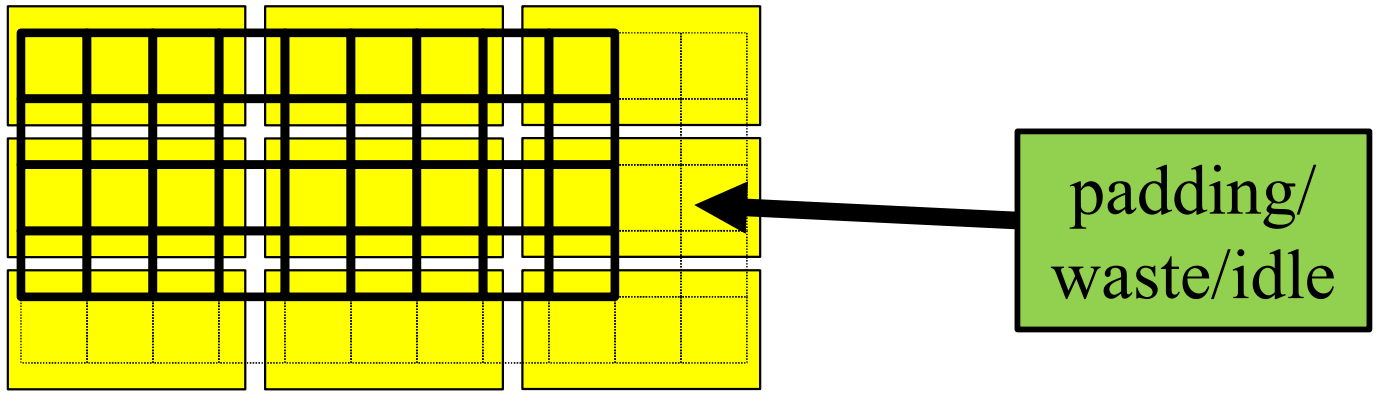


Coarsening Increases Number of Idle Threads



- Now **coarsen to two grid points** per thread (in X dim.).
  - Note **increased thread idling** at the boundaries,
  - a classic quantization effect.
- **Mitigate by reducing** number of **threads per block**.

Reducing Blocks per Thread Avoids Increased Waste



- **Reduce thread block size** from **2×4** to **2×2**.
  - **Data tile** size **same as** that of **original kernel**.
  - Avoids increasing number of idle threads.

A Simple Quiz (Just an Example!)

- Assume
  - **1000×1000** energy grid
  - **16×16** thread blocks

1. **How many thread blocks are needed without thread coarsening?**

2. **How many thread blocks are needed if each thread computes four grid points in the x-dimension?**

Quiz Answers

1. **One grid point per thread**:
   - To cover **1000** points,
     - we need $\lceil 1000/16 \rceil = 63$ blocks
     - in both X and Y dimensions.
   - We have **a total of 63 * 63 blocks**.
2. **Four grid points per thread**:
   - To cover **1000** points in the X dimension, we need $\lceil 1000/64 \rceil = 16$ blocks.
   - Number of blocks in Y dimension remains **63**.
   - We have **a total of 16 * 63 blocks**.

Register Tiling Possible after Coarsening

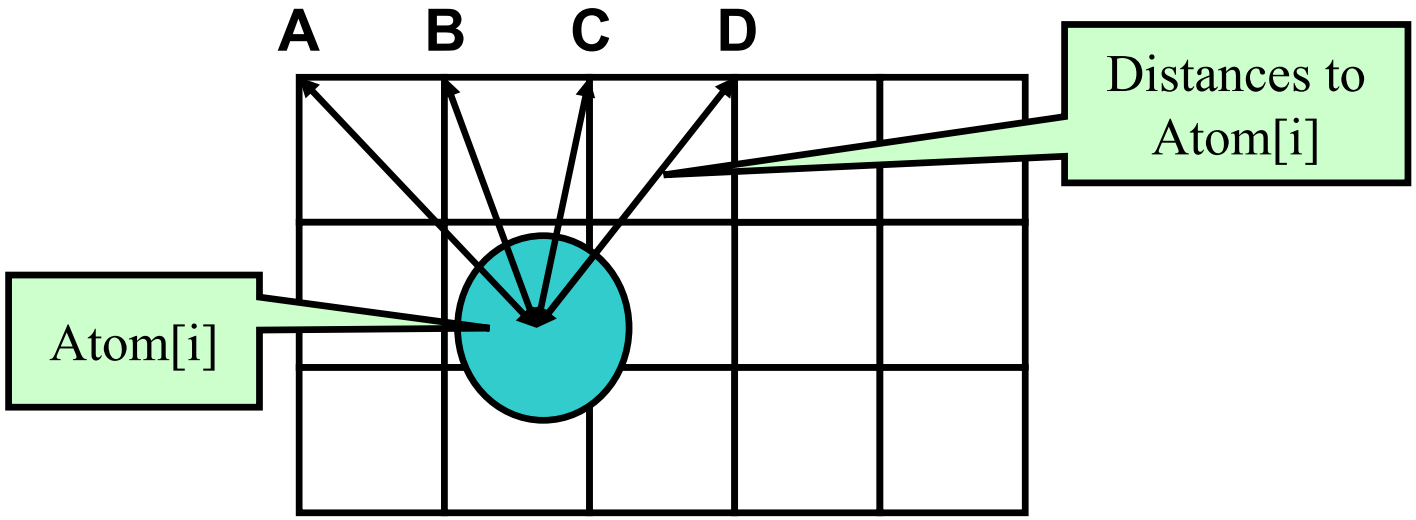
Now for **the benefit: register tiling**.

- Recall the **properties of GPU registers**:
  - **extremely fast** (short latency), and
  - **extremely high throughput**: register file allows access to multiple registers per thread per cycle,
  - but **private to each thread**: cannot use to share computation or loaded memory data, and
  - named directly: any "arrays" must be **fully expanded to constant indices** (and loops unrolled).

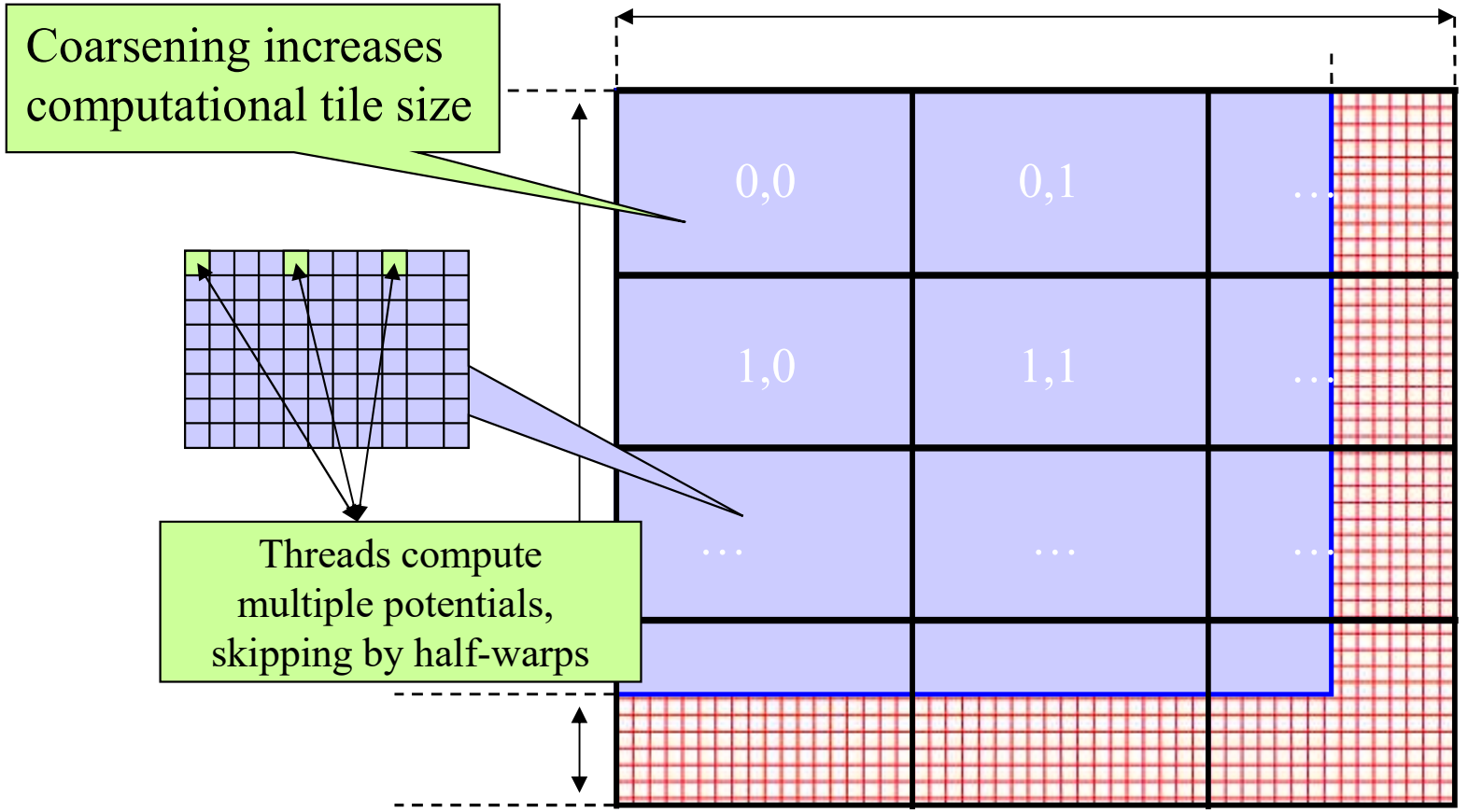
With thread coarsening, computation from merged threads can be shared through registers!

Keep All But DX in Registers

- **Thread adds** each atom's contribution **to several lattice points**.
  - **Distances differ only in X component**.
  - `potentialA += charge[i] / (distanceA to atom[i])`
  - `potentialB += charge[i] / (distanceB to atom[i])`
  - …



A Thread's Grid Points are NOT Adjacent!



Properties of the Coarsened Kernel

- Coarsened kernel **processes four grid points** in inner loop
- Each **thread's grid points**
  - **offset by** at least half-warp (**16**) **elements**
  - **to** guarantee **coalesced memory accesses**.
- As before,
  - **accumulates** contributions **into registers** (now using more registers), and
  - **updates** global **memory after** inner **loop**.

Example of Coarsened DCS Gather Inner Loop

```
for (int n = 0; n < atomarrdim; n += 4) {
    float atomx     = atoms[n + 0];              // X coordinate
    float dy        = coory - atoms[n + 1];      // Y coordinate
    float dysqpdzsq = (dy * dy) + atoms[n + 2]; // dz^2 passed in Z
    float charge    = atoms[n + 3];              // charge
```

Start by reading and processing atom coordinates and charge.

Fewer memory reads and computation per atom.

```
}
```

Example of Coarsened DCS Gather Inner Loop

```
for (int n = 0; n < atomarrdim; n += 4) {
    float atomx     = atoms[n + 0];              // X coordinate
    float dy        = coory - atoms[n + 1];      // Y coordinate
    float dysqpdzsq = (dy * dy) + atoms[n + 2]; // dz^2 passed in Z
    float charge    = atoms[n + 3];              // charge
    float dx1 = coorx1 - atomx;
    float dx2 = coorx2 - atomx;
    float dx3 = coorx3 - atomx;
    float dx4 = coorx4 - atomx;
}
```

Next, compute X components of distance.

No additional memory accesses needed!

Example of Coarsened DCS Gather Inner Loop

```
for (int n = 0; n < atomarrdim; n += 4) {
    float atomx     = atoms[n + 0];                  // X coordinate
                                                  n +
                                                  oms[                    Z
    float dx2 = coorx2 - atomx;
    float dx3 = coorx3 - atomx;
    float dx4 = coorx4 - atomx;
    energyvalx1 += charge / sqrtf (dx1 * dx1 + dysqpdzsq);
    energyvalx2 += charge / sqrtf (dx2 * dx2 + dysqpdzsq);
    energyvalx3 += charge / sqrtf (dx3 * dx3 + dysqpdzsq);
    energyvalx4 += charge / sqrtf (dx4 * dx4 + dysqpdzsq);
}
```

Finally, accumulate contributions per grid point.

Again, no additional memory accesses needed, and minimal computation.

Reference Version: Coarsened DCS Gather Inner Loop

```
for (int n = 0; n < atomarrdim; n += 4) {
    float atomx     = atoms[n + 0];              // X coordinate
    float dy        = coory - atoms[n + 1];      // Y coordinate
    float dysqpdzsq = (dy * dy) + atoms[n + 2]; // dz^2 passed in Z
    float charge    = atoms[n + 3];              // charge
    float dx1 = coorx1 - atomx;
    float dx2 = coorx2 - atomx;
    float dx3 = coorx3 - atomx;
    float dx4 = coorx4 - atomx;
    energyvalx1 += charge / sqrtf (dx1 * dx1 + dysqpdzsq);
    energyvalx2 += charge / sqrtf (dx2 * dx2 + dysqpdzsq);
    energyvalx3 += charge / sqrtf (dx3 * dx3 + dysqpdzsq);
    energyvalx4 += charge / sqrtf (dx4 * dx4 + dysqpdzsq);
}
```


Pros and Cons of Coarsened DCS Gather Kernel

Pros

- **Reduces number of loads** by reusing atom coordinate and charge values.
- **Eliminates redundant computation**
  - such as reuse of dy^2 + dz^2,
  - much like the fast CPU version.
- **Good balance** between efficiency, locality, and parallelism.

Cons

- Uses **more registers**, a limited resource.
- **Increases tile size or decreases thread count** per block.

Can the Compiler Help? Maybe.

Hand-unrolling is ugly, error-prone, and an unnecessary challenge for compilers on platforms that do not require it.

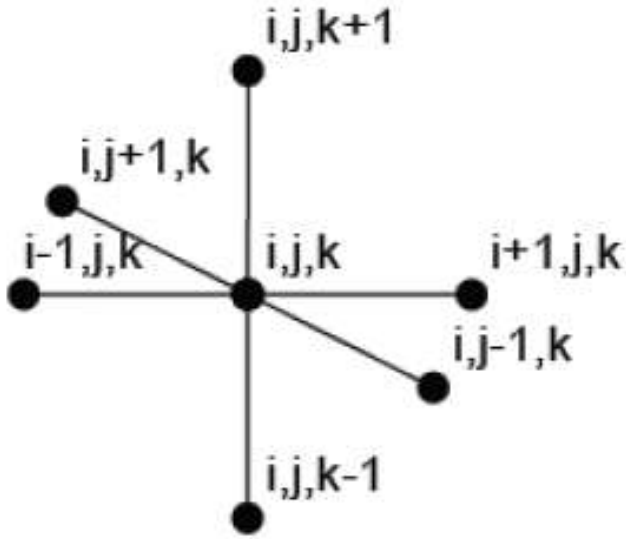
Can the CUDA compiler unroll loops and use register tiling?

- **Sometimes**,
  - for **small "arrays" of** local **primitive types** (int / float),
  - **provided** that fully unrolled loops **completely eliminate indirection in register names**.
- We'll show some examples in later lectures.

Second Example: a Stencil Computation

What is a stencil computation?

- A **computation based on neighbors in** a structured **grid**.
- In computational science,
  - **arises from Jacobi iterative method**
  - for solving partial differential equations.
  - During **each iteration**,
    - **each grid point** is **updated**
    - **with** a weighted **linear combination**
    - **of** a **subset of neighboring values** and itself.



Stencils Require Little Computation per Operand

Common aspects of stencil computations:

- **abundant parallelism**: all grid points updated in parallel;
- **memory intensive**: each update requires many points; and
- **little computation**: one multiply-and-add per value.

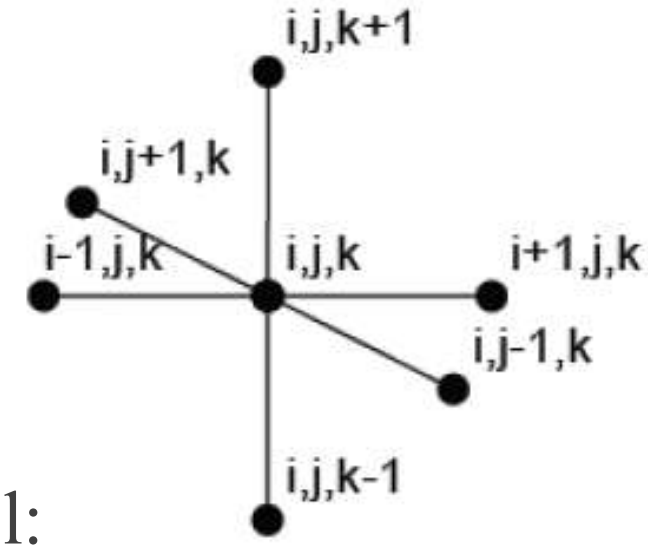
base case: output parallelism (one thread per grid point)

The challenge? To exploit parallelism without overusing memory bandwidth.

Apply Optimizations from 408 and This Lecture

We **apply several optimizations**:

- improving **locality** and **data reuse**;
- 2D **tiling** in **shared memory**; and
- **coarsening** and **register tiling**.



For simplicity,

- focus on an **order 1, 7-point** stencil:
- 1 neighbor in six directions + grid point itself.

Real PDE solvers often use more neighbors;

for example, order **4** (on each side), or **25**-point.

Simplify Further for Our Example (and Lab 2!)

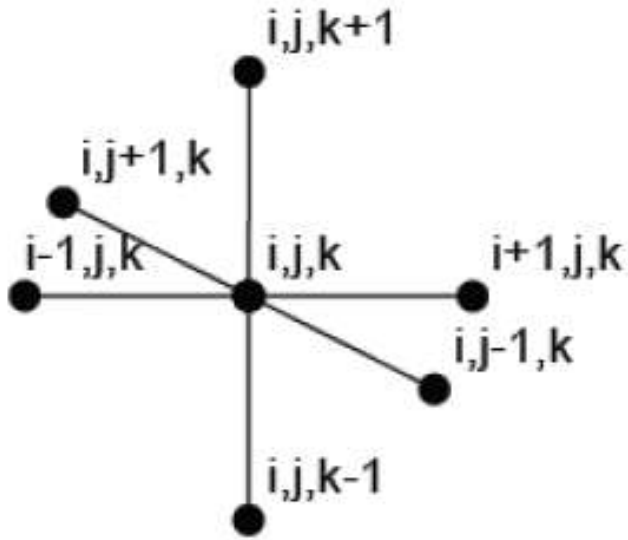
For a **symmetric** and **homogeneous 7-point stencil**, we have

```
out(i, j, k) = C0 * in(i, j, k) +
   C1 * (in(i-1, j, k) + in(i, j-1, k) + in(i, j, k-1) +
         in(i+1, j, k) + in(i, j+1, k) + in(i, j, k+1) )
```

Separating read (**in**) and write (**out**) arrays avoids read/write dependences.

Again for simplicity, we use

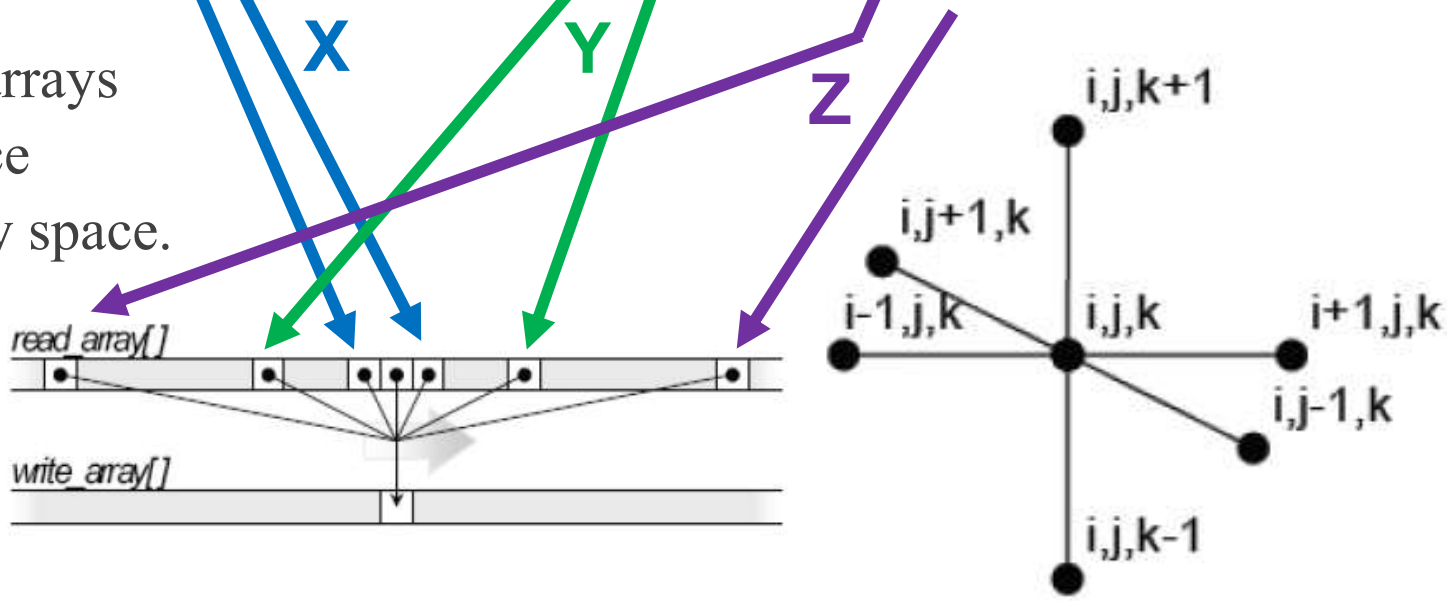
$C_0 = -6$ and $C_1 = 1$.



3D Arrays Mapped as Z-Major, Y Next

```
out(i, j, k) = C0 * in(i, j, k) +
    C1 * (in(i-1, j, k) + in(i, j-1, k) + in(i, j, k-1) +
          in(i+1, j, k) + in(i, j+1, k) + in(i, j, k+1) )
```

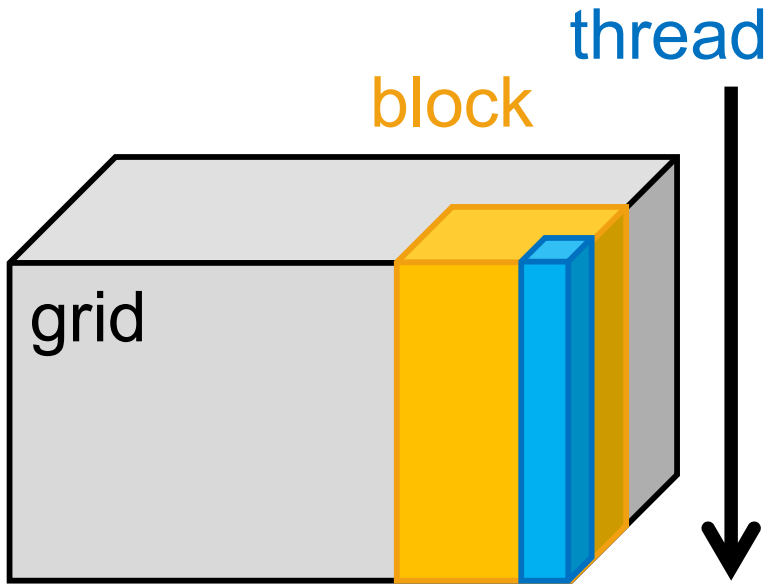
Mapping of arrays from 3D space to linear array space.



Coarsen Each Thread to Compute a Pencil in Z

First, apply **thread coarsening**.

- **Coarsen** across **entire grid in Z** dimension.
- Each **thread computes**
  - a one-element thin **pencil**
  - **along** the **Z dimension**.
- Each **thread block computes**
  - a **right cuboid** (right rectilinear prism)
  - **along** the **Z dimension**.



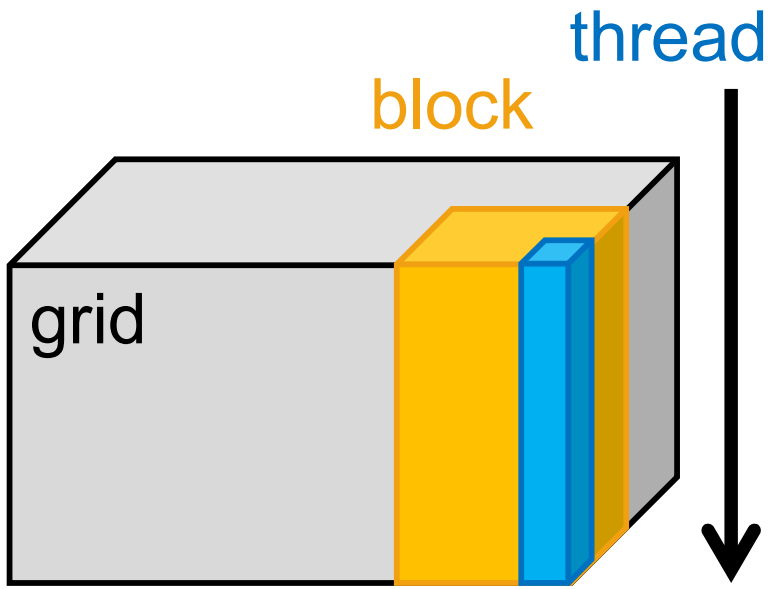
Thread coarsening speedup: 1.21 ×

Coarsening Enables Data Reuse in Z Dimension

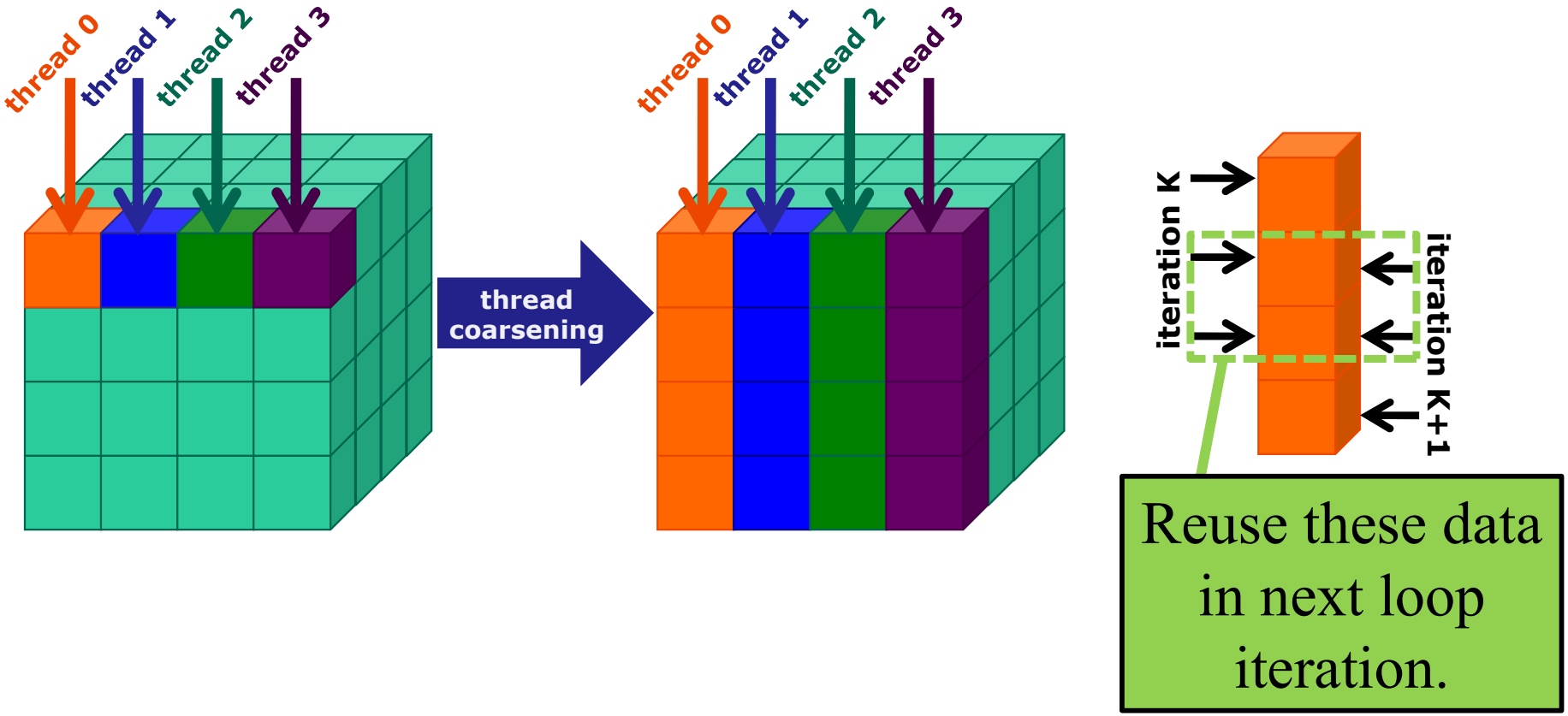
So far, threads access memory independently, without reuse:

- **read 7 elements** from **`in`**, then
- **write one value** to **`out`**,
- all **using global memory**.

Next, **let's reuse inputs from** the **Z dimension** (within a thread).



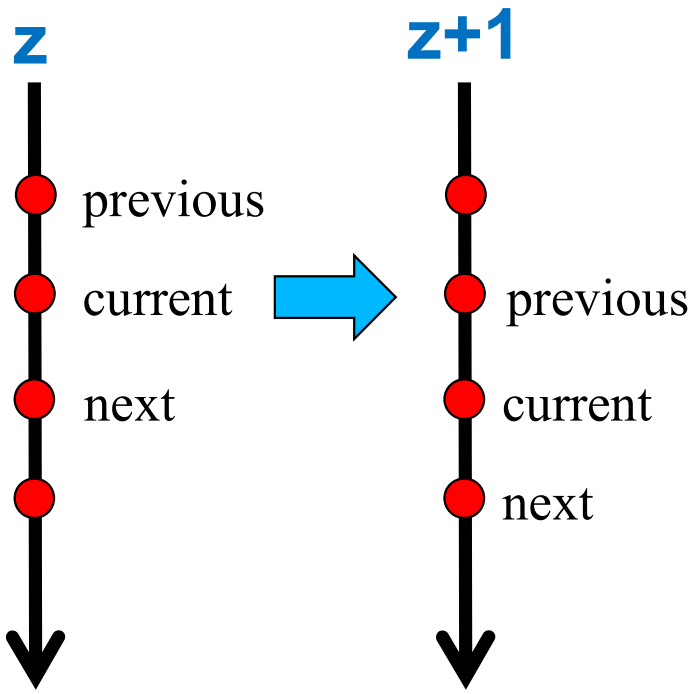
Thread Coarsening Enables Reuse with Register Tiling



Register Tiling Applied to a Z Pencil

Register tiling the Z dimension:

- **reuse data** along the Z dimension
- when moving **from z to z+1**.
  - **`current`** input becomes **`previous`**,
  - **`next`** input becomes **`current`**, and
  - new **`next`** input loaded from memory.



Simplifying Assumptions for Lab 2 and Slides

- **`in`** (**`A0`**) **and** **`out`** (**`Anext`**) arrays
  - **padded with** one **extra slice of 0 elements**
  - on each end of X, Y, and Z dimensions.
- **`nx`**, **`ny`**, **`nz`**: **dimensions of** the **arrays supplied** to kernel
  - include the padded elements, so
  - **`nx`** is the number of elements to be computed plus 2.

- **Slides use macro definitions** **`_in`** and **`_out`** (based on **`nx`**, **`ny`**, **`nz`**) **to simplify index expressions**.

Define Access Macros, Perform Thread Index Calcs

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
```

Write macro definitions and compute X and Y coordinates.

Initialize Registers for Tiling in Z

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
float previous = _in(i, j, 0);
float current  = _in(i, j, 1);
float next     = _in(i, j, 2);
```

Initialize registers used for tiling.

Loop Over Output Pencil in Z

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
float previous = _in(i, j, 0);
float current  = _in(i, j, 1);
float next     = _in(i, j, 2);
for (k = 1; k < Nz - 1; k++) {



}
```

Loop over output elements in Z dimension.

Note that padding is skipped.

Compute and Store One Output

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
float previous = _in(i, j, 0);
float current  = _in(i, j, 1);
float next     = _in(i, j, 2);
for (k = 1; k < Nz - 1; k++) {
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {
    _out(i,j,k) = -6 * current + previous + next +
                  _in(i-1, j, k) + _in(i+1, j, k) +
                  _in(i, j-1, k) + _in(i, j+1, k);
  }

}
```

Boundary (padding) and excess X-Y threads are idle.

Compute and store one output.

Finally, Advance Tiled Registers

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
float previous = _in(i, j, 0);
float current  = _in(i, j, 1);
float next     = _in(i, j, 2);
for (k = 1; k < Nz - 1; k++) {
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {
    _out(i,j,k) = -6 * current + previous + next +
                  _in(i-1, j, k) + _in(i+1, j, k) +
                  _in(i, j-1, k) + _in(i, j+1, k);
  }
  previous = current; current = next; next = _in(i, j, k+2);
}
```

Advance tiled registers to next Z position.

Reference Version of Coarsened Stencil

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
float previous = _in(i, j, 0);
float current  = _in(i, j, 1);
float next     = _in(i, j, 2);
for (k = 1; k < Nz - 1; k++) {
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {
    _out(i,j,k) = -6 * current + previous + next +
                  _in(i-1, j, k) + _in(i+1, j, k) +
                  _in(i, j-1, k) + _in(i, j+1, k);
  }
  previous = current; current = next; next = _in(i, j, k+2);
}
```

Warning: includes a bug.

Reuse Reduces Global Memory Accesses by 25%

What does our register tiling accomplish?

- **Without tiling**,
  - each **thread loads 7 inputs**
  - **for each output** element written.
- **With** register **tiling** / data reuse,
  - each **thread loads 5 inputs***
  - **for each output** element written.

Savings: **25% reduction in global memory accesses**.

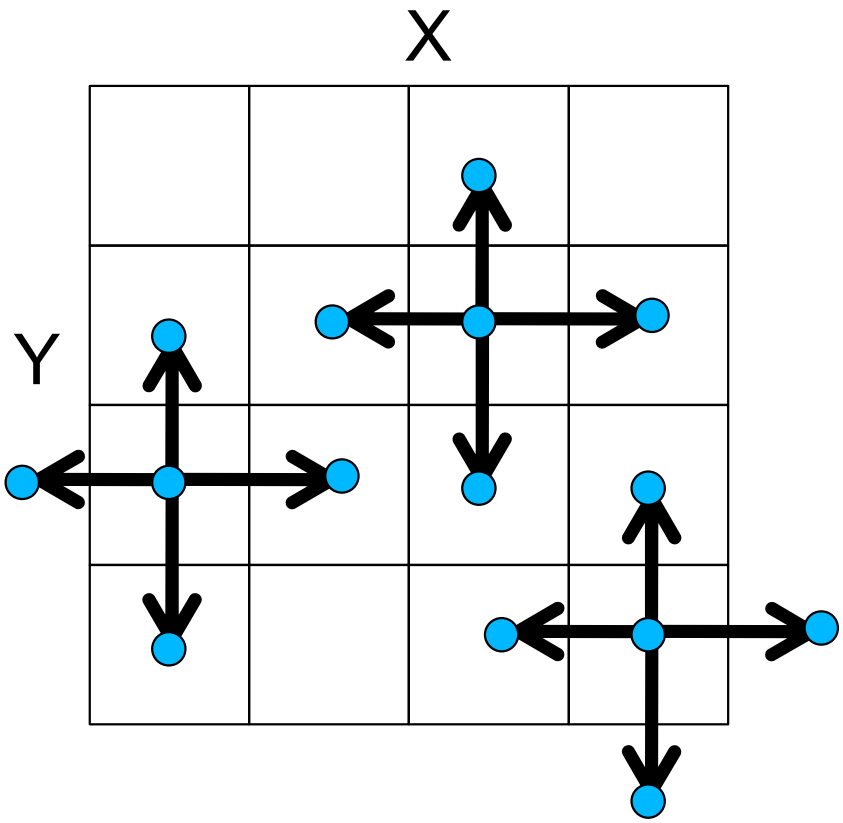
*Asymptotic limit for large **`Nz`**.

More Reuse Should Be Possible

Is more reuse possible?

How many outputs are affected by a single input (asymptotically, for a large grid)?

- 7: self, 4 planar neighbors, top and bottom neighbors
- (Surface, edge, and corner points used for fewer.)



Use Shared Memory to Enable Further Reuse

Consider threads in a block computing some Z value.

- Each thread's `current`
  - **needed by** (up to) **four neighbors**, but
  - **registers cannot be shared** between threads.

What can we do?

Make use of shared memory!

What else is necessary?

Synchronize between computation of Z values in the block.

Steps Required for Use of Shared Memory

Specifically, **for each Z value**,

- **copy `current` into shared memory**,
- **synchronize** (barrier synchronization),
- use neighboring values
  - from shared memory
  - to **compute and write output**,
- **synchronize**, and
- **advance tiled registers**.

Three Slices Maintained for Inter-Thread Reuse

- In each loop iteration,
  - threads in a block compute
  - a 2-D slice of the block's output prism.

- We then **maintain**
  - **three slices of input** data in **on-chip** memories
  - **`previous`, `current`, and `next`** slices **spread across** the threads' private **registers**, and
  - a second **copy of `current` in shared memory**.

Review: Coarsened Stencil without Shared Memory

```
/* Nx, Ny, Nz: width of grid in X, Y, and Z directions, respectively. */
#define _in(i, j, k)  in[((k)*Ny + (j))*Nx +(i)]
#define _out(i, j, k) out[((k)*Ny + (j))*Nx +(i)]
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;
float previous = _in(i, j, 0);
float current  = _in(i, j, 1);
float next     = _in(i, j, 2);
for (k = 1; k < Nz - 1; k++) {
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {
    _out(i,j,k) = -6 * current + previous + next +
                  _in(i-1, j, k) + _in(i+1, j, k) +
                  _in(i, j-1, k) + _in(i, j+1, k);
  }
  previous = current; current = next; next = _in(i, j, k+2);
}
```

Let's rewrite the loop from our previous kernel.

Add a Tile of Shared Memory

```
// Put this line at top of kernel (not in place of loop).
__shared__ float ds_A[TILE_SIZE][TILE_SIZE];
```

First, we need some shared memory.

Much of the Kernel is the Same

```
// Put this line at top of kernel (not in place of loop).
__shared__ float ds_A[TILE_SIZE][TILE_SIZE];

for (k = 1; k < Nz - 1; k++) {


  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {



  }

  // Next line is unmodified.
  previous = current; current = next; next = _in(i, j, k+2);
}
```

Loop structure, output condition, and register tiling unmodified.

We Leave Some Parts to You

```
// Put this line at top of kernel (not in place of loop).
__shared__ float ds_A[TILE_SIZE][TILE_SIZE];

for (k = 1; k < Nz - 1; k++) {
  // Copy current into shared memory.
  // Barrier: wait for current to become visible.
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {

  }
  // Barrier: wait until finished using shared memory.
  // Next line is unmodified.
  previous = current; current = next; next = _in(i, j, k+2);
}
```

You should know how to do these things…

Is Output Straightforward?

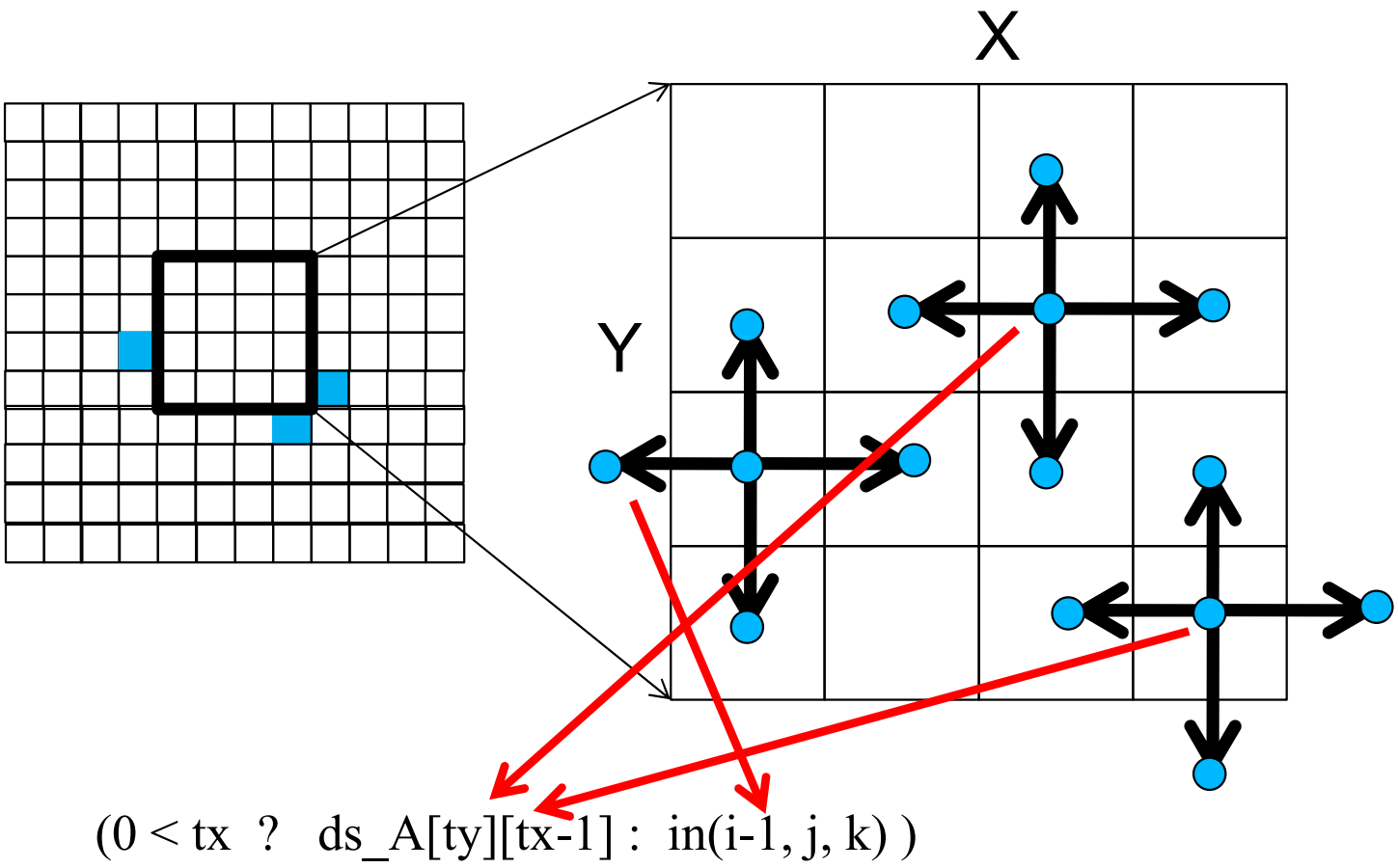
```
// Put this line at top of kernel (not in place of loop).
__shared__ float ds_A[TILE_SIZE][TILE_SIZE];

for (k = 1; k < Nz - 1; k++) {
  // Copy current into shared memory.
  // Barrier: wait for current to become visible.
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {

  }
  // Barrier: wait until finished using shared memory.
  // Next line is unmodified.
  previous = current; current = next; next = _in(i, j, k+2);
}
```

What about the output calculation?

Some Neighbors Not in Shared Memory



Outline of Stencil Kernel with Shared Memory

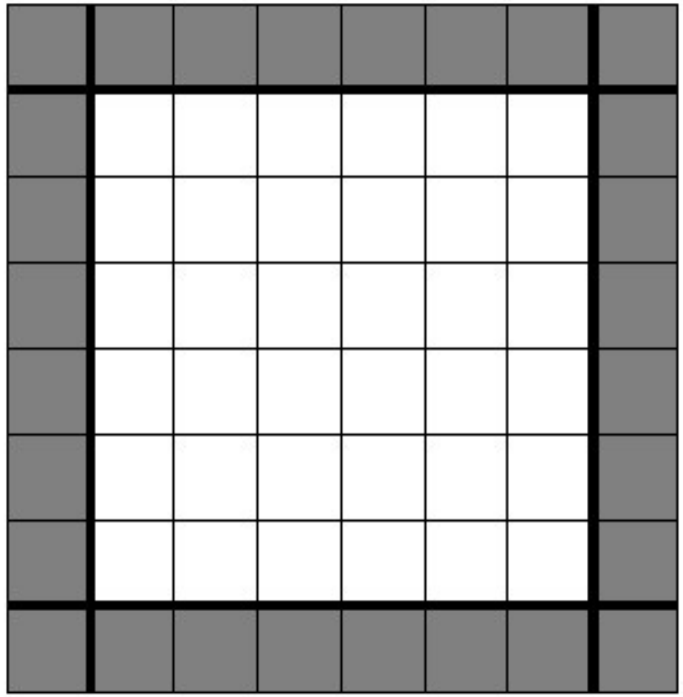
```
// Put this line at top of kernel (not in place of loop).
__shared__ float ds_A[TILE_SIZE][TILE_SIZE];

for (k = 1; k < Nz - 1; k++) {
  // Copy current into shared memory.
  // Barrier: wait for current to become visible.
  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) {
    _out(i,j,k) = -6 * current + previous + next +
                  (0 < tx? ds_A[ty][tx-1] : _in(i-1, j, k))
                  // Add conditional terms for other neighbors.
  }
  // Barrier: wait until finished using shared memory.
  // Next line is unmodified.
  previous = current; current = next; next = _in(i, j, k+2);
}
```

Halo Data Can Limit Reuse

Halo data can be a problem!

For each **N×M** tile to be computed, we load **(N + 2) × (M + 2) – 4** inputs.



N and M limited by registers per SM.

Analysis Illustrates Limit for Reasonable N and M

Consider **N=16** and **M=8**.

- **Without shared memory**,
  - each thread loads 5 values,
  - for a total of **16 × 8 × 5 = 640** global memory **reads**.
- **Shared memory reduces loads to**
  (16 + 2) × (8 + 2) – 4 = 176 values.

Ratio of **improvement is 640 / 176 = 3.6**
rather than the ideal value of 5.

Other Approaches are Possible

Other approaches are possible. **In early GPUs** (and 408!), **parallelizing global loads was better**:

- each thread loads one value from global memory to shared,
- and some threads compute outputs,
- reducing non-coalesced accesses during output calculation
- and replacing control divergence with idle threads.

On recent GPUs, however, **L2 cache holds** most **halo** values!

- Pulled into cache by neighboring thread blocks.
- Conditional **approach** (as **shown**) **is** actually **better**!

Performance Data: Parallelized Loads or Conditionals?

Lumetta's implementations

1. Parallelized loads: **each block reads 32×32** X-Y slice to shared memory, **computes 30×30** X-Y output slice
2. Conditions in output calc.: **32×32 blocks**; each **input implemented as a conditional** from shared (interior) or global memory (on boundary).

Executed on **30 August 2021** using exclusive queue on a **4096×4096×64** grid (**Titan V GPU**).

Time with parallelized loads: 20.8 msec

Time with conditions in output calc.: 19.0 msec

ANY QUESTIONS?