Lecture 12:

Structured Grids (Stencils)

Modern Parallel Computing

John Owens

EEC 289Q, UC Davis, Winter 2018

Credits

- Original lecture: Kerry Seitz, with material from:
  - Tobias Brandvik (Whittle Laboratory, University of Cambridge)
    - "Stencil operations in CUDA."
    - http://www.nvidia.com/content/PDF/isc-2011/Brandvik.pdf
  - Samuel Williams (Lawrence Berkeley National Laboratory)
    - "HPGMG-FV"
  - Jonathan Ragan-Kelley (MIT CSAIL; now at Berkeley)
    - "Decoupling Algorithms from the Organization of Computation for High-Performance Imaging"

What is a stencil?

- Class of iterative algorithms that update array elements according to some fixed pattern (stencil)

- Special case of a gather
  - Gather addresses are fixed offsets relative to the output location

- Used in simulations, solving PDEs, image processing, and cellular automata

- One of the "dwarfs" that we talked about several classes ago (structured grid)

What is a stencil?

- Code performs a sequence of sweeps (timesteps) through an array (usually 2D or 3D)

- In each timestep, all cells are updated using neighboring elements in a fixed pattern (stencil – shape is application-pecific)

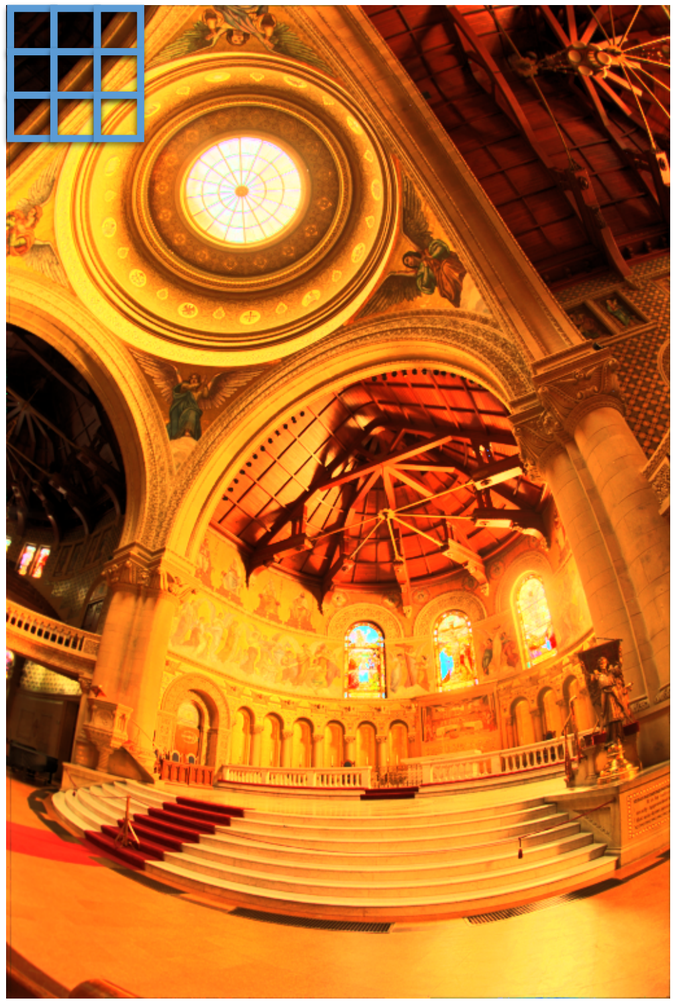
- Each timestep uses data from the previous timestep(s) only
  - Great for parallel processing

- Data access pattern is repeated

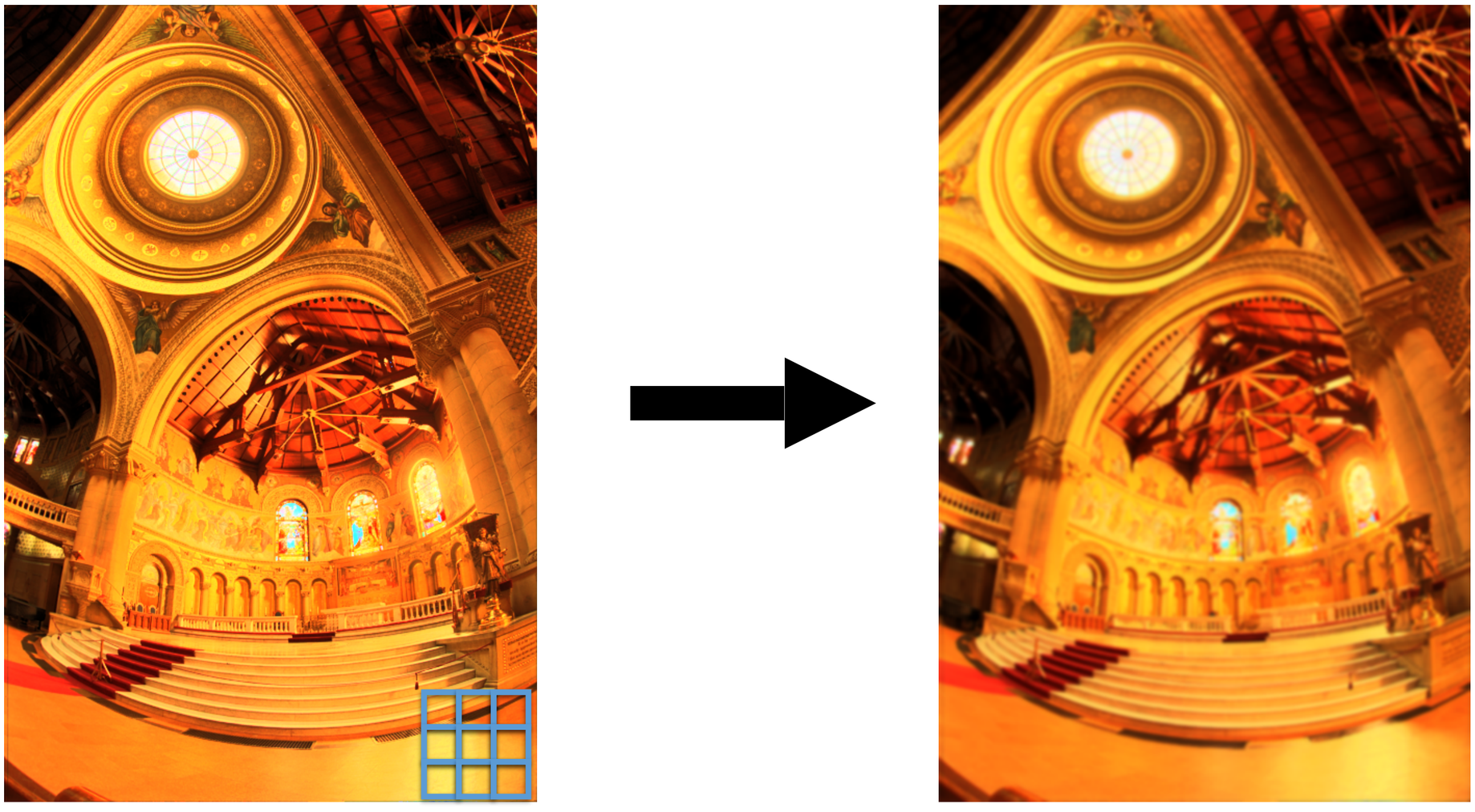
Stencil operations in one slide

- Parallelize over input or output?
  - Output
- What does each thread do?
  - Read all inputs in stencil
  - Do a math operation on those inputs
  - (Optionally) use some weights in that computation
    - Weights can be stored locally since they are reused, e.g., shared memory
- Likely limited by memory bandwidth
- Keys: efficient memory access; exploit locality

Box Blur (3x3)



Box Blur (3x3)



Box Blur (3x3)

```
void blur(Image &in, Image &blurred) {

  for(int y=0; y < in.height; ++y)
    for(int x=0; x < in.width; ++x)
      blurred(x,y) = (in(x-1, y-1)  + in(x, y-1)  + in(x+1, y-1)  +
                      in(x-1, y)    + in(x, y)    + in(x+1, y)    +
                      in(x-1, y+1) + in(x, y+1) + in(x+1, y+1)    ) / 9;
}
```

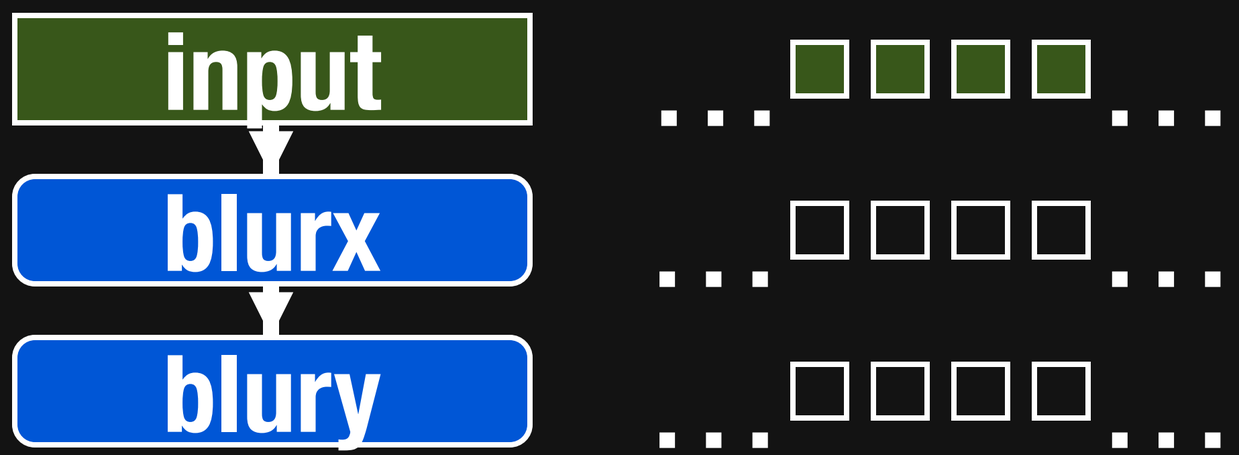
Separable Box Blur

```
void blur(Image &in, Image &blurred) {
  Image tmp(in.width(), in.height);

  for(int y=0; y < in.height; ++y)
    for(int x=0; x < in.width; ++x)
      tmp(x,y) = (in(x-1, y) + in(x, y) + in(x+1, y)) / 3;

  for(int y=0; y < in.height; ++y)
    for(int x=0; x < in.width; ++x)
      blurred(x,y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1)) / 3;
}
```

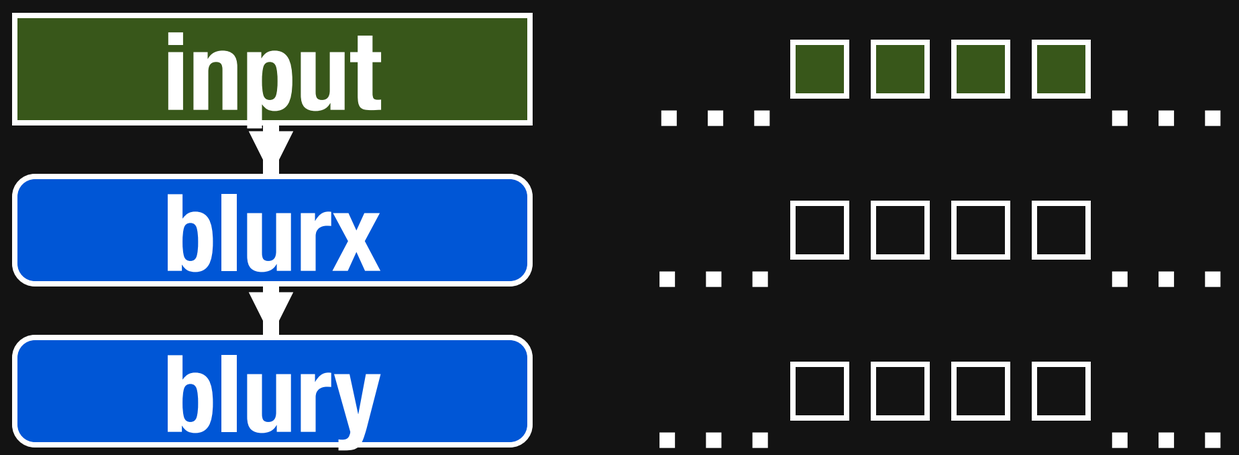
First option for GPU Implementation

- Do entire x-blur, then entire y-blur
- https://youtu.be/3uiEyEKji0M?t=6m32s

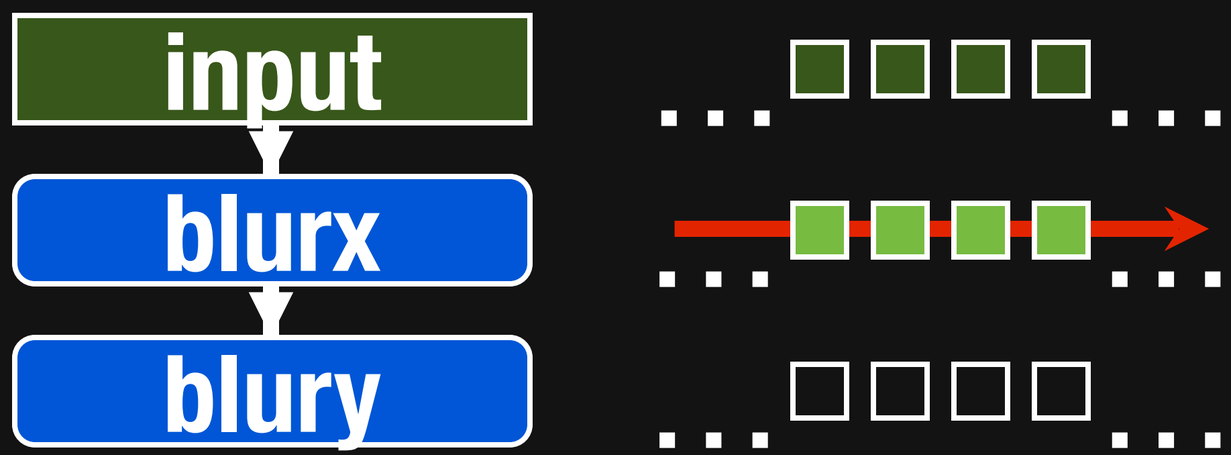
Organizing a data-parallel pipeline



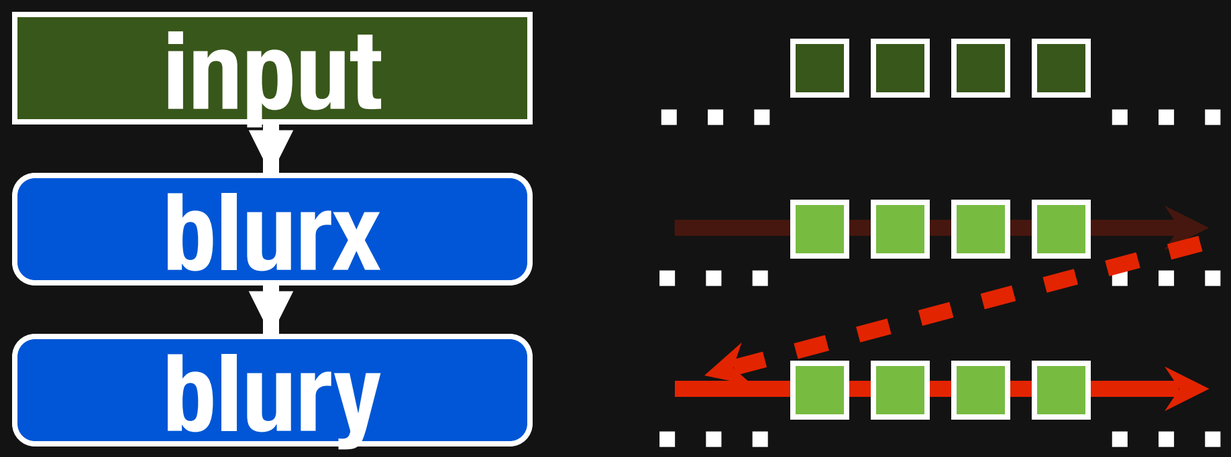
Simple loops execute **breadth-first** across stages



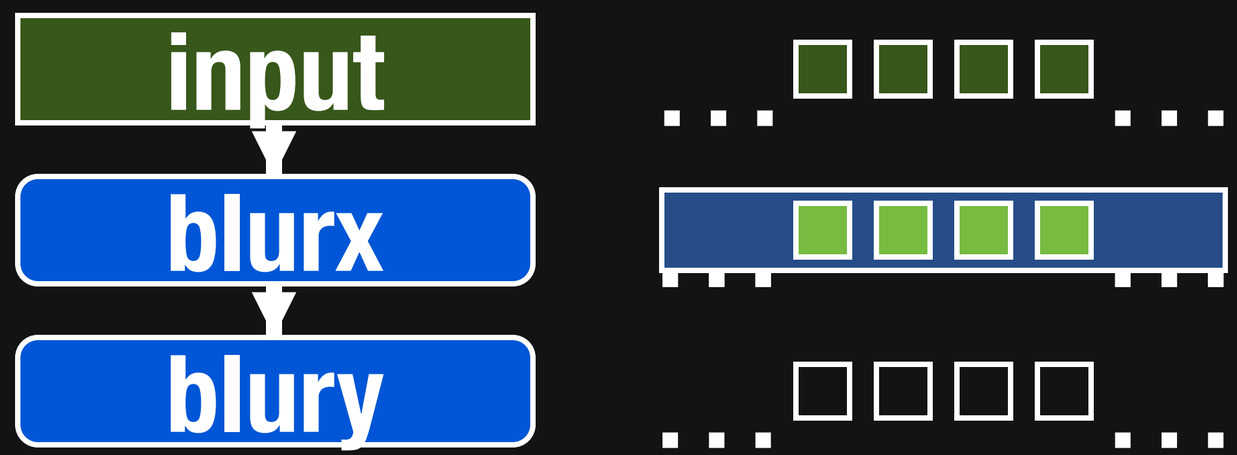
Simple loops execute **breadth-first** across stages



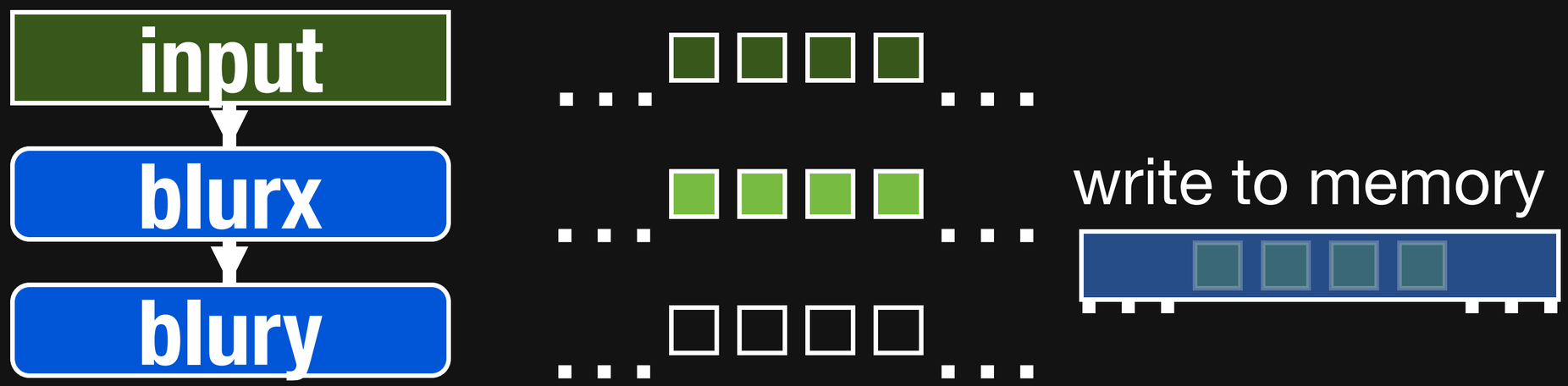
Simple loops execute **breadth-first** across stages



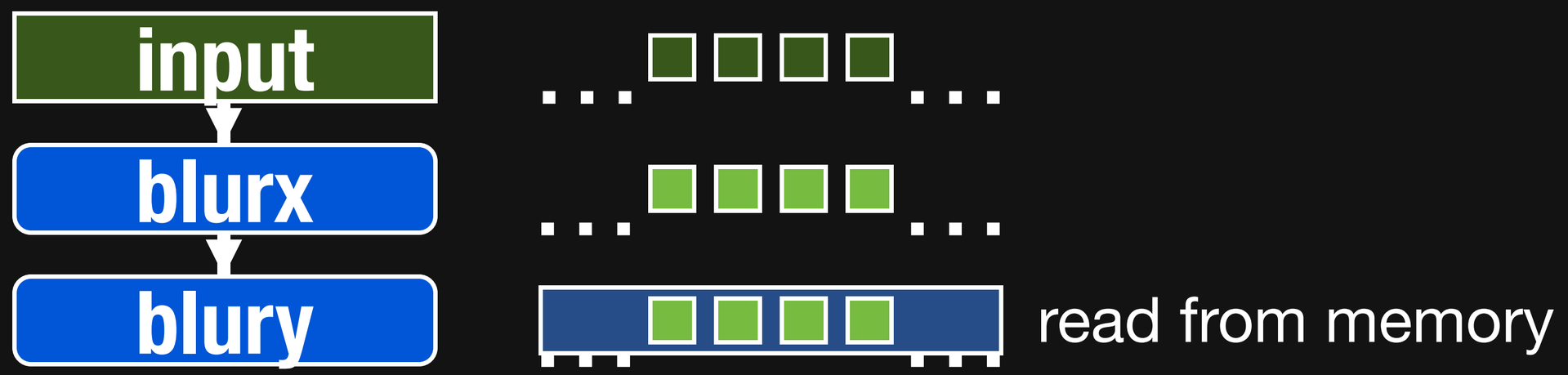
Simple loops execute **breadth-first** across stages



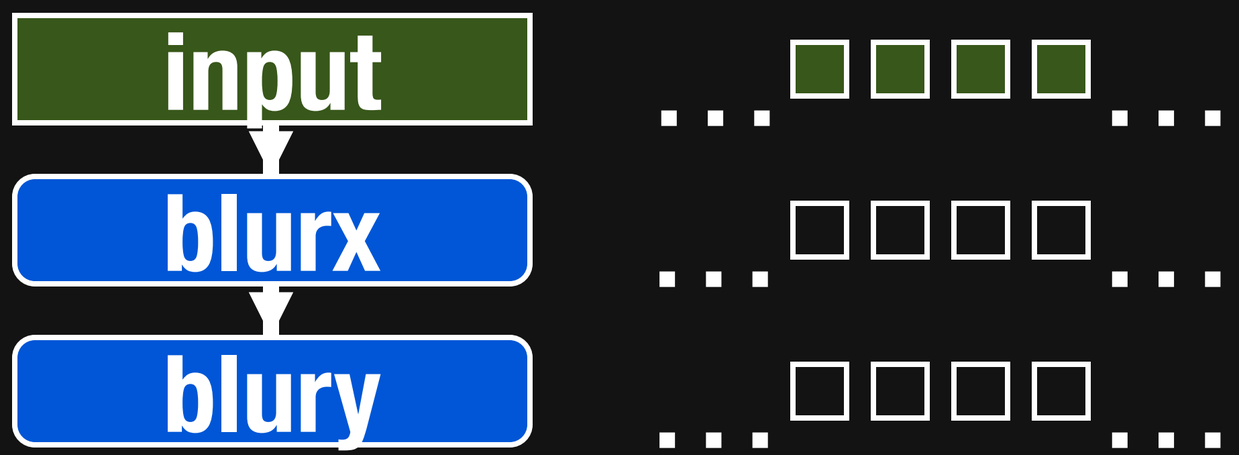
Simple loops execute **breadth-first** across stages



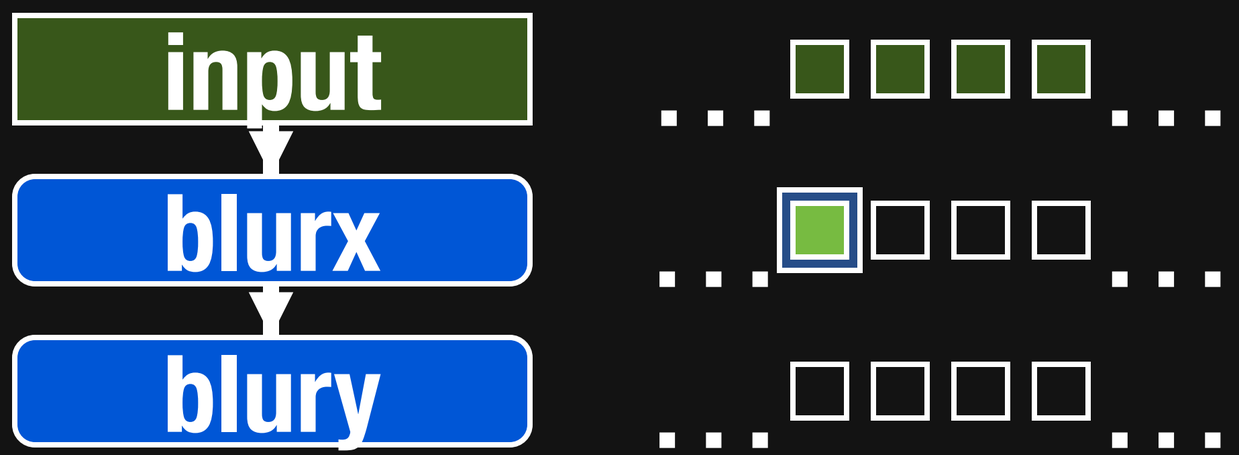
Simple loops execute **breadth-first** across stages



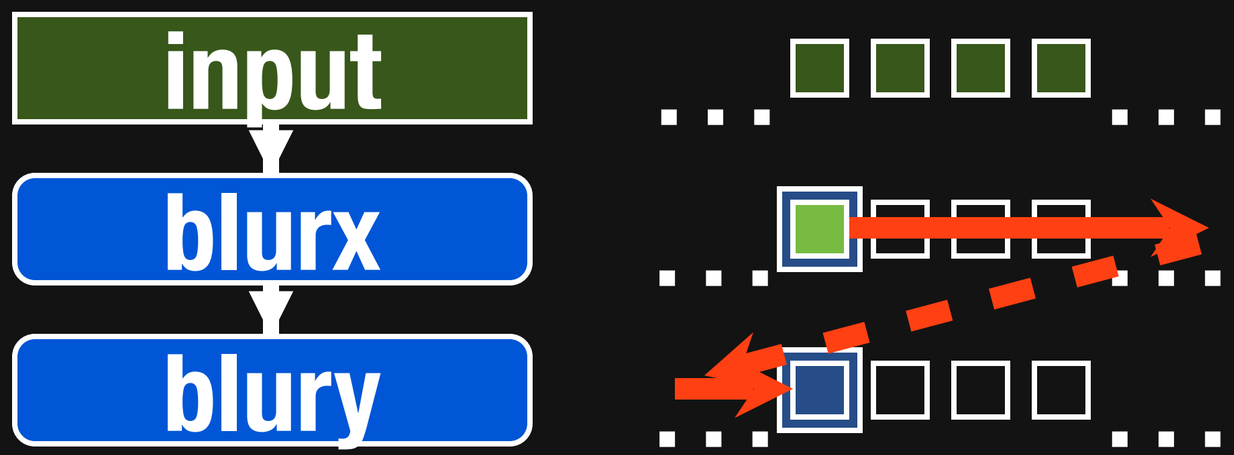
Breadth-first execution sacrifices locality



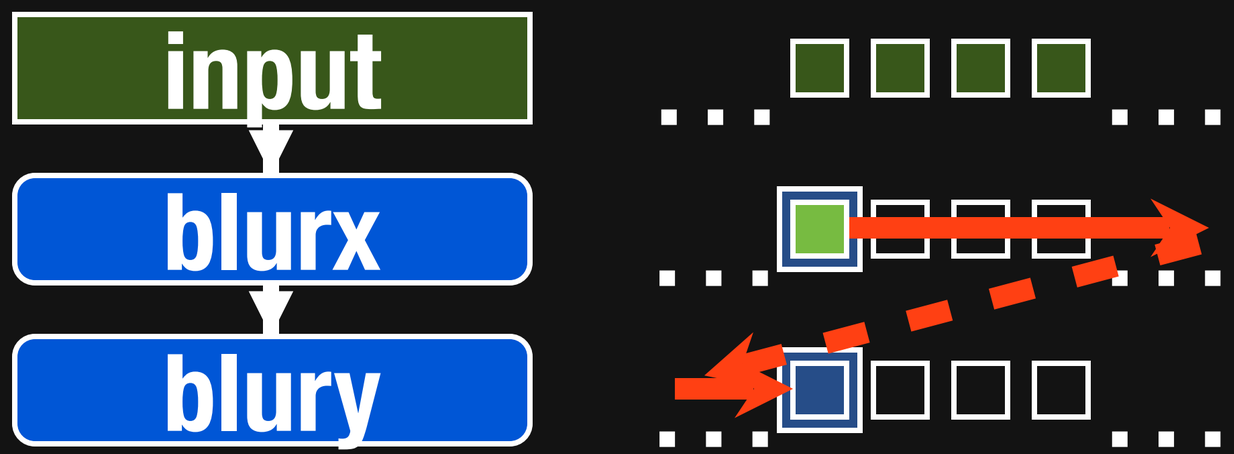
Breadth-first execution sacrifices locality



Breadth-first execution sacrifices locality



Breadth-first execution sacrifices locality



locality is a function of **reuse distance**

Two Options for GPU Implementation

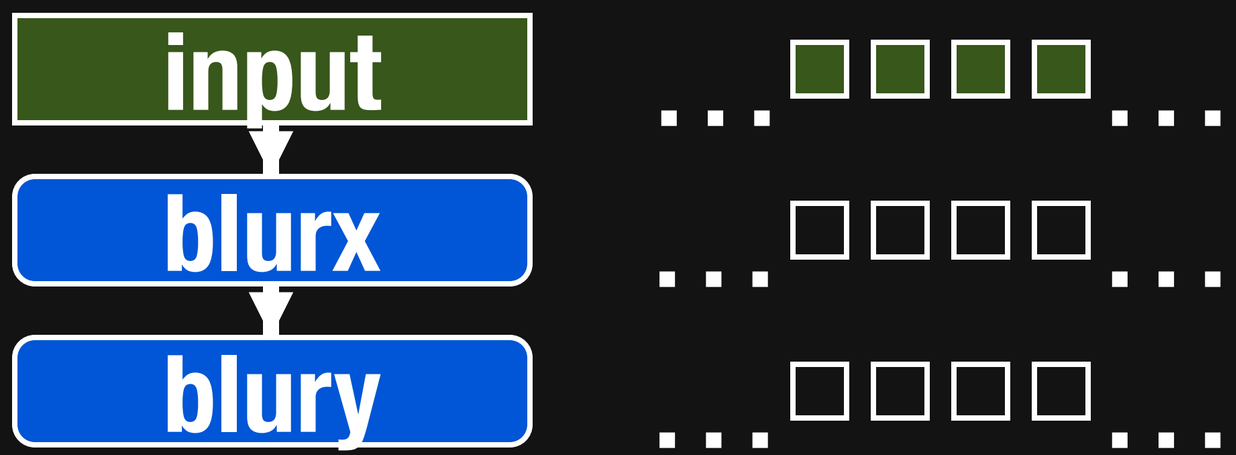
- Do entire x-blur, then entire y-blur
  - https://youtu.be/3uiEyEKji0M?t=6m32s

- Do y-blur and compute x-blur as you go
  - https://youtu.be/3uiEyEKji0M?t=7m47s

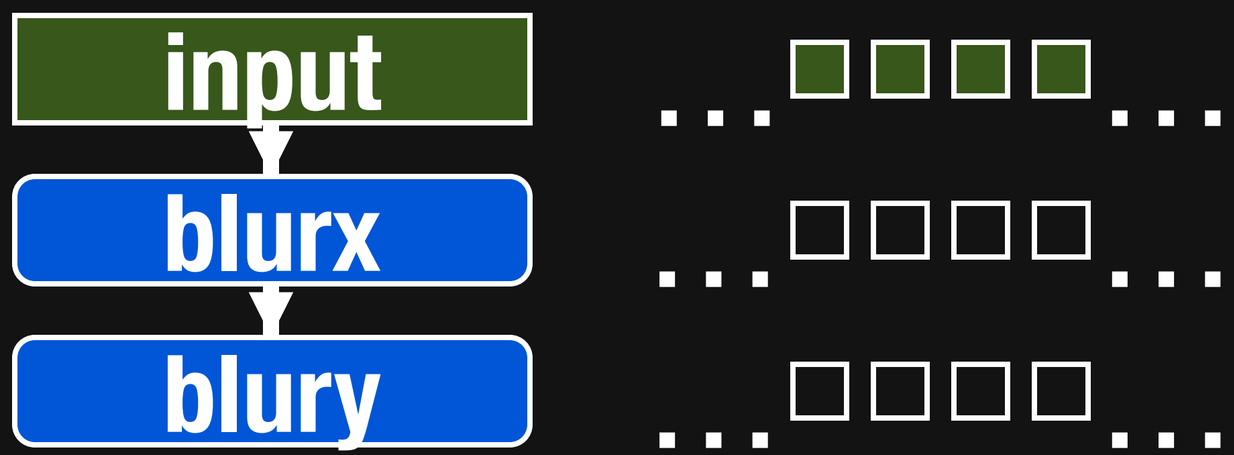
Compute X-Blur as needed

```
__global__ void blurX(Image &in, Image &out) {
  int x = threadIdx.x + blockIdx.x * blockDim.x
  int y = threadIdx.y + blockIdx.y * blockDim.y

  float tmp[3];
  for(int i = 0; i < 3; ++i) {
    int yi = y + i - 1;
    tmp[i] = (in(x-1, yi) + in(x, yi) + in(x+1, yi)) / 3;
  }

  out(x,y) = (tmp[0] + tmp[1] + tmp[2]) / 3;
}
```

Interleaved execution improves locality

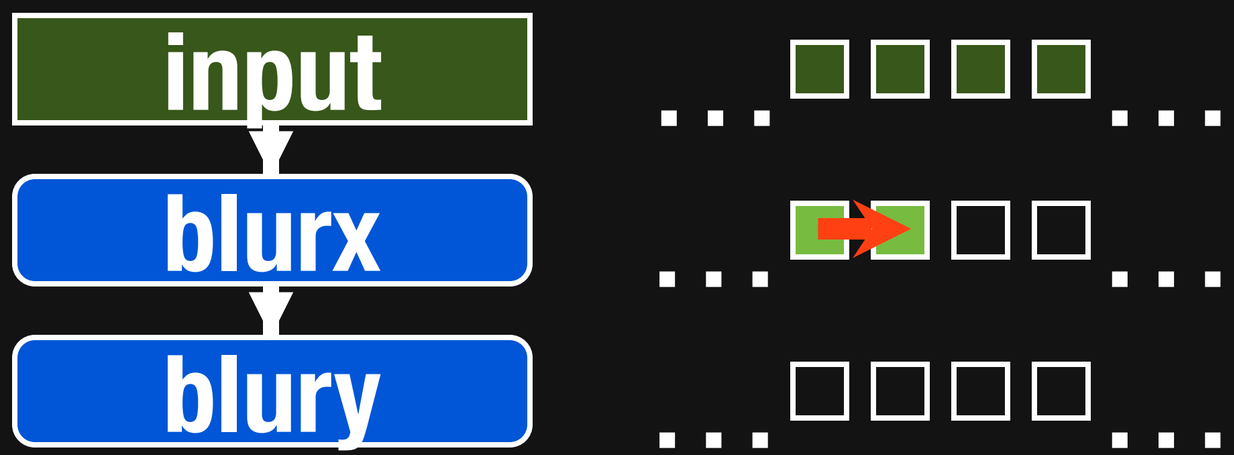


Interleaved execution improves locality



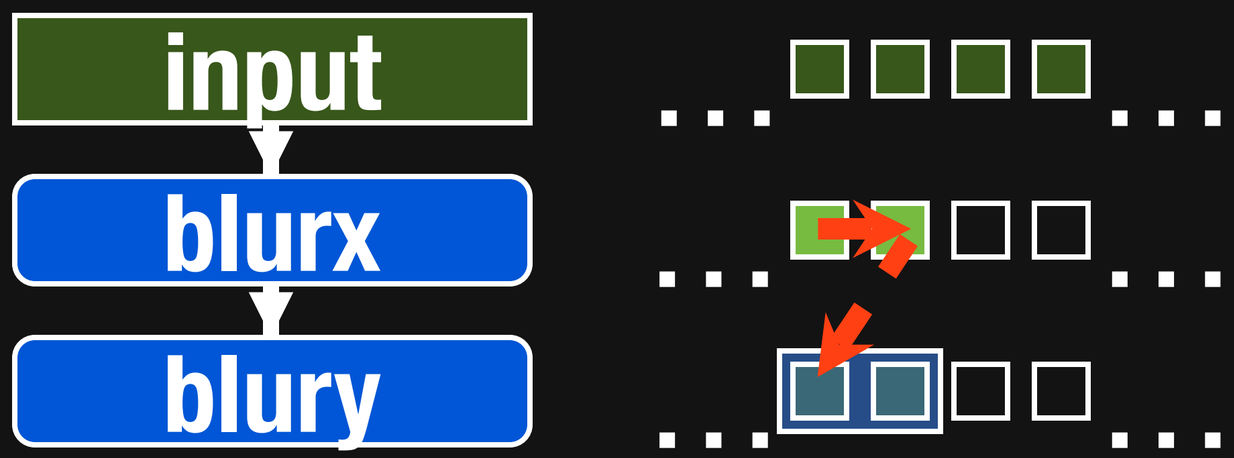
(simplified)

Interleaved execution improves locality



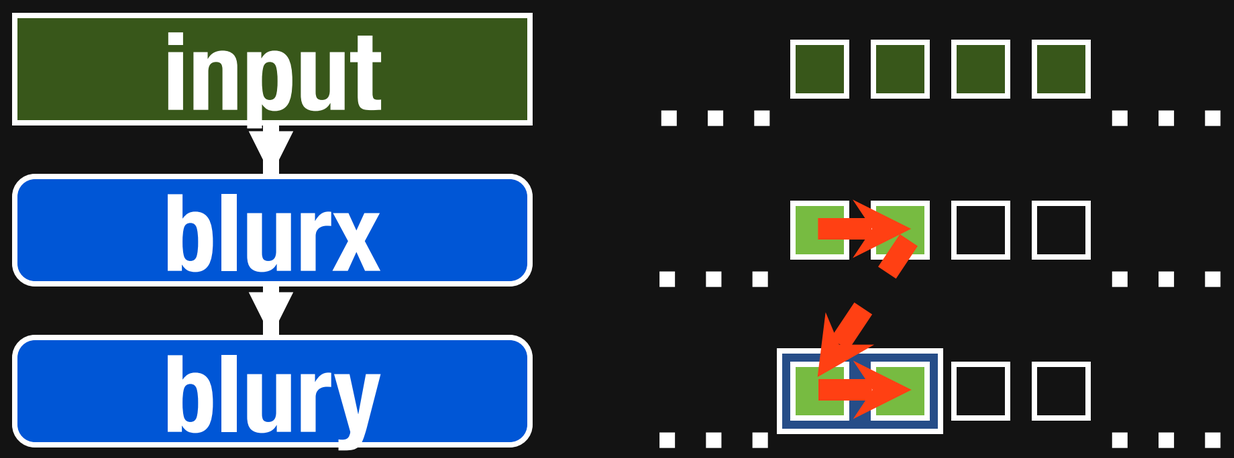
(simplified)

Interleaved execution improves locality



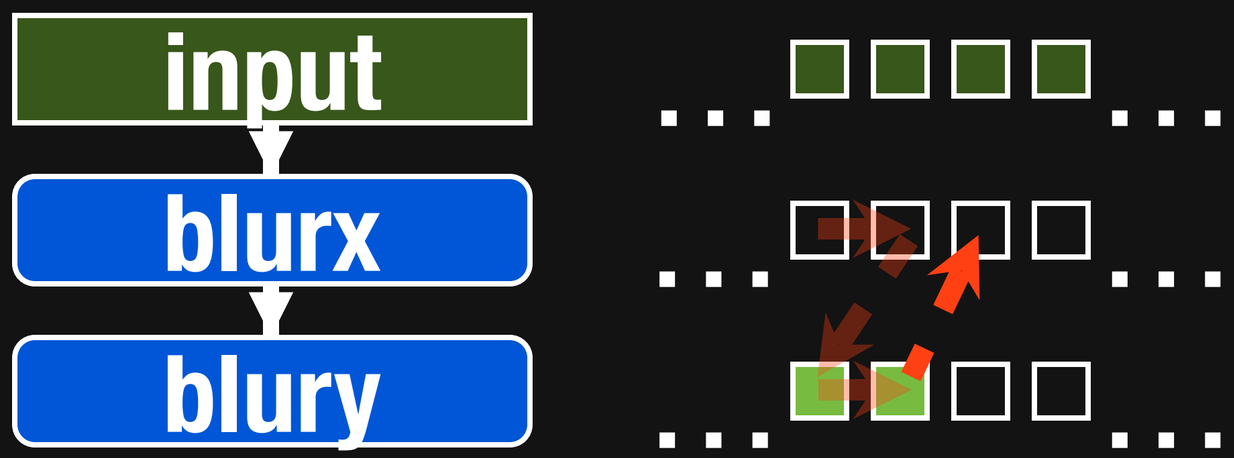
(simplified)

Interleaved execution improves locality



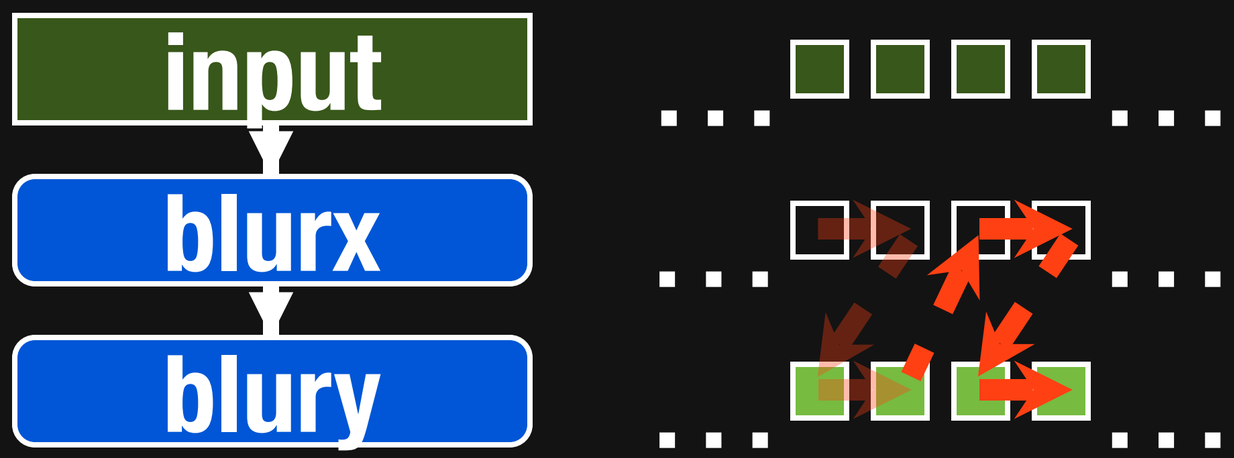
(simplified)

Interleaved execution improves locality



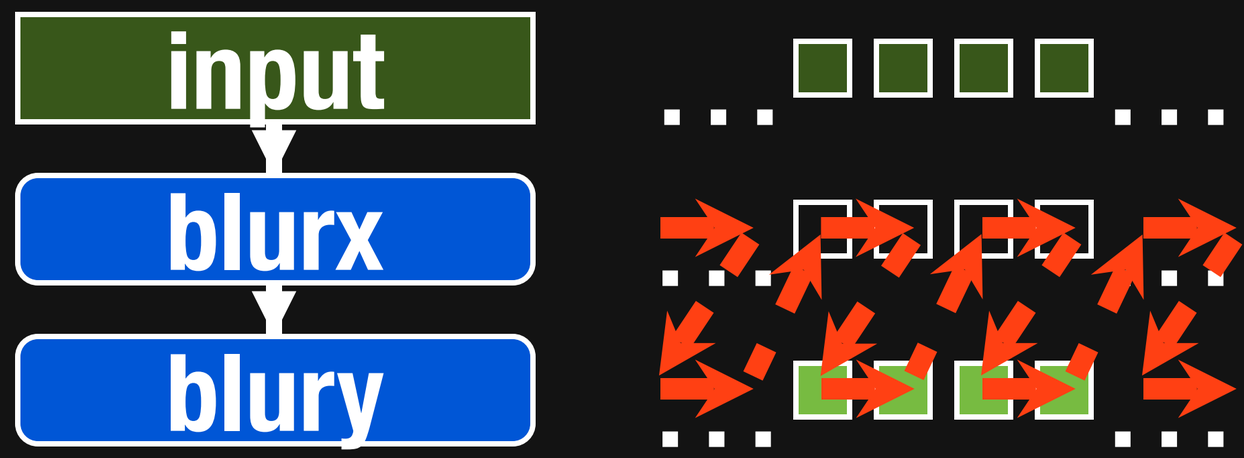
(simplified)

Interleaved execution improves locality

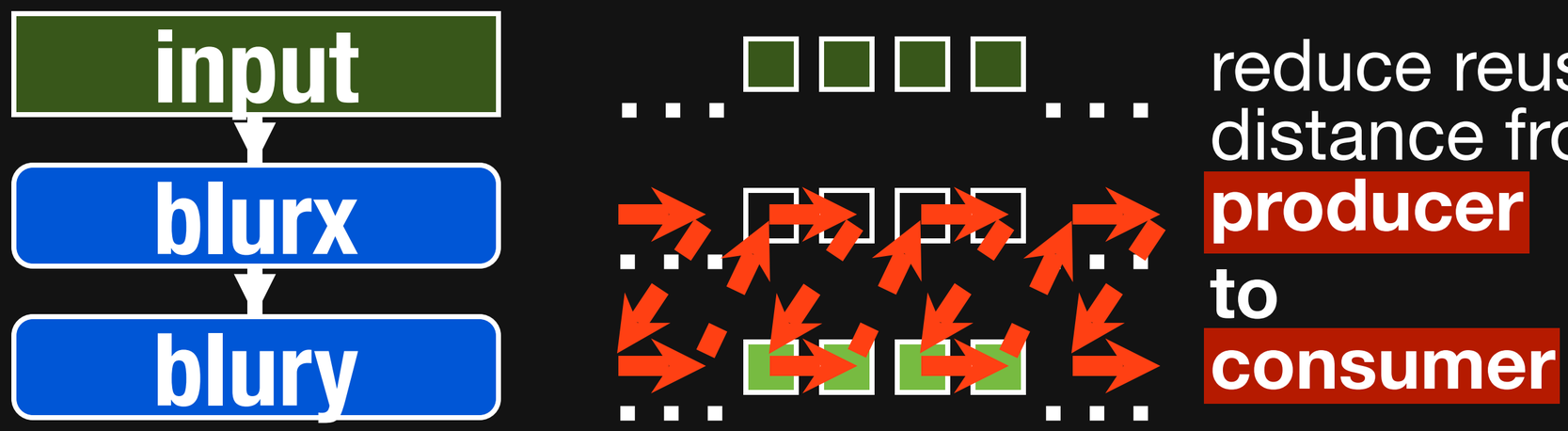


(simplified)

Interleaved execution improves locality

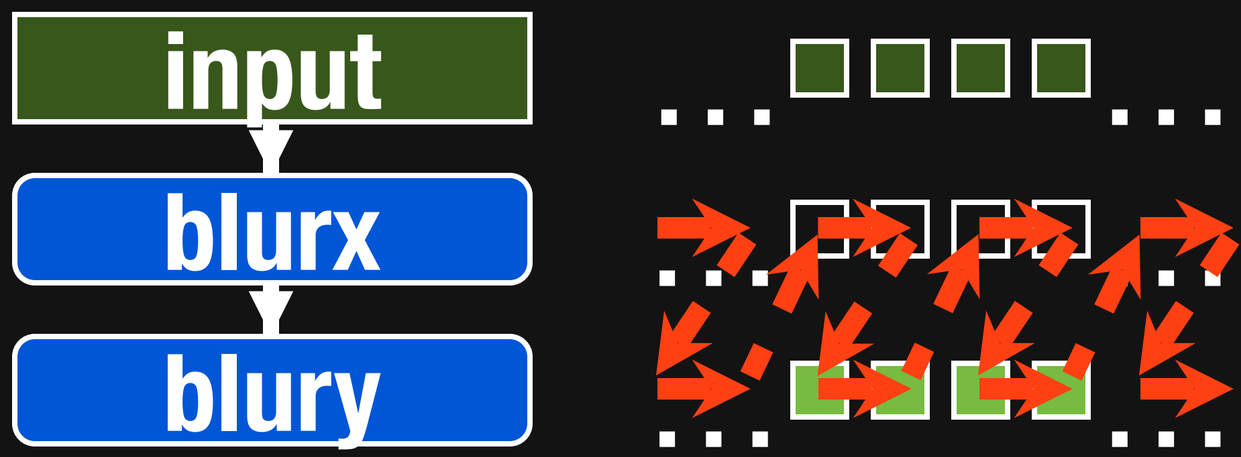


Interleaved execution improves locality

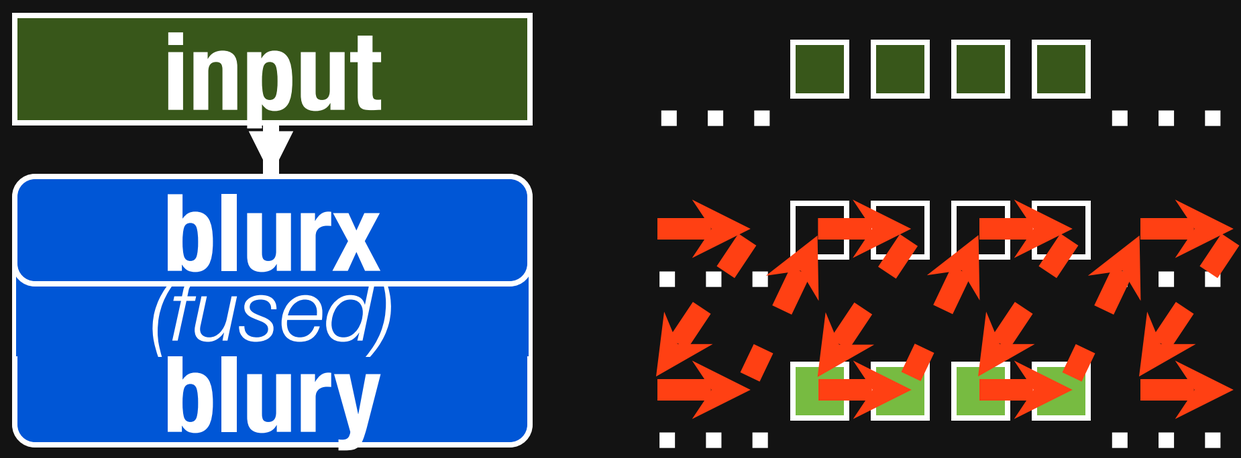


reduce reuse distance from **producer** to **consumer**

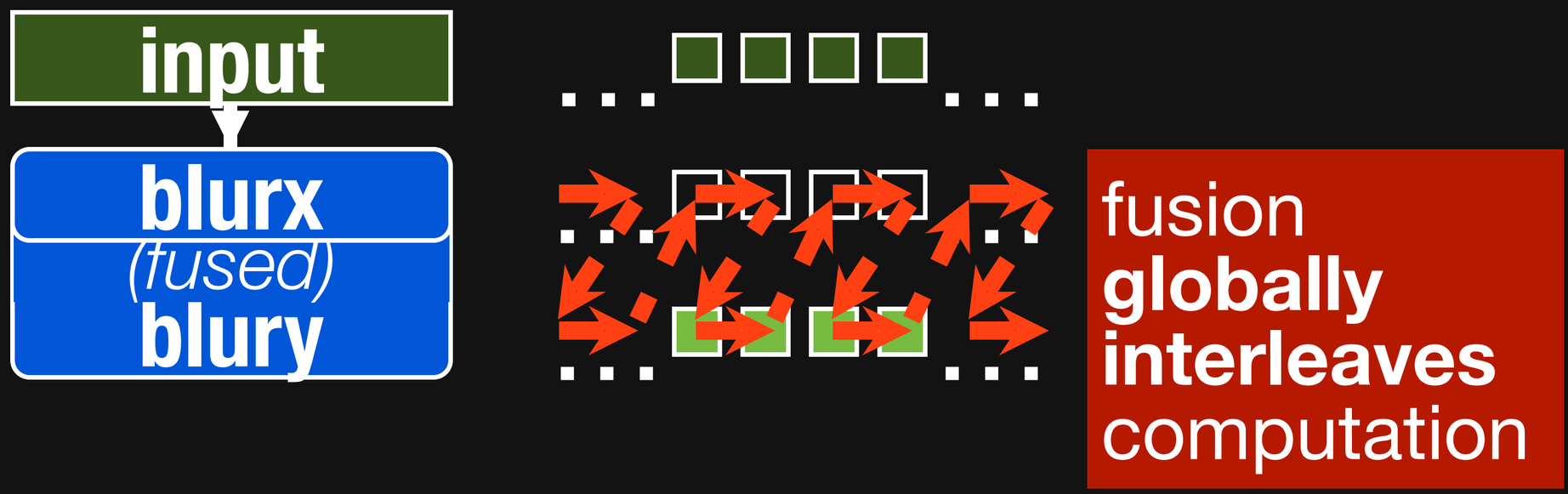
Fusion improves locality



Fusion improves locality



Fusion improves locality



Two Options for GPU Implementation

X-Blur then Y-Blur

- Requires communication between threads (kernel break)
- No redundant computation

X-Blur as needed

- No communication between threads
- Lots of redundant computation

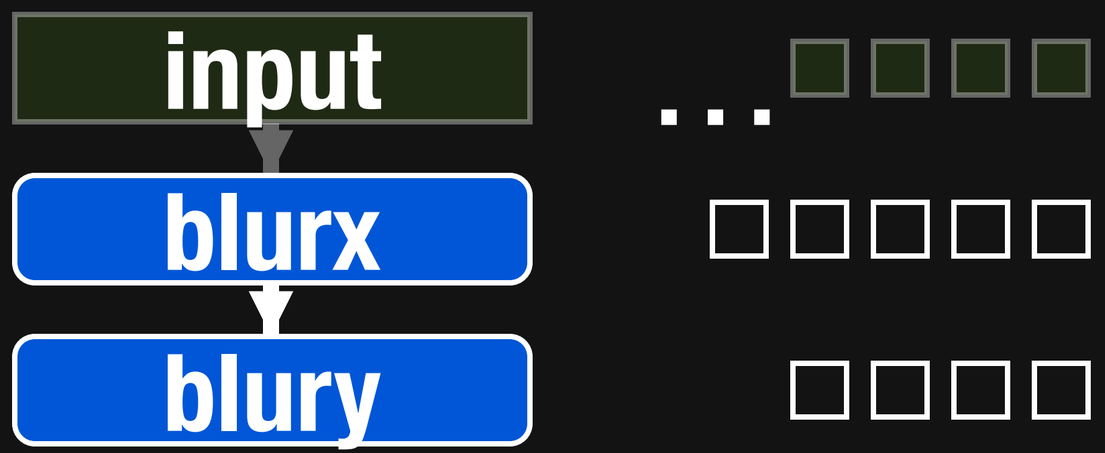
Understanding dependencies

Understanding dependencies

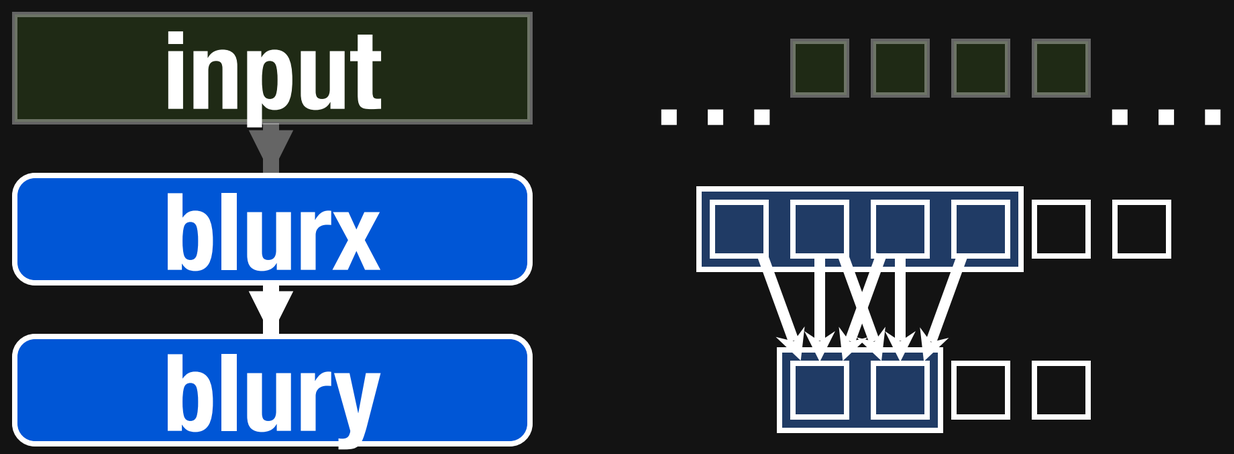
Understanding dependencies

Understanding dependencies

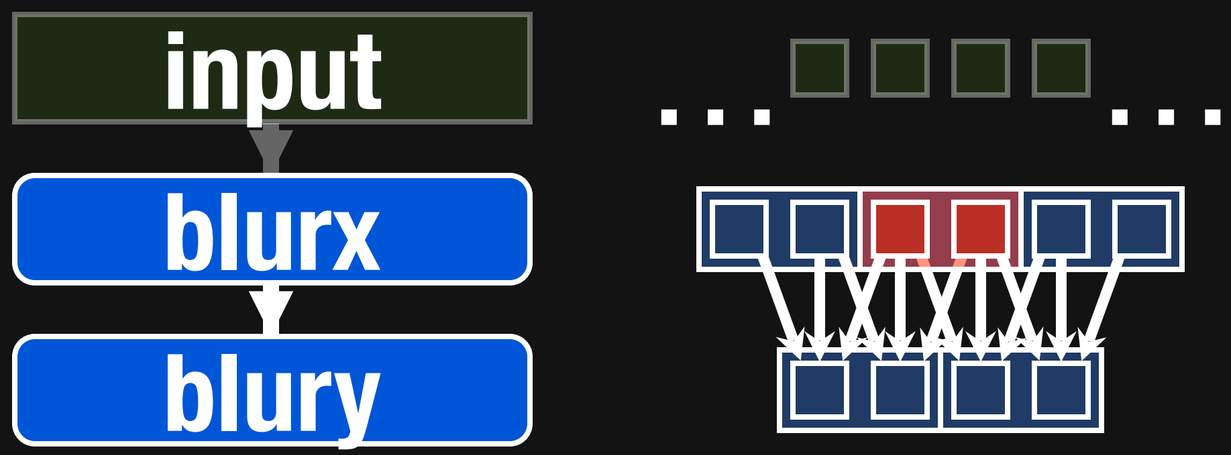
Stencils have overlapping dependencies



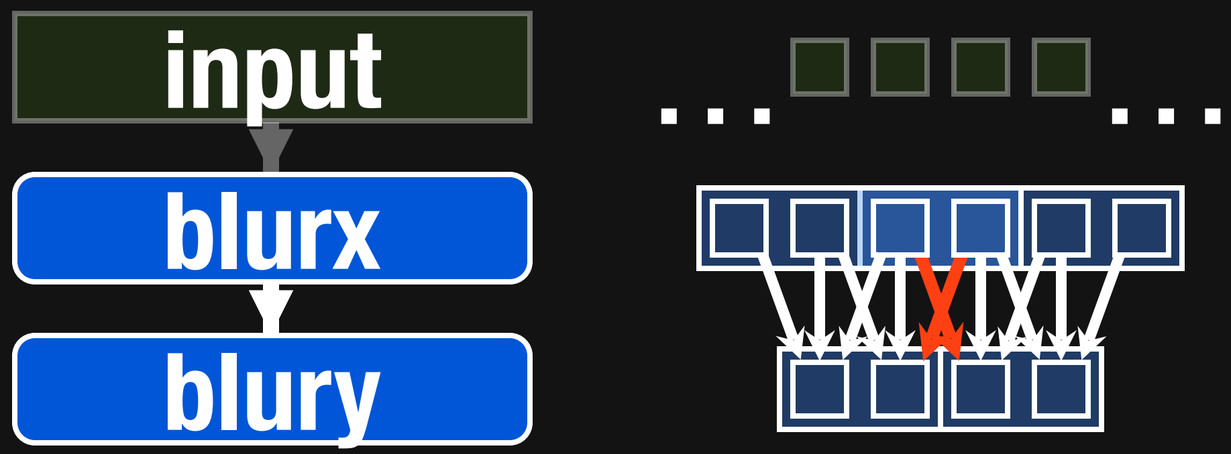
Stencils have overlapping dependencies



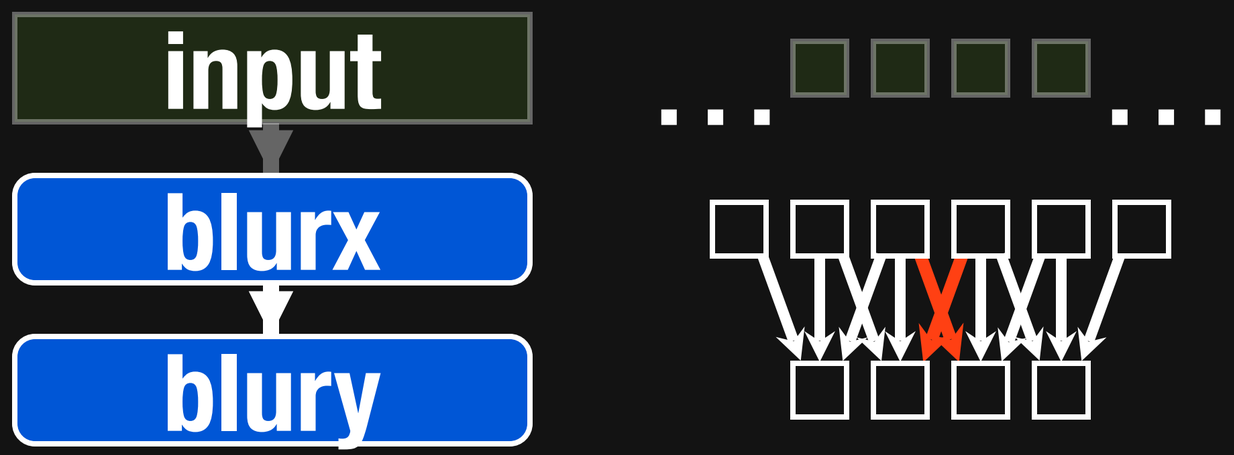
Stencils have overlapping dependencies



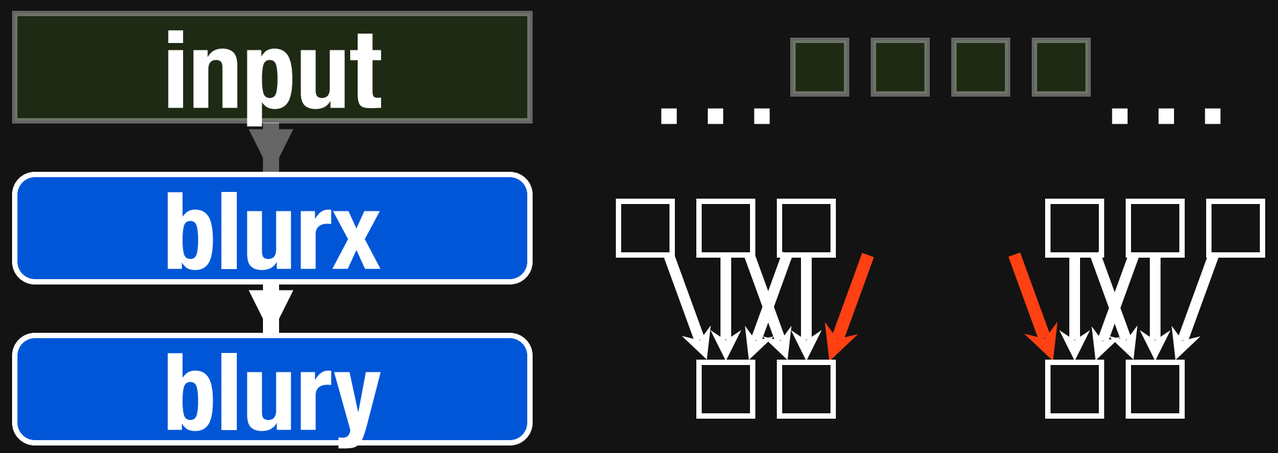
Stencils have overlapping dependencies



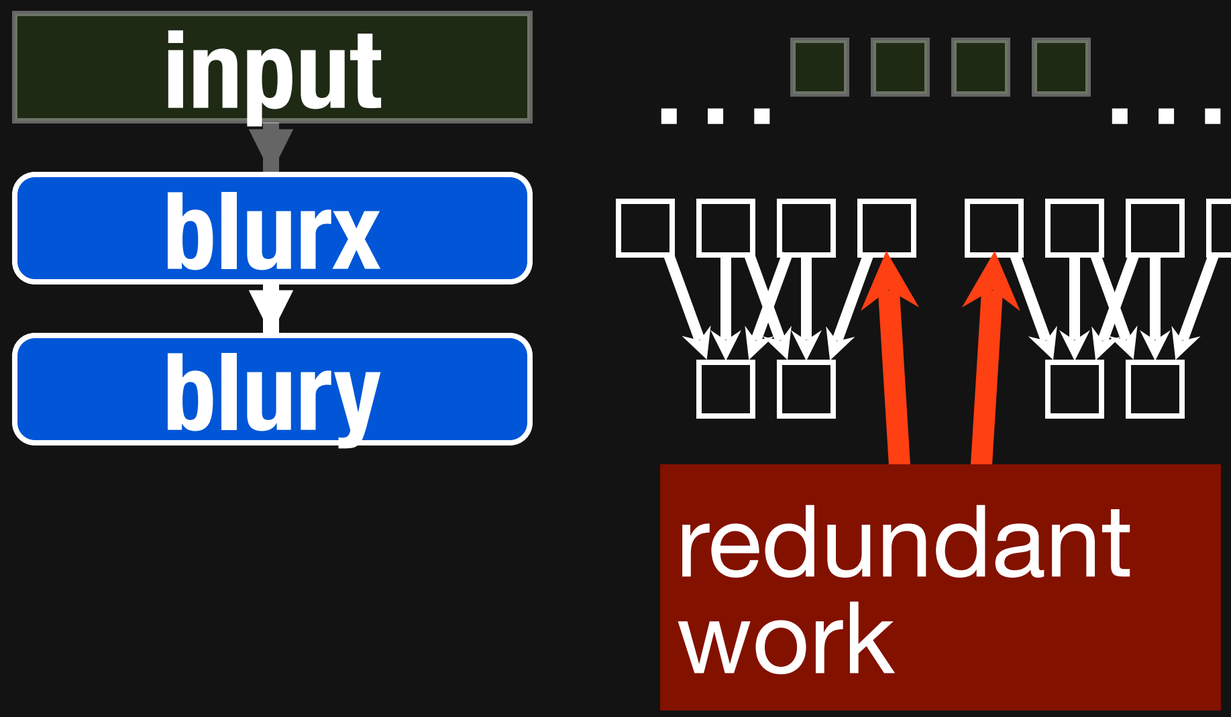
Breaking dependencies introduces redundant work



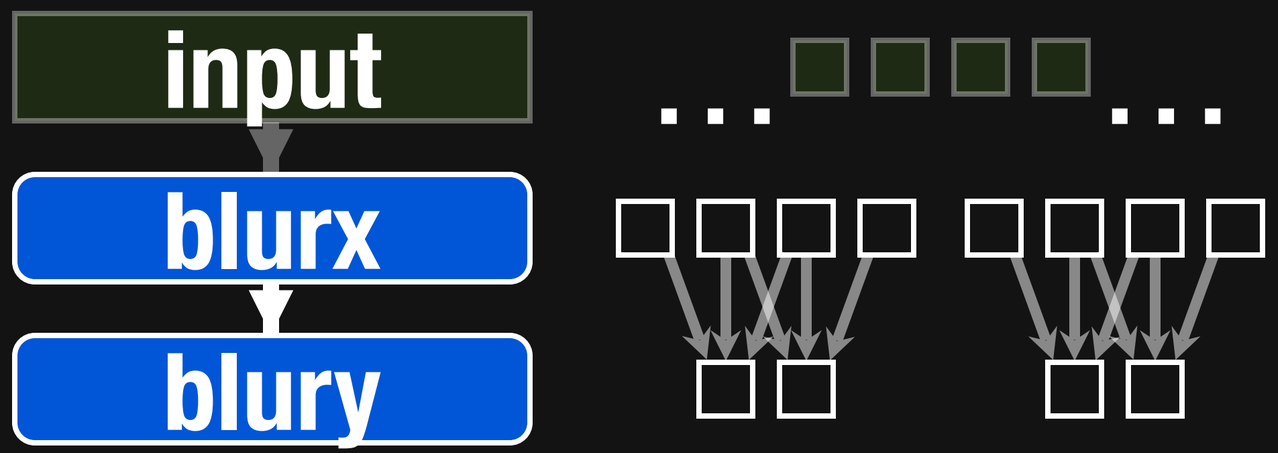
Breaking dependencies introduces redundant work



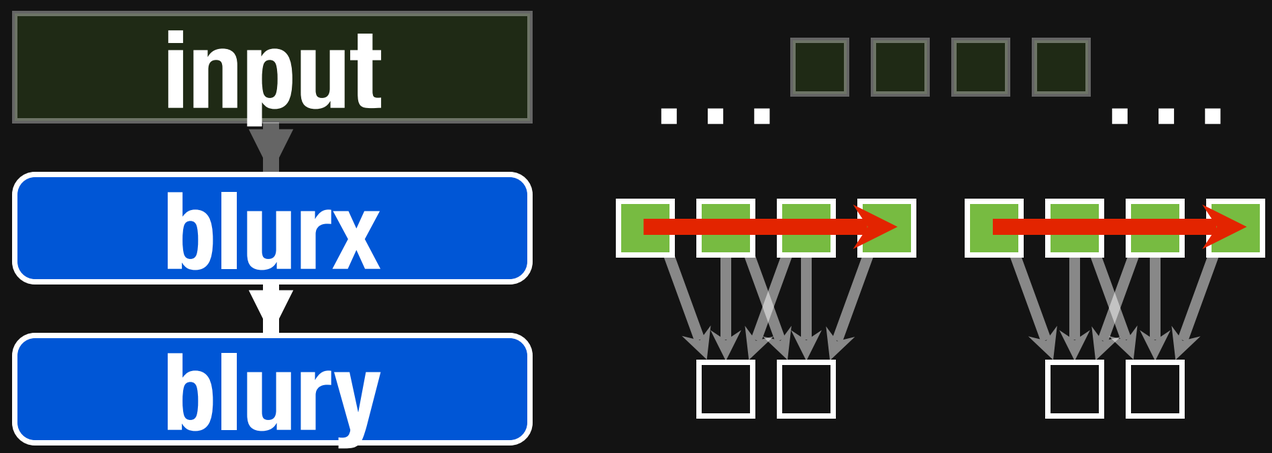
Breaking dependencies introduces redundant work



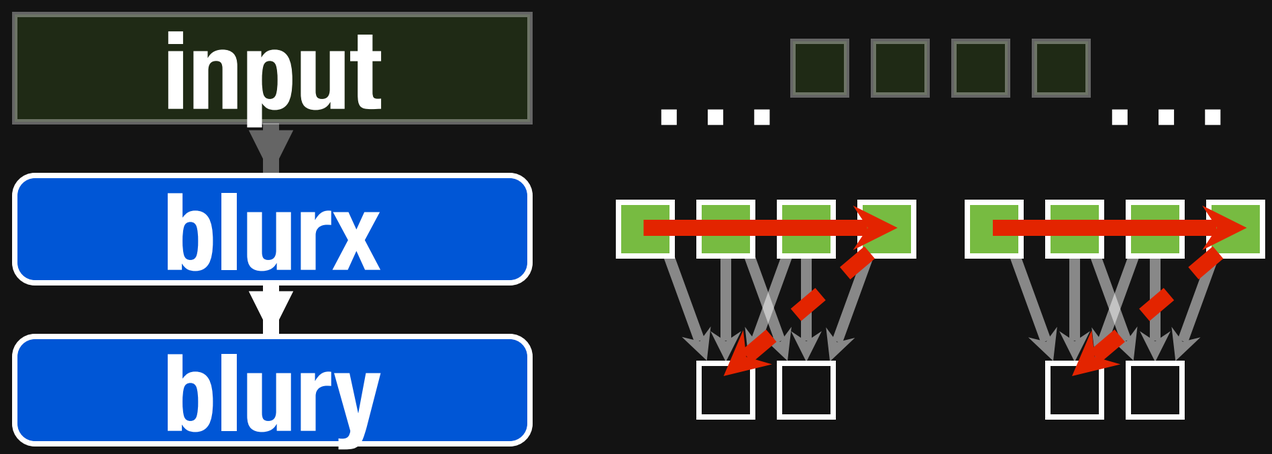
Decoupled tiles optimize parallelism & locality



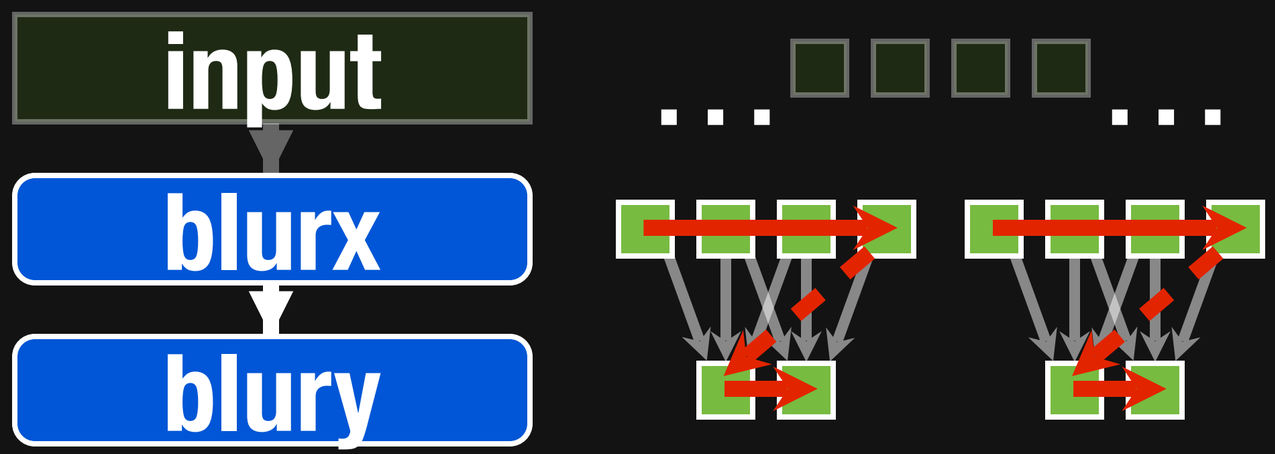
Decoupled tiles optimize parallelism & locality



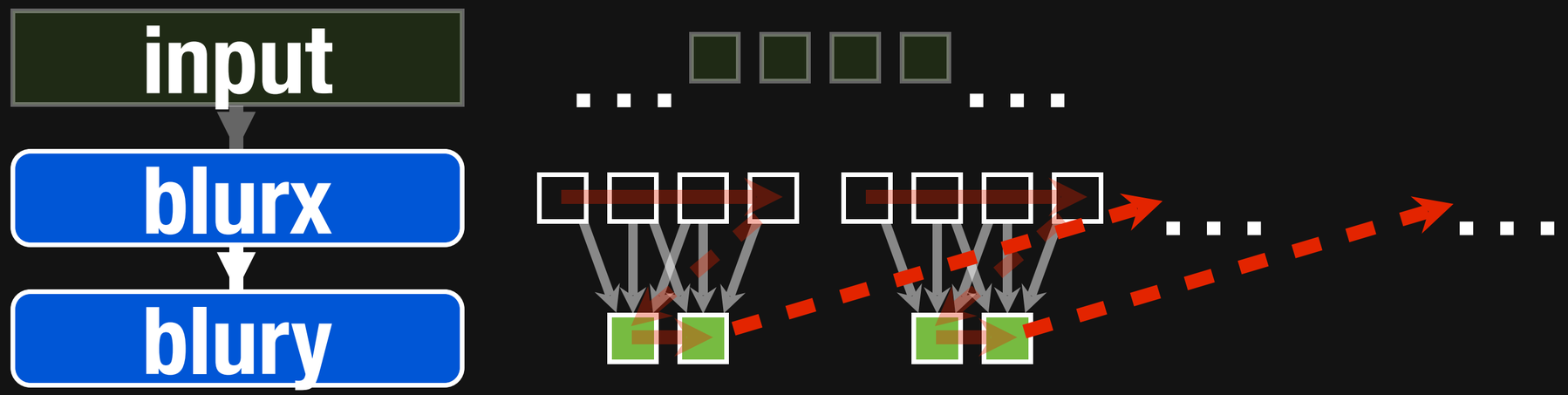
Decoupled tiles optimize parallelism & locality



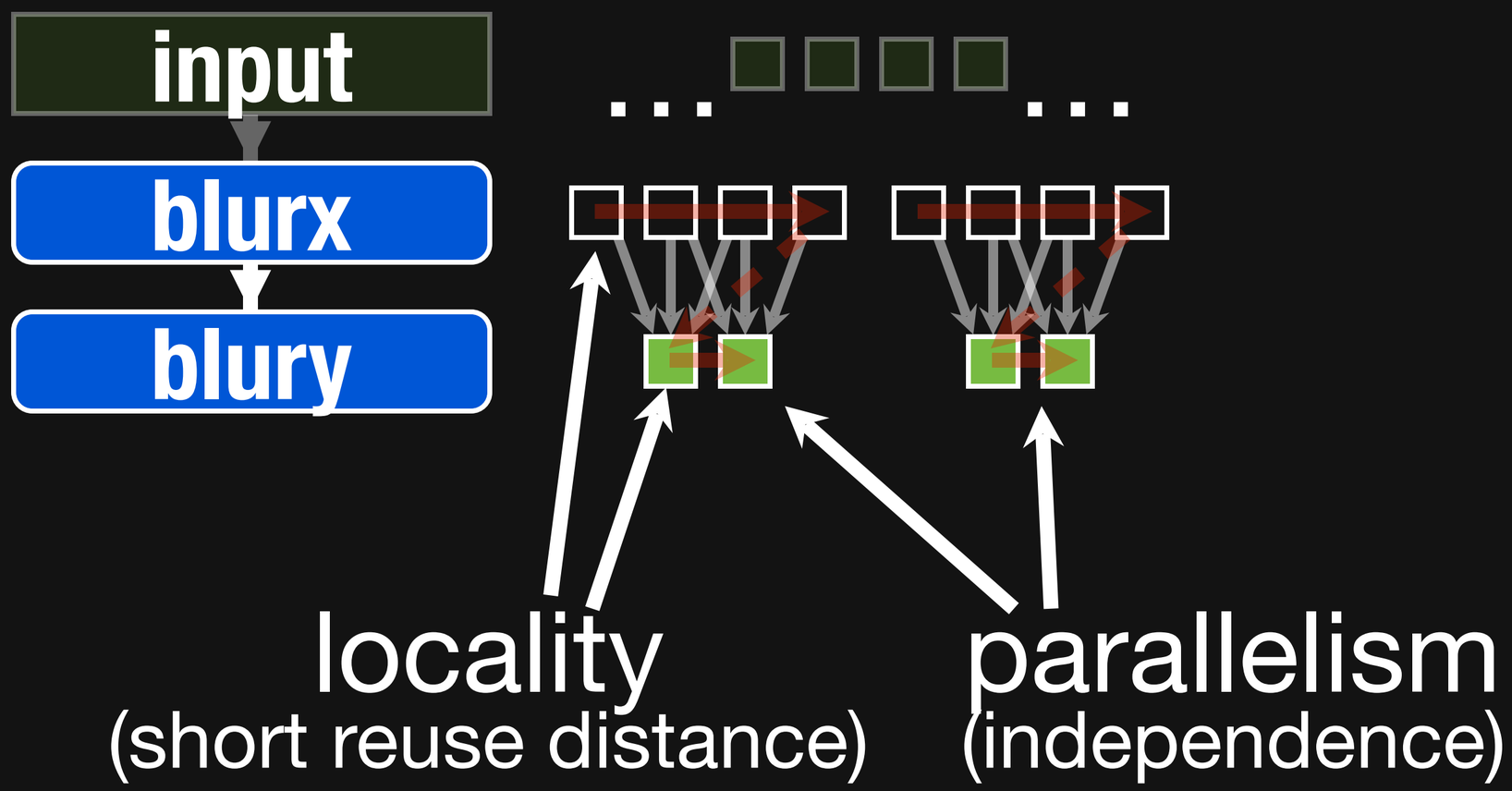
Decoupled tiles optimize parallelism & locality



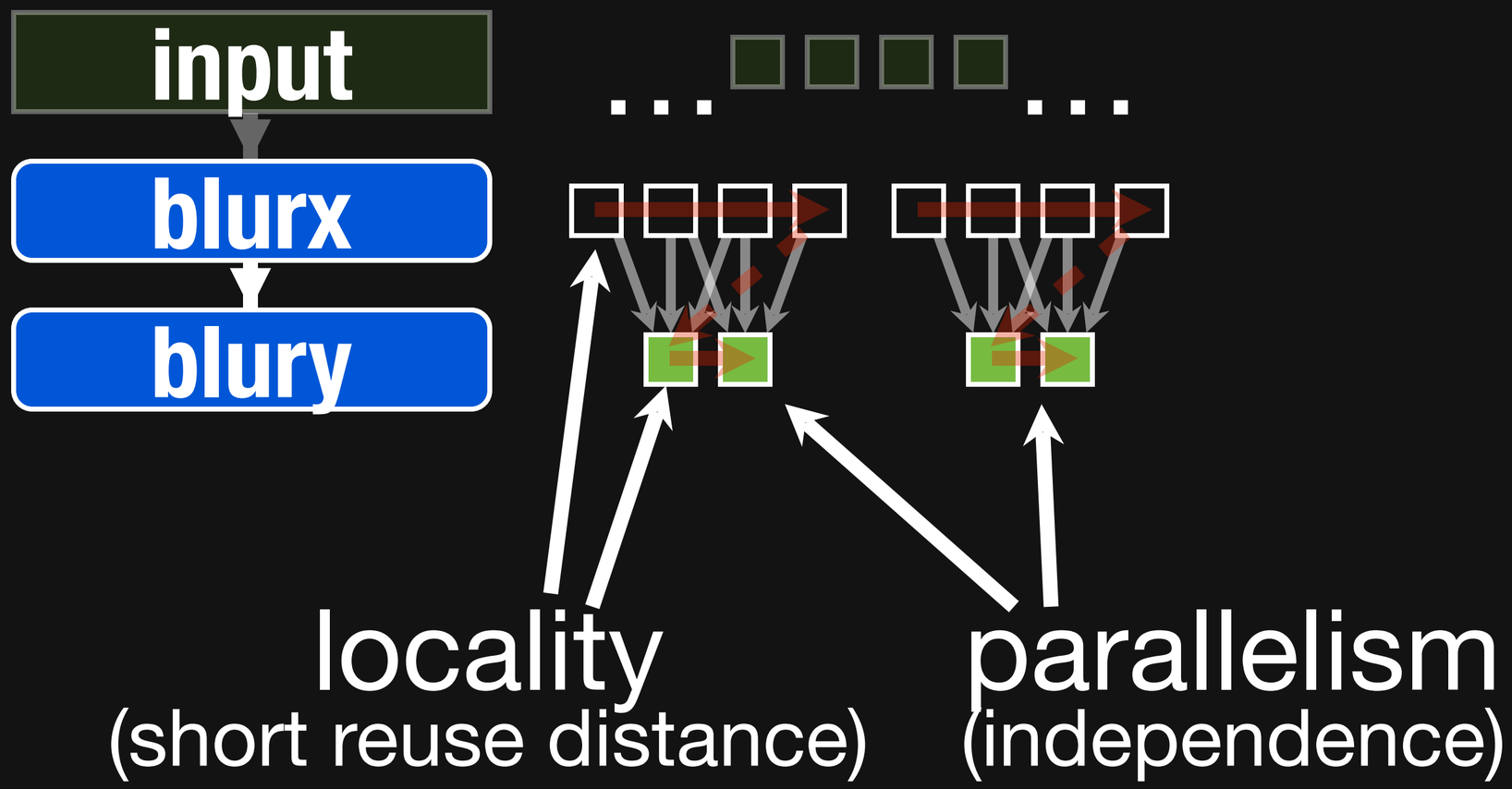
Decoupled tiles optimize parallelism & locality



Decoupled tiles optimize parallelism & locality



Decoupled tiles optimize parallelism & locality



... but at the cost of redundant work.

Halide

a new language & compiler for image processing

1. **Decouple *algorithm* from *schedule***

   Algorithm: *what* is computed

   Schedule: *where* and *when* it's computed

Halide

a new language & compiler for image processing

1. **Decouple *algorithm* from *schedule***

   Algorithm: *what* is computed

   Schedule: *where* and *when* it's computed

2. **Single, unified model for *all* schedules**

Halide

a new language & compiler for image processing

1. **Decouple *algorithm* from *schedule***

   Algorithm: *what* is computed
   Schedule: *where* and *when* it's computed

2. **Single, unified model for *all* schedules**

   Simple enough to search, expose to user
   Powerful enough to beat expert-tuned code

Aside: Halide

(a) Clean C++ : 9.94 ms per megapixel

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

(b) Fast C++ (for x86) : 0.90 ms per megapixel

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

(c) Halide : 0.90 ms per megapixel

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

 // The algorithm
 tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

 // The schedule
 blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
 tmp.chunk(x).vectorize(x, 8);

 return blurred;
}
```

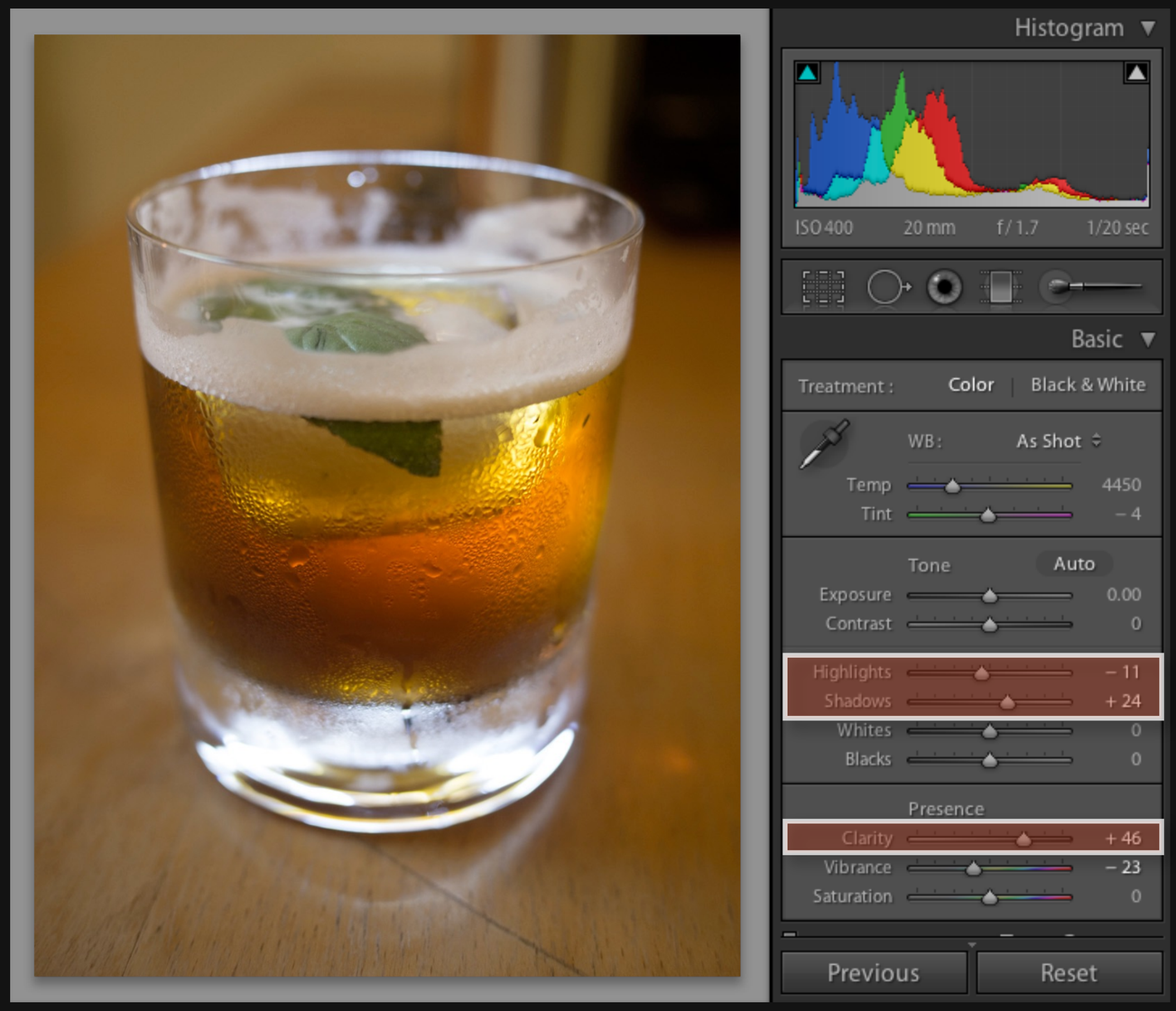
Local Laplacian Filters

Prototype for Adobe Photoshop Camera Raw / Lightroom

Adobe: 1500 lines of
expert-optimized C++
multithreaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 intern-day

Halide vs. Adobe:
2x faster on same CPU,
9x faster on GPU



Aside: Halide

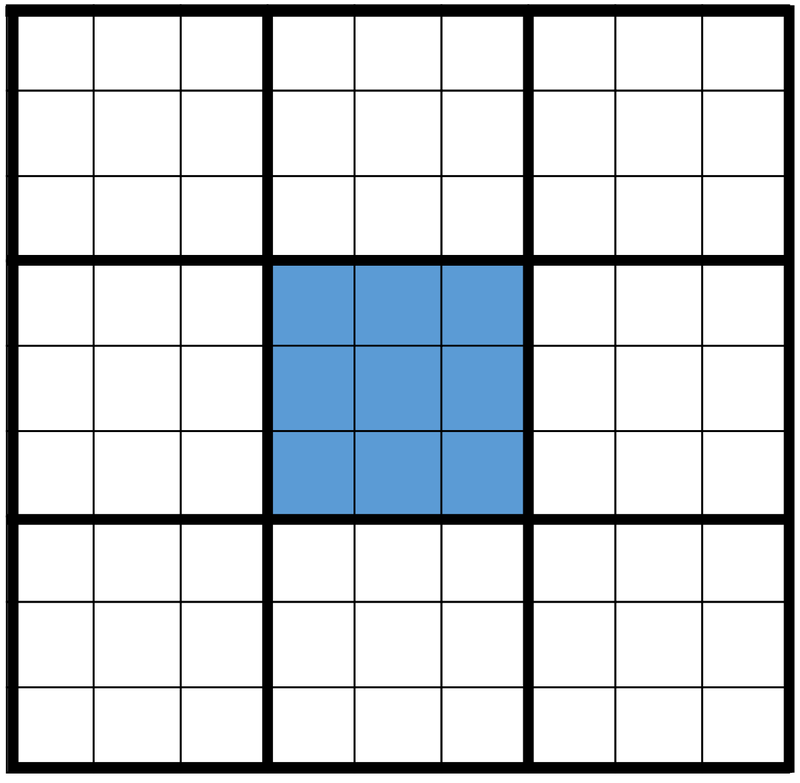
halide-lang.org

J. Ragan-Kelly, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines.” ACM Transactions on Graphics 31(4) (In Proceedings of SIGGRAPH 2012).

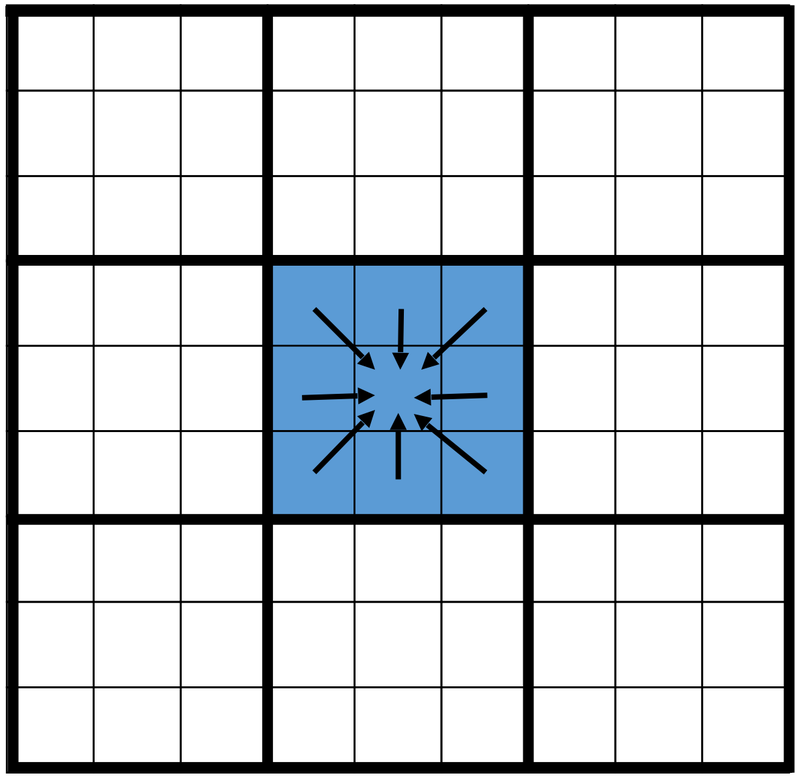
Third Option: Use tiling (with shared memory) to reduce redundant computation

- Read values into shared memory
- Local sync
- Compute x-blur on the shared memory values
- Local sync
- Use x-blur to compute y-blur

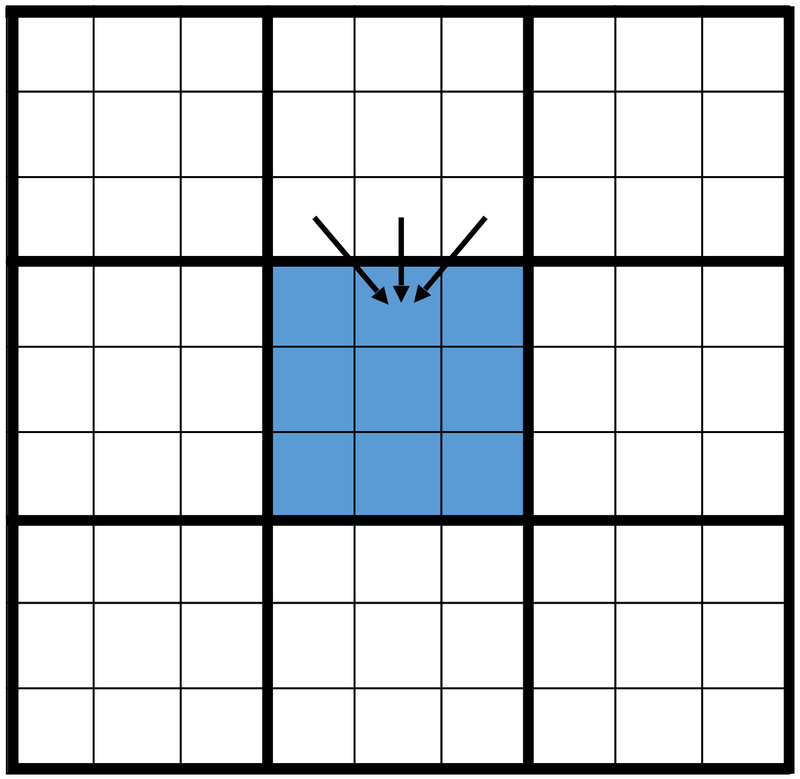
Blur with Tiling



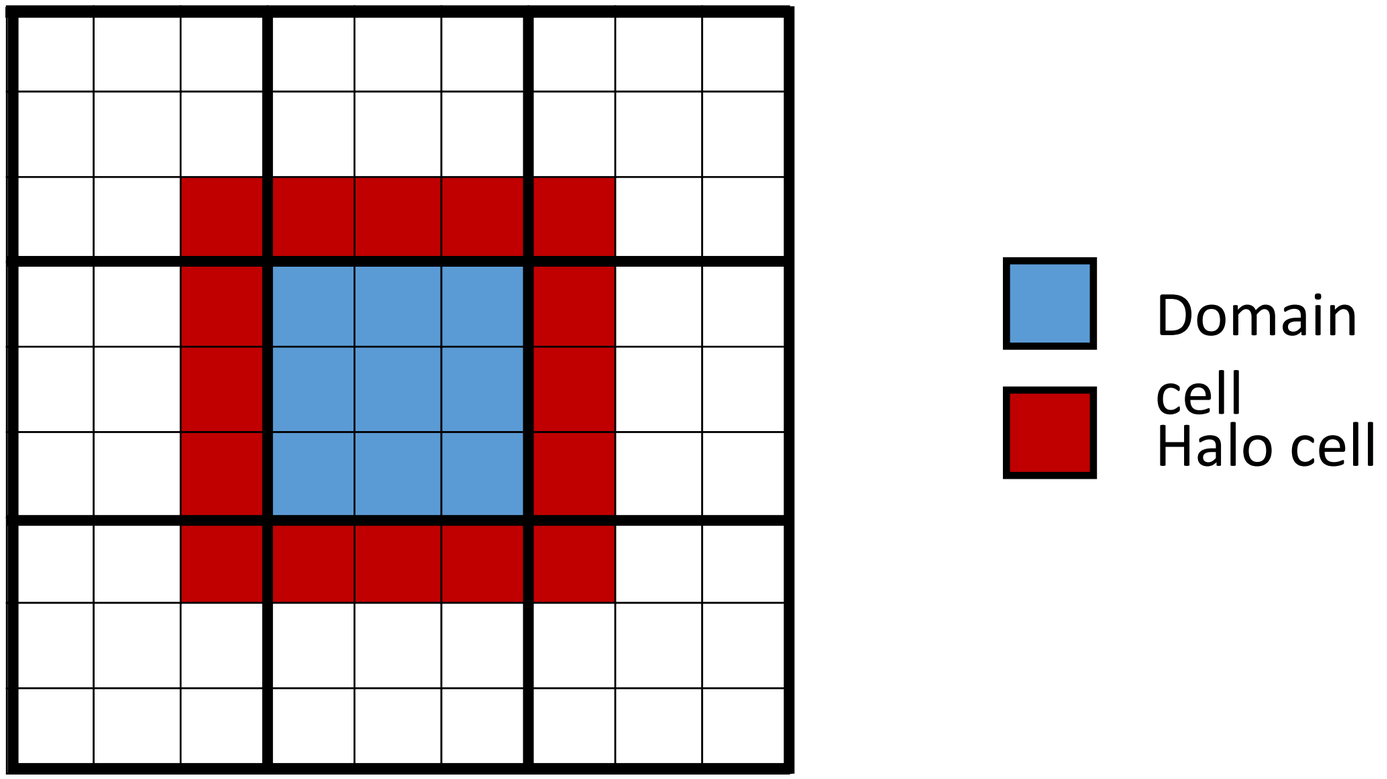
Blur with Tiling



Blur with Tiling



Halo/Ghost Region



Halo/Ghost Region

- A thread block needs data that “belongs” to a different block
- Read that extra data – known as “halo” or “ghost” region
  - Known as “overfetch”
- Some algorithms also require redundant computation
  - E.g., separable blur
  - Avoids communication

Tuning is required for best performance

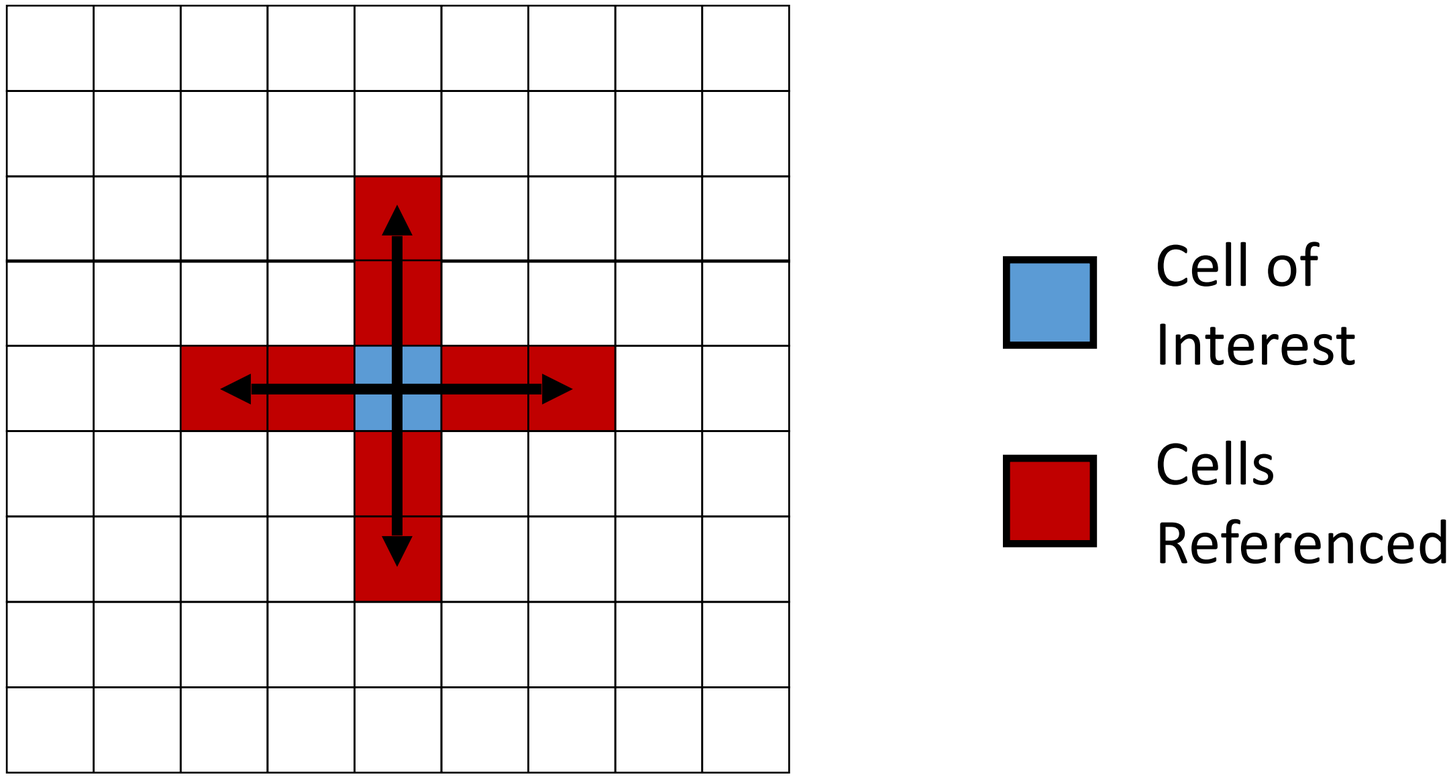
- Larger blocks = less overfetch (and fewer redundant computations)
- Smaller blocks = more parallelism across blocks (more latency hiding)
  less parallelism within a block
- Trade-off between data reuse and parallelism
  - Depends on the algorithm, data layout, and hardware specifics

Two types of locality

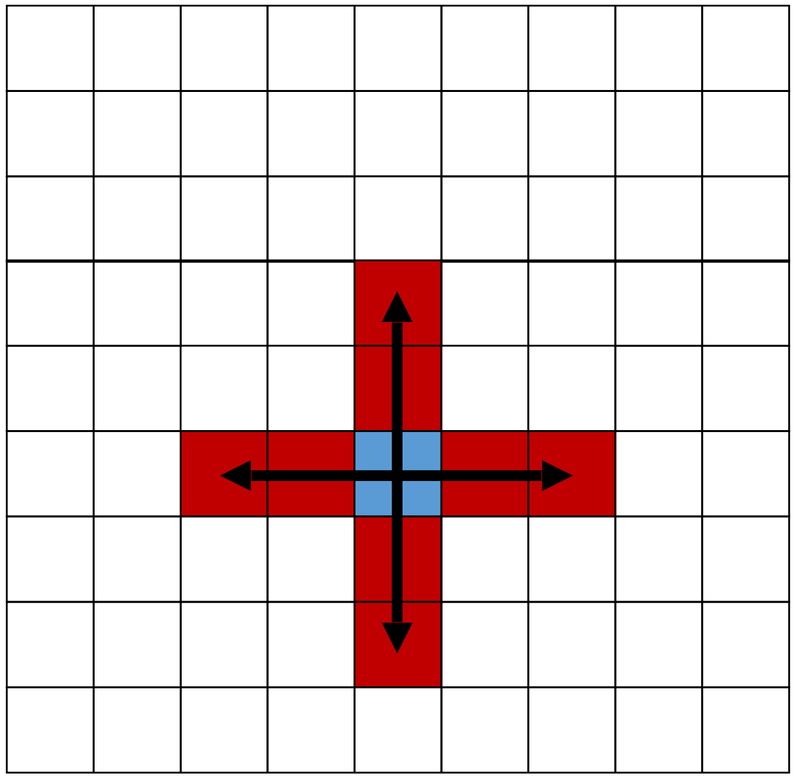
- Spatial locality
  - Elements stored near each other should be used at the same time
  - (coalescing)

- Temporal locality
  - Elements calculated recently should be reused by subsequent calculations

Spatial / temporal Locality



Spatial / temporal Locality

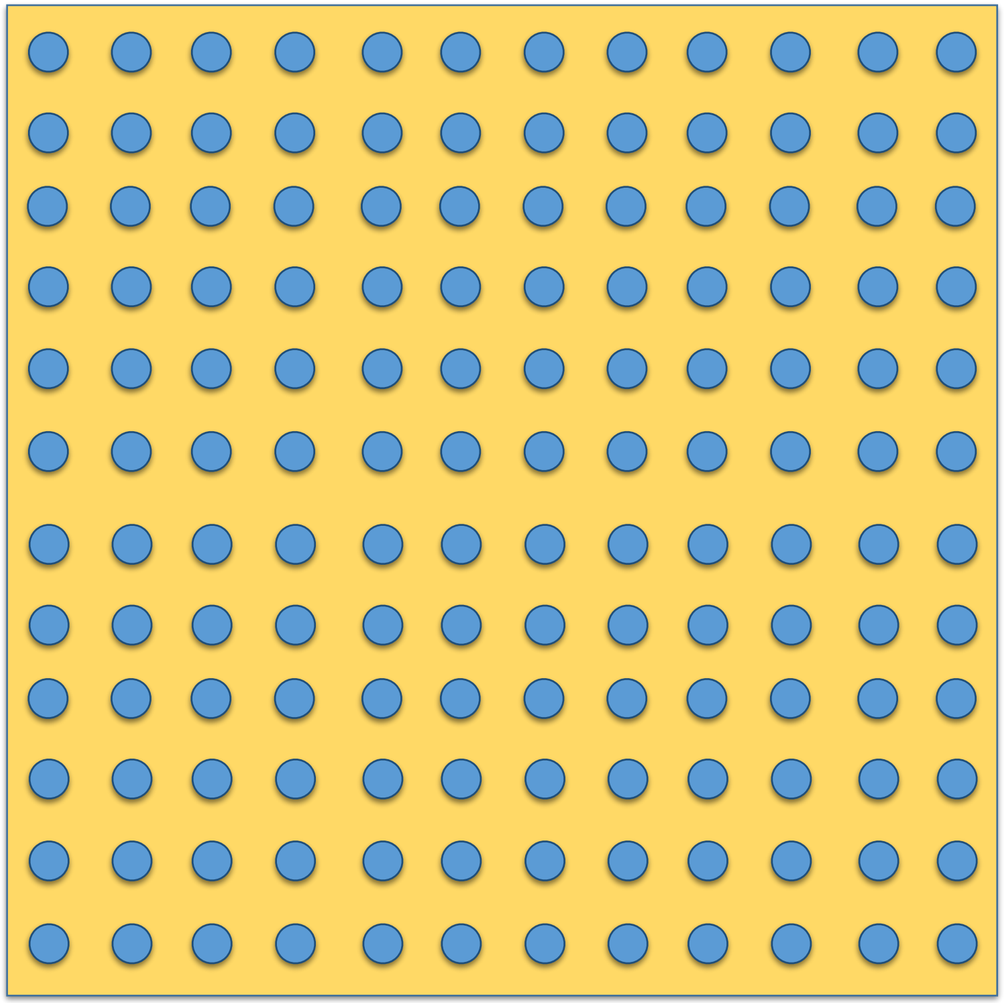


Cell of Interest

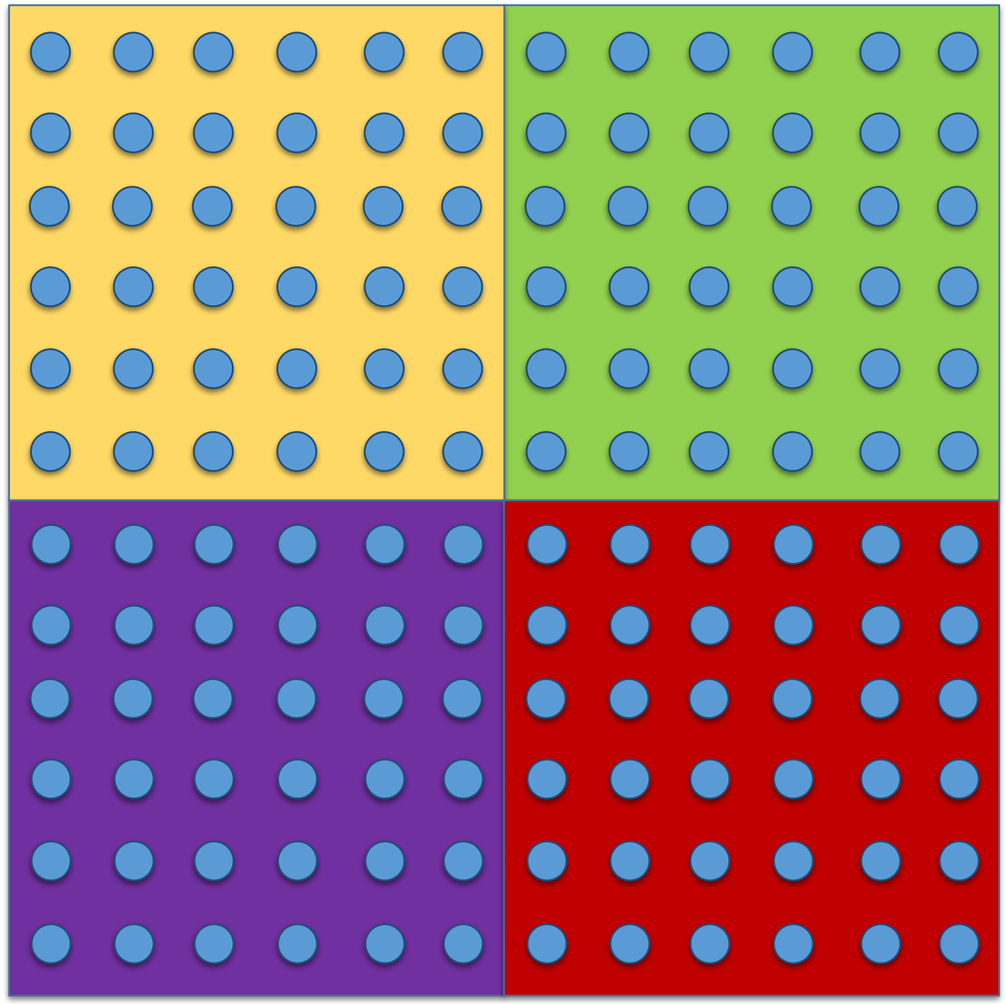
Cells Referenced

General Approach for Stencils

Divide grid into sub-blocks



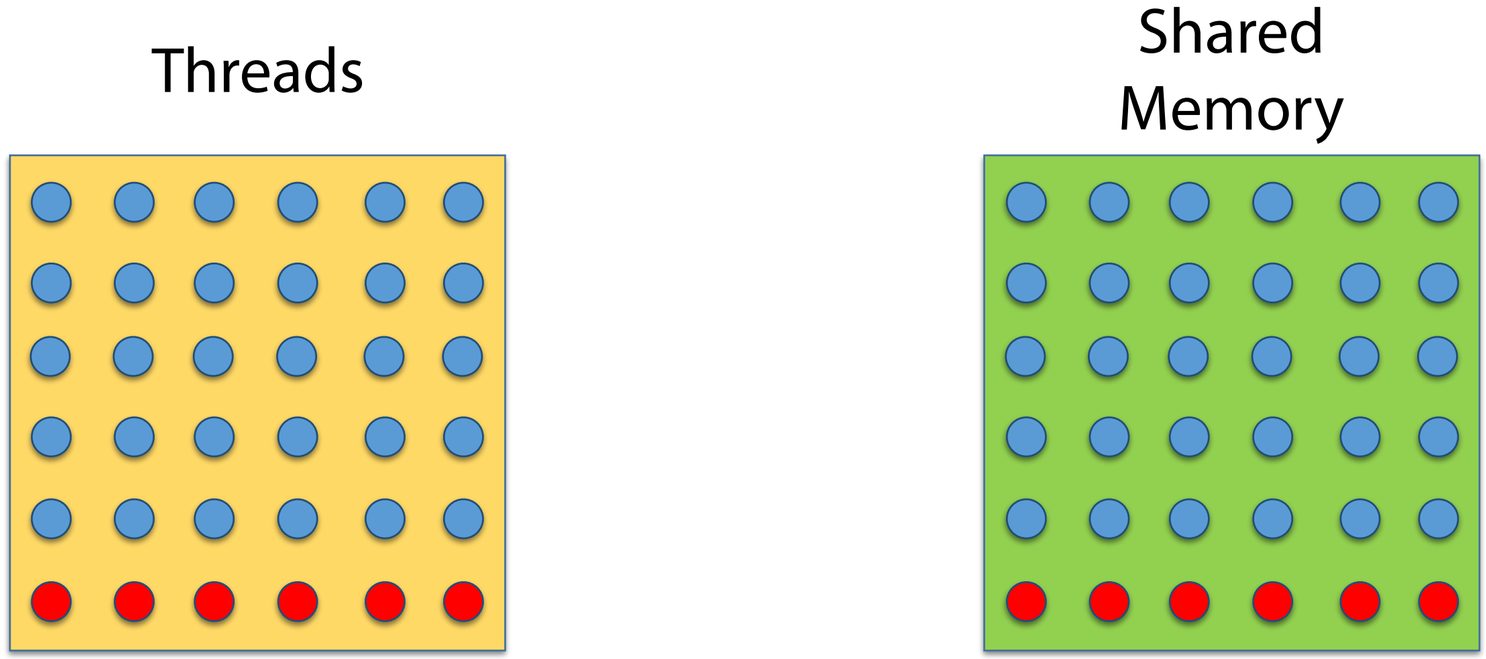
Divide grid into sub-blocks



Sub-block implementation

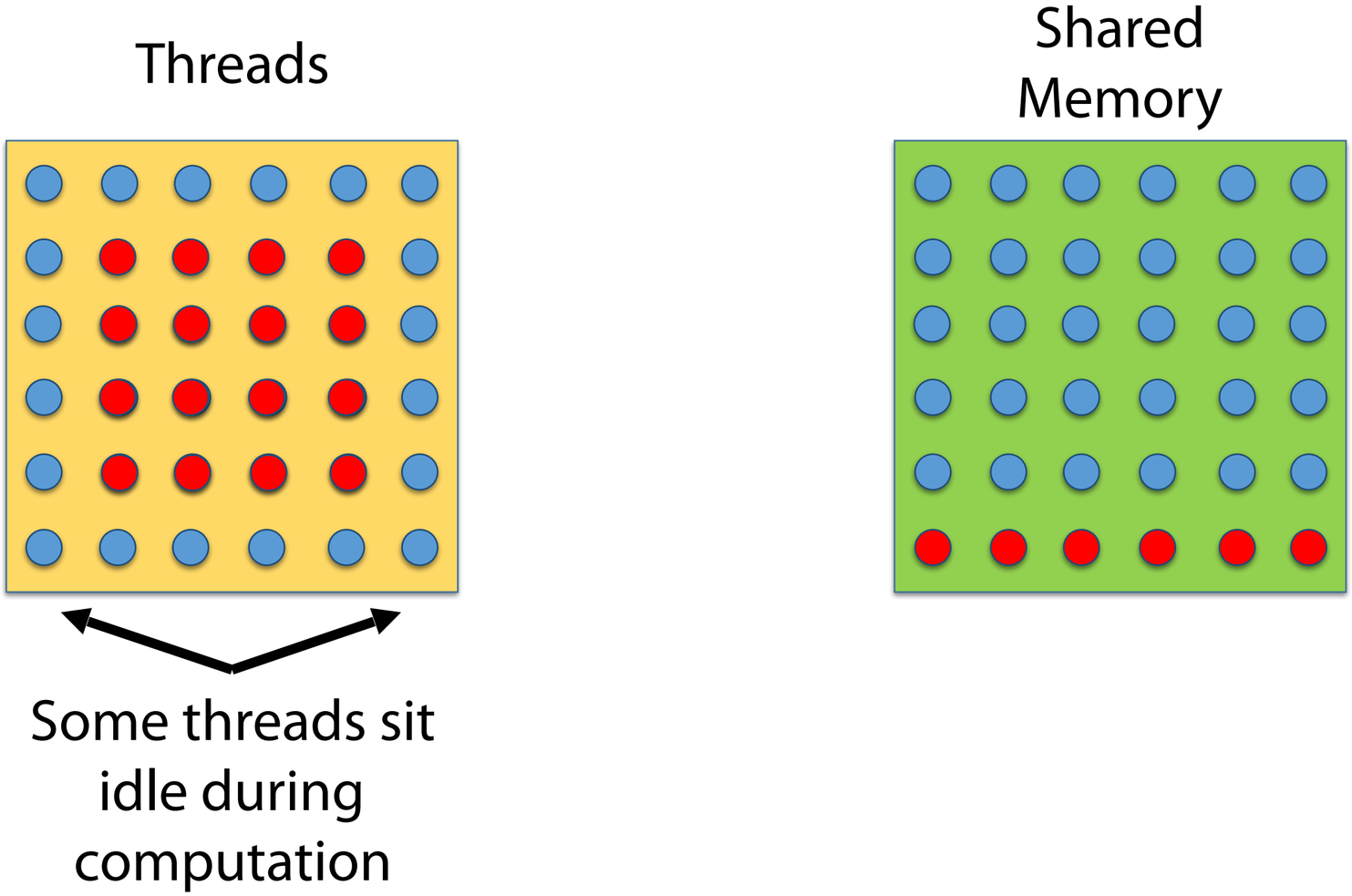
- Assign each thread block a different sub-block
- Each thread in a block reads sub-block data into shared memory
  - Use extra threads for halo/ghost region
- Update nodes in sub-block and output to global memory

Problem: Idle threads during computation

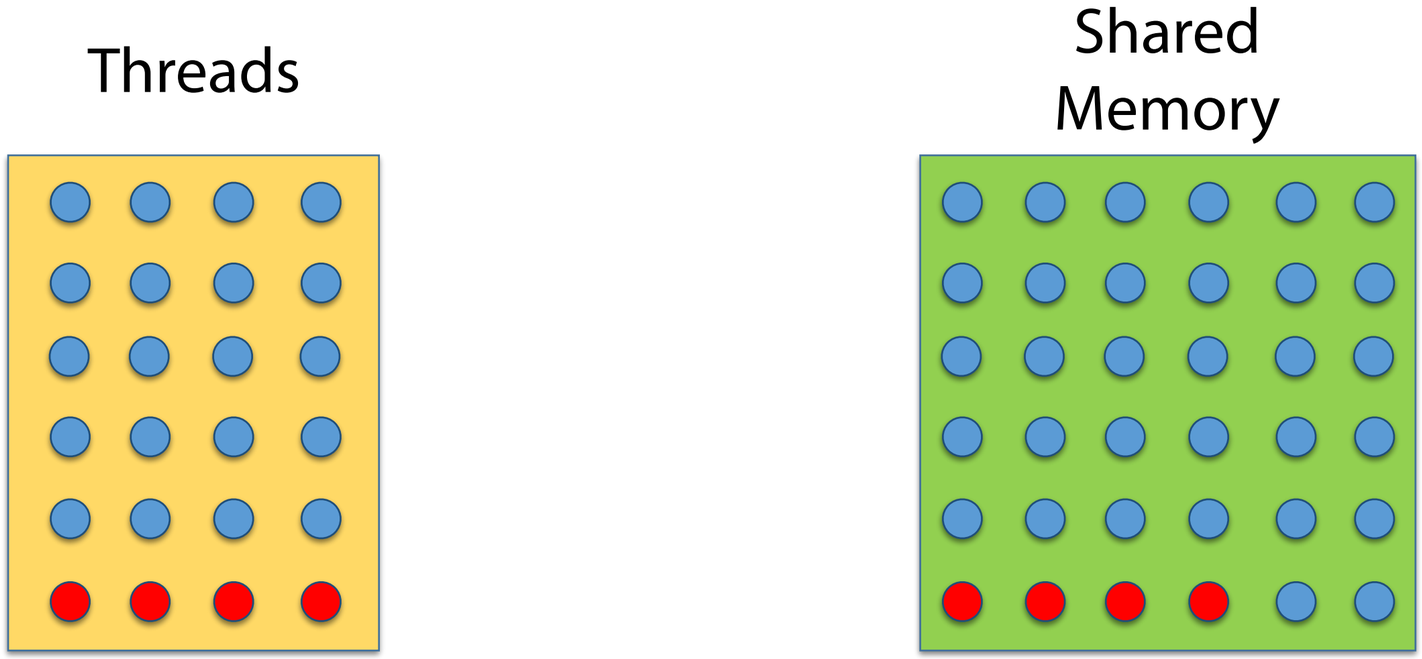


All threads active to read into shared memory

Problem: Idle threads during computation

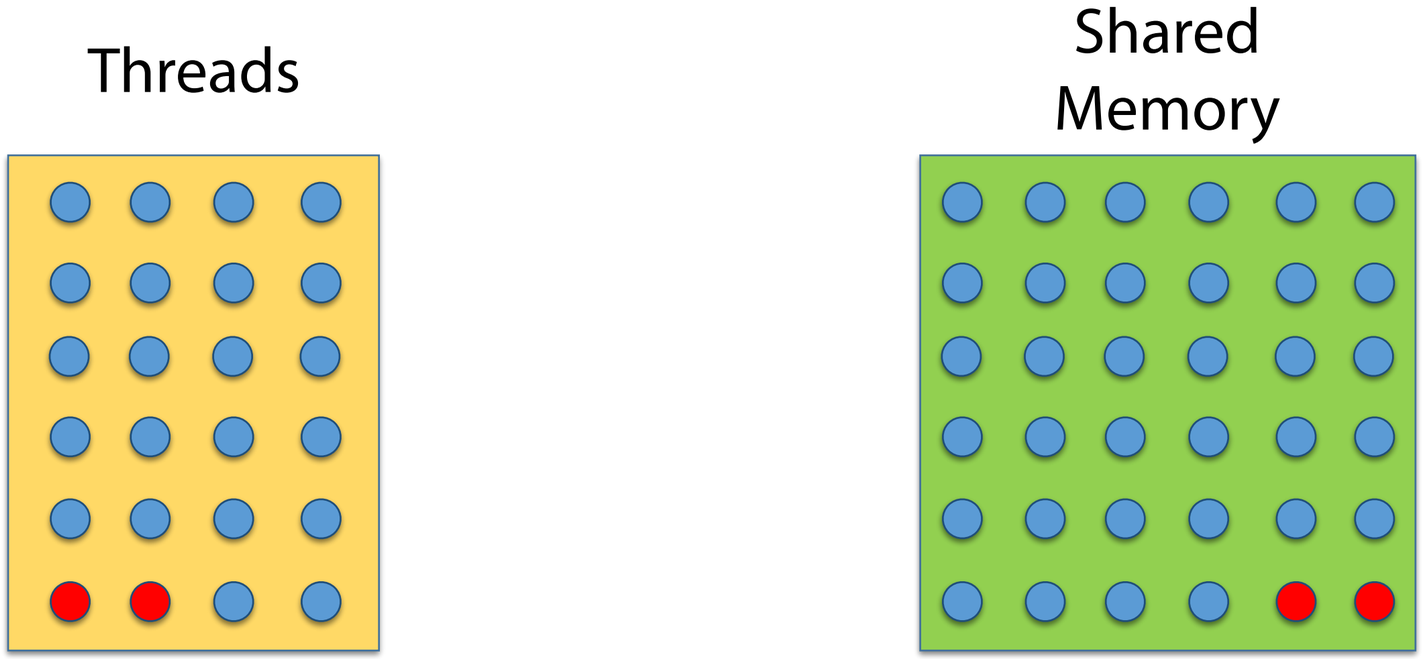


Optimization #1: No load-only threads



Launch fewer threads:
1st load – all threads read into Smem

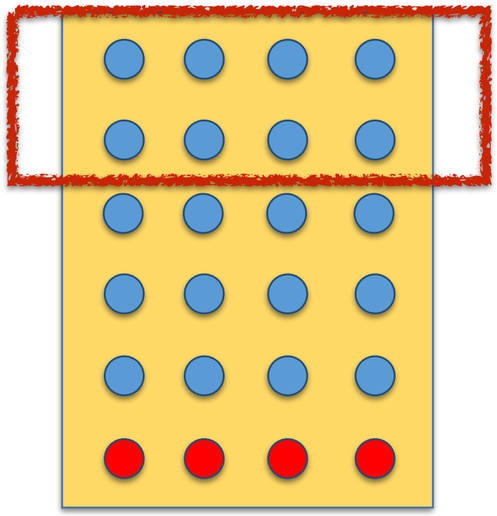
Optimization #1: No load-only threads

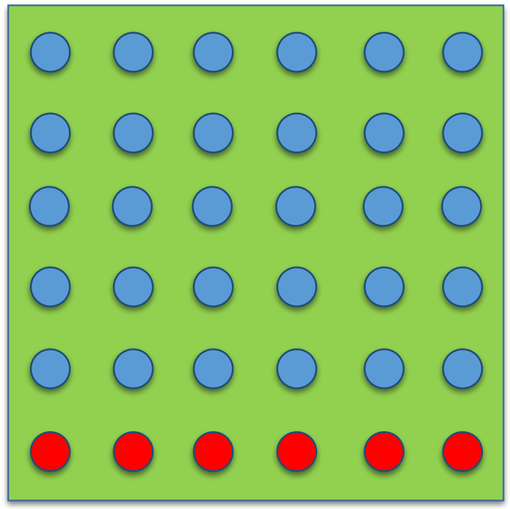


Launch fewer threads:
1st load – all threads read into Smem
2nd load – two threads read into Smem

Optimization #1: No load-only threads

Do you need these?





Launch fewer threads:
1st load – all threads read into Smem
2nd load – two threads read into Smem
All threads active during computation

Problem: Limited memory size

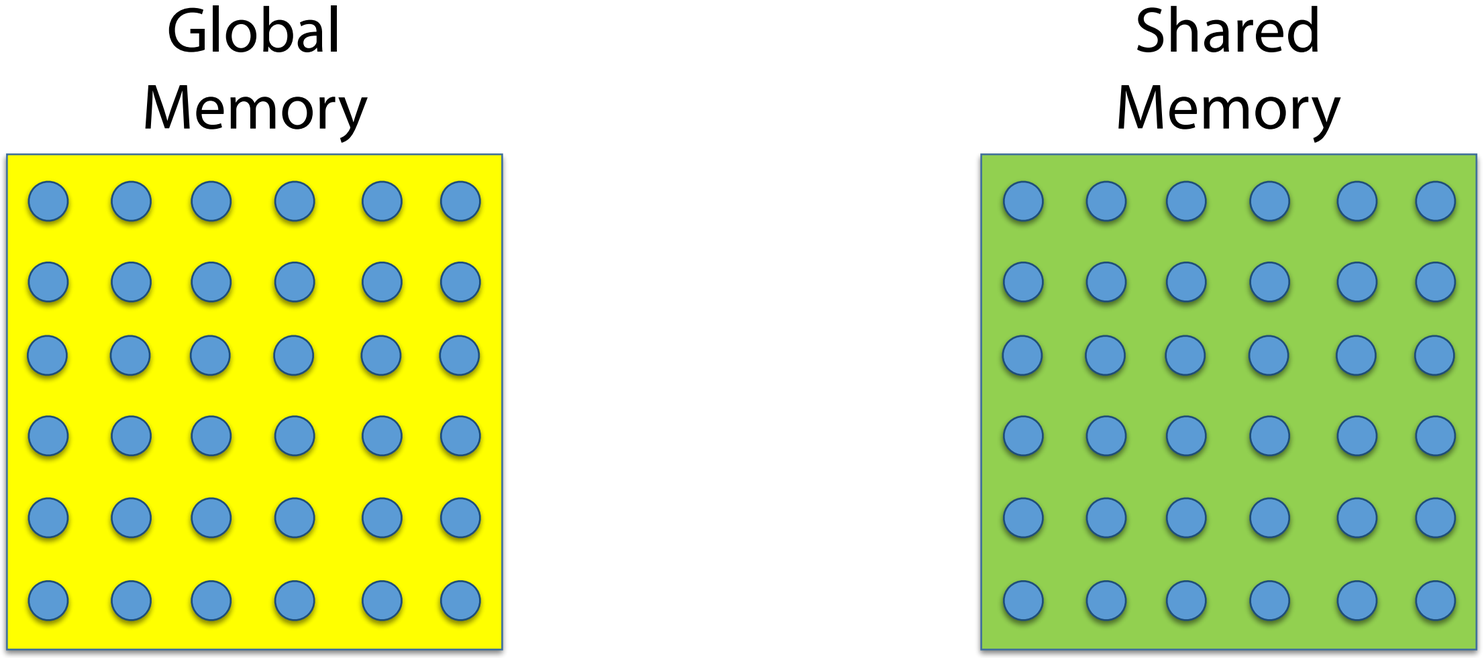
- Real applications often have many stencils, inputs, and outputs
- More data -> more shared memory per block -> fewer blocks per SM -> less latency hiding potential
- E.g.,
  - Inputs: A, B, C, D, E
  - Outputs: Y1, Y2, Y3
  - Stencils: f1, f2, f3, f4
  - $Y1 = f1(A, B, C) + f2(D, E)$
  - $Y2 = Y1 + f3(B, C, D)$
  - $Y3 = Y1 + Y2 + f4(A,E)$

Optimization #2: Use other memory resources

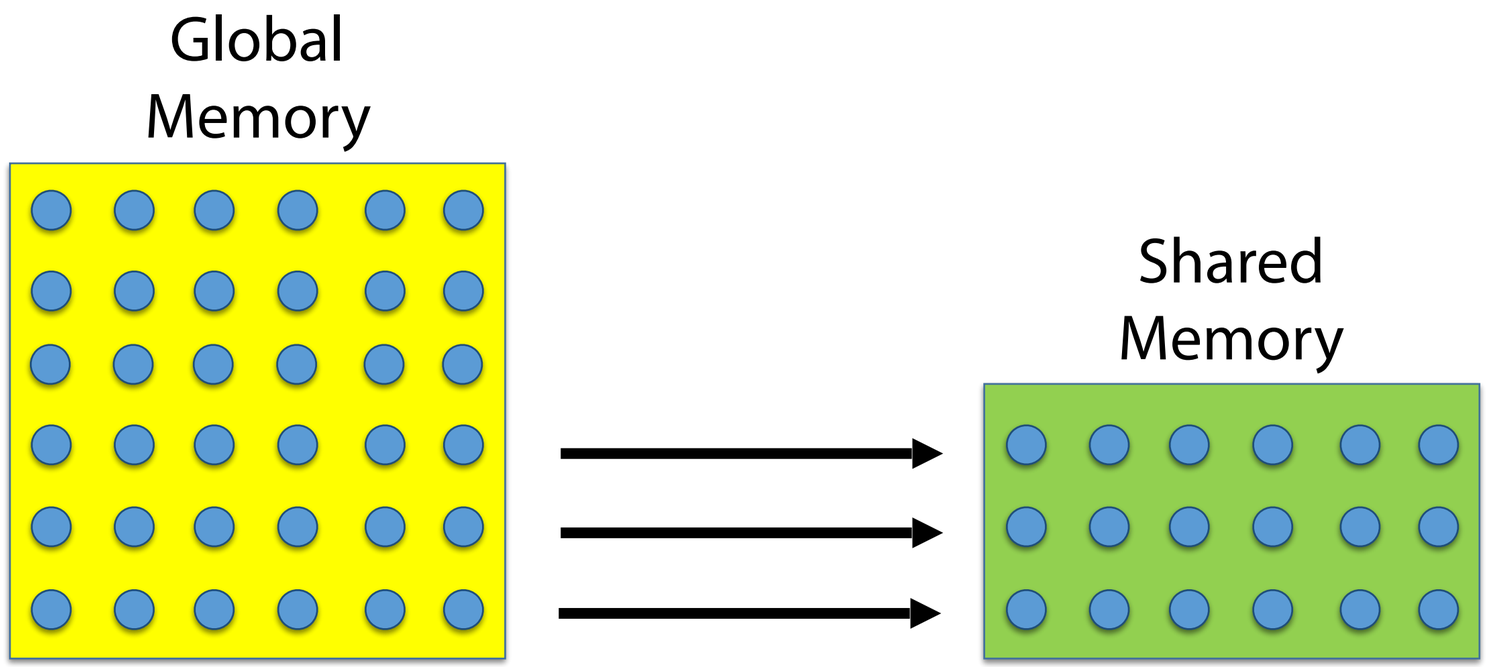
- Put some data into registers (more on this later)

- Store some inputs in texture memory
  - Use data directly from texture memory (don't load it into shared memory)
  - Optimized for 2D spatial locality
  - Faster than global memory for access patterns that exploit 2D spatial locality

Problem: Shared memory is limited, but we want to maximize reuse

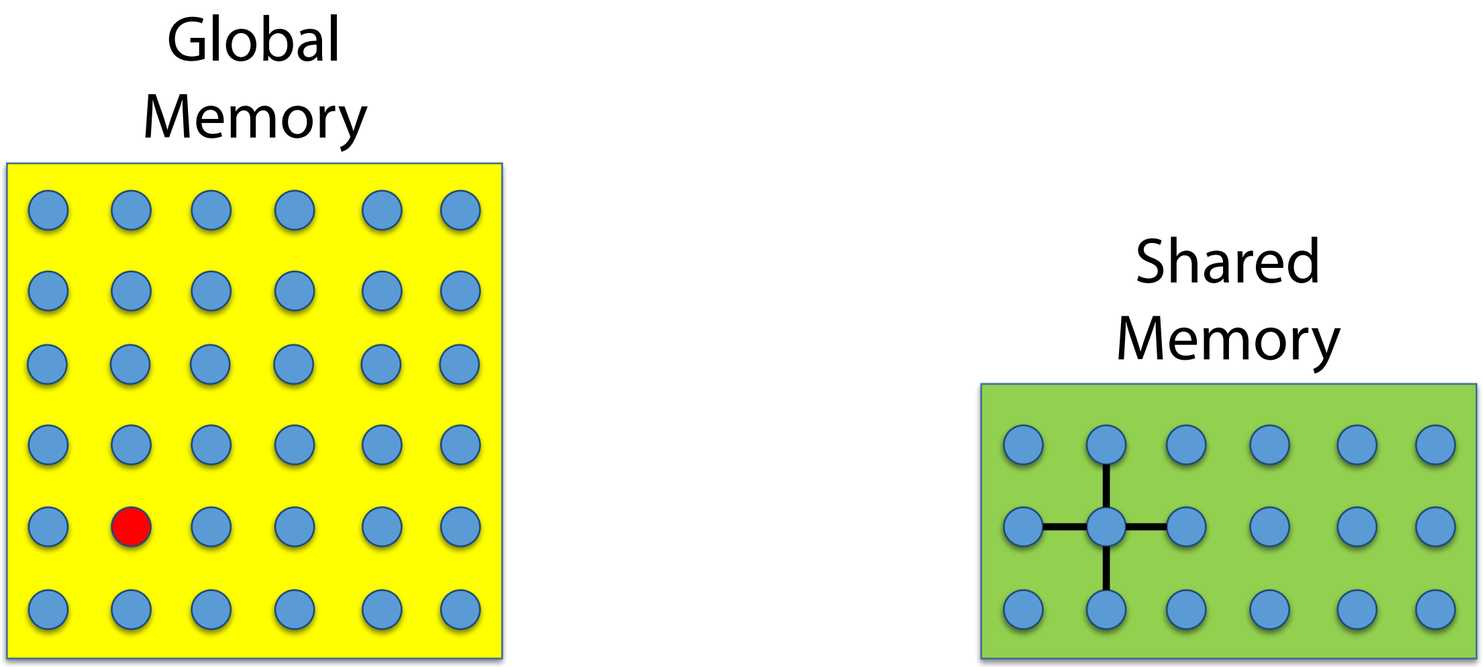


Optimization #3: March through domain plane-by-plane



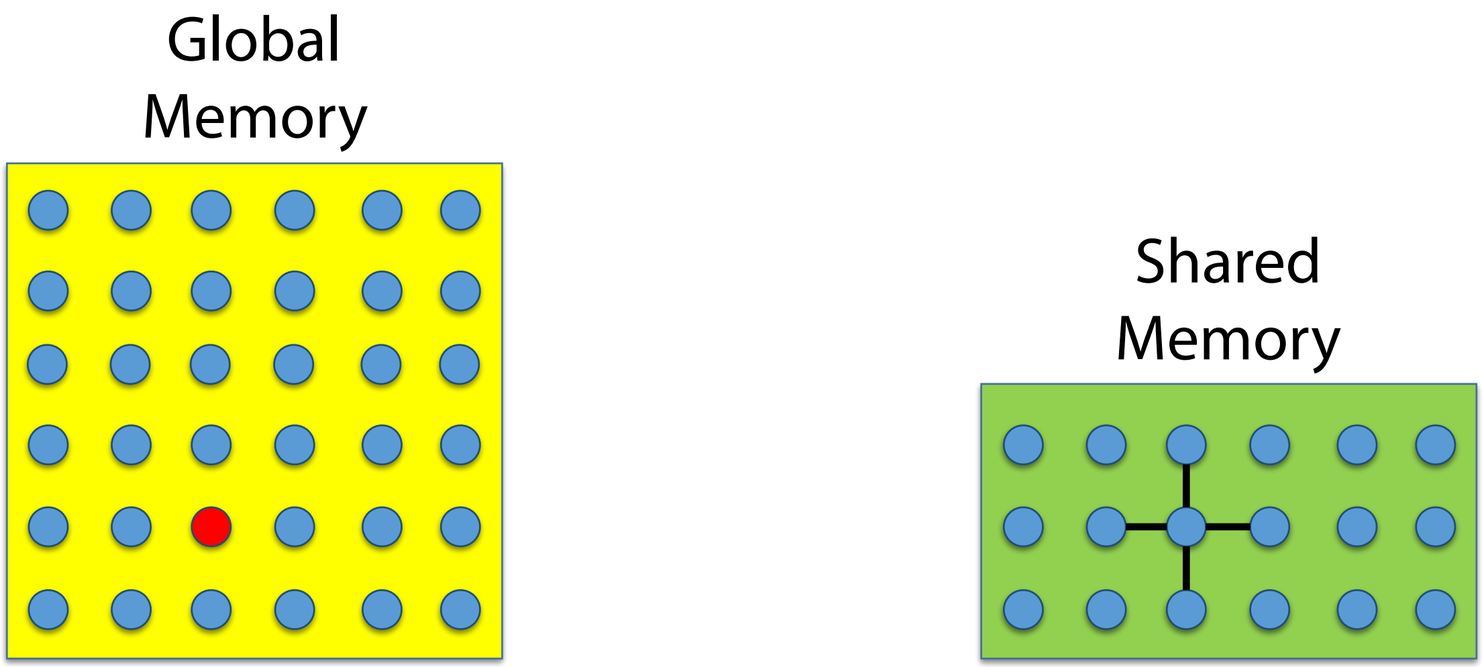
Fill shared memory array

Optimization #3: March through domain plane-by-plane



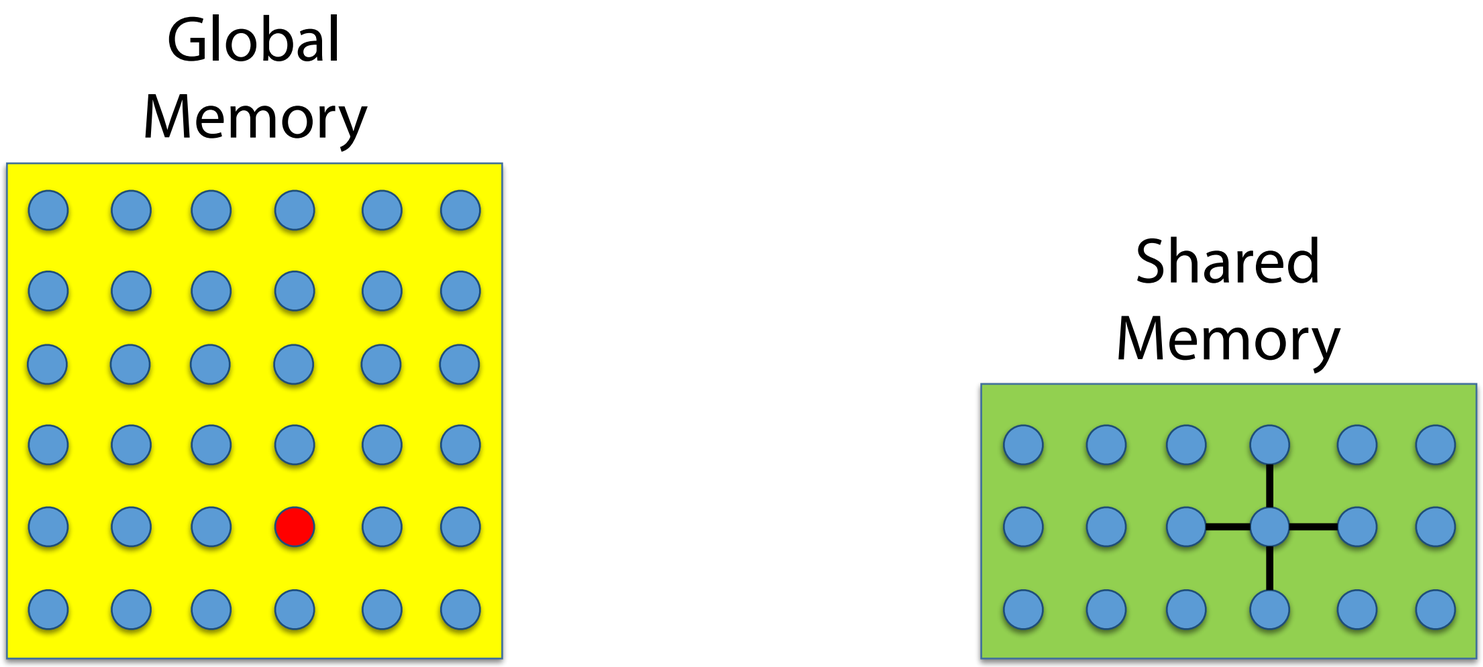
Evaluate stencil and store result in global memory

Optimization #3: March through domain plane-by-plane



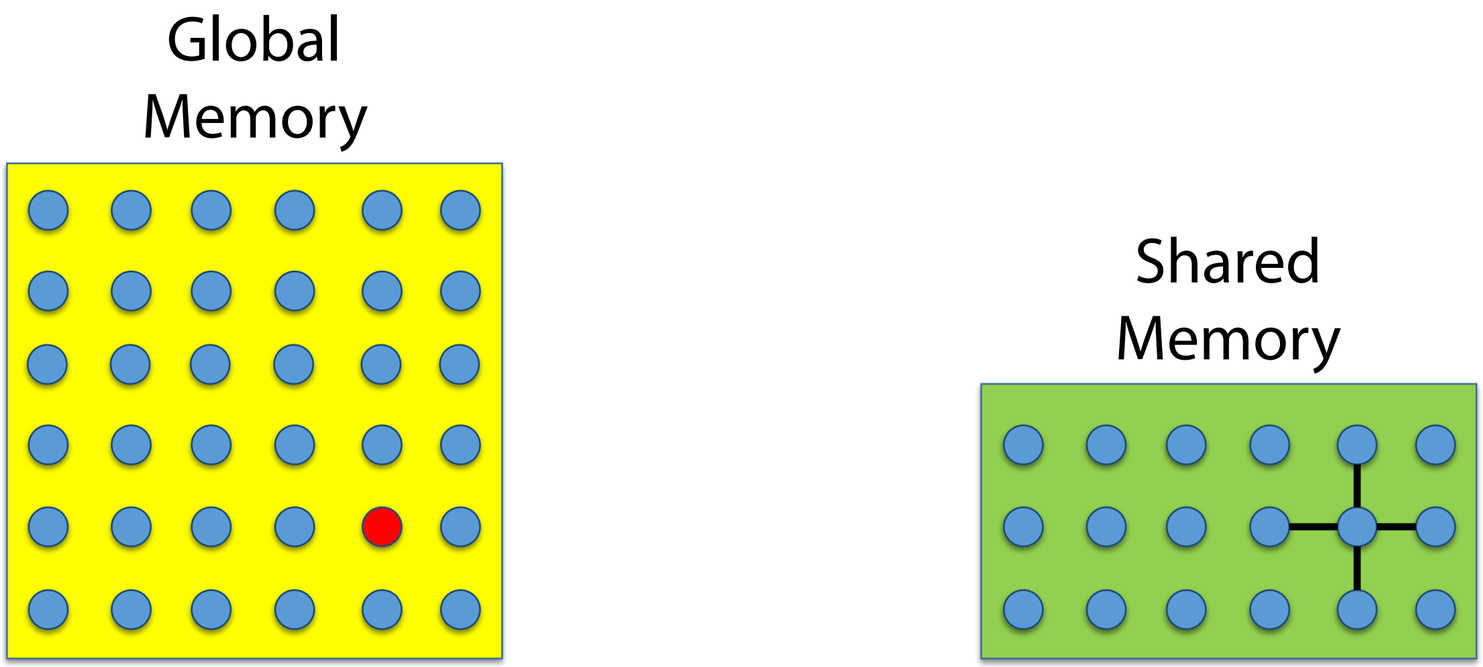
Evaluate stencil and store result in global memory

Optimization #3: March through domain plane-by-plane



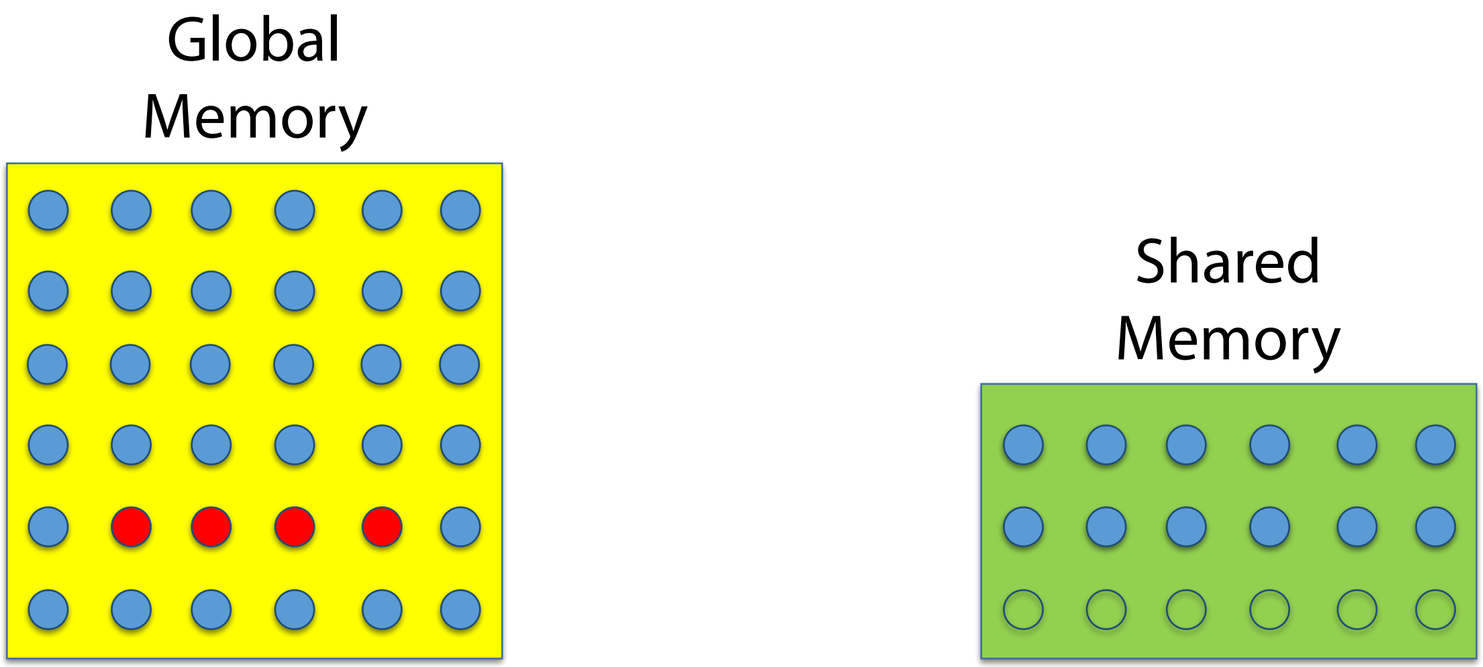
Evaluate stencil and store result in global memory

Optimization #3: March through domain plane-by-plane



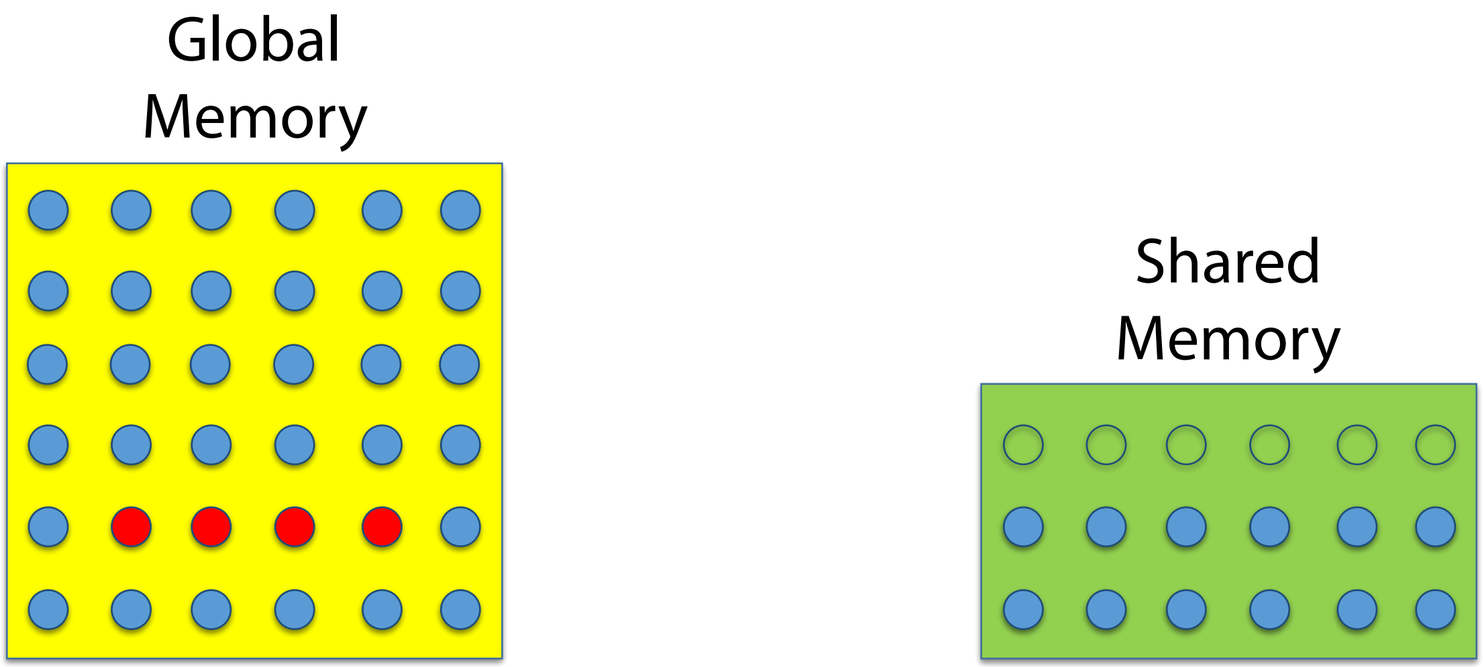
Evaluate stencil and store result in global memory

Optimization #3: March through domain plane-by-plane



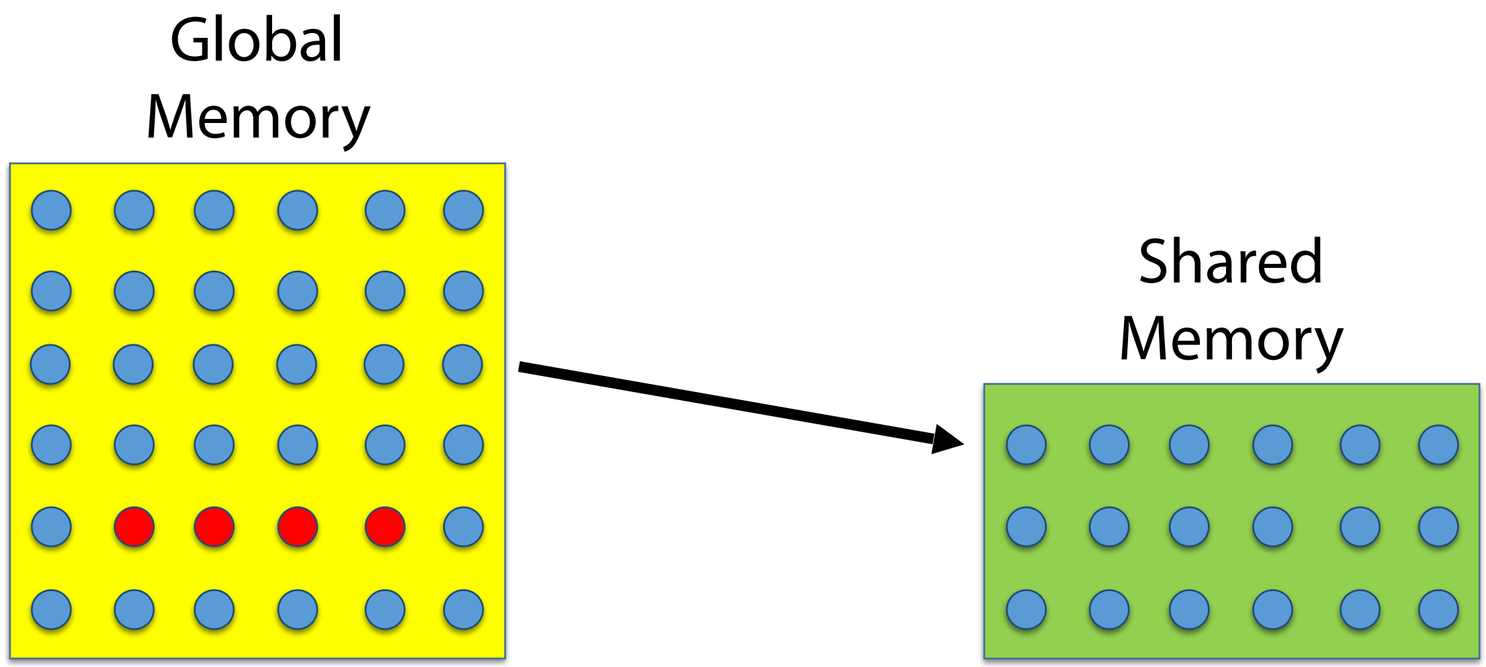
Load next row into shared memory

Optimization #3: March through domain plane-by-plane



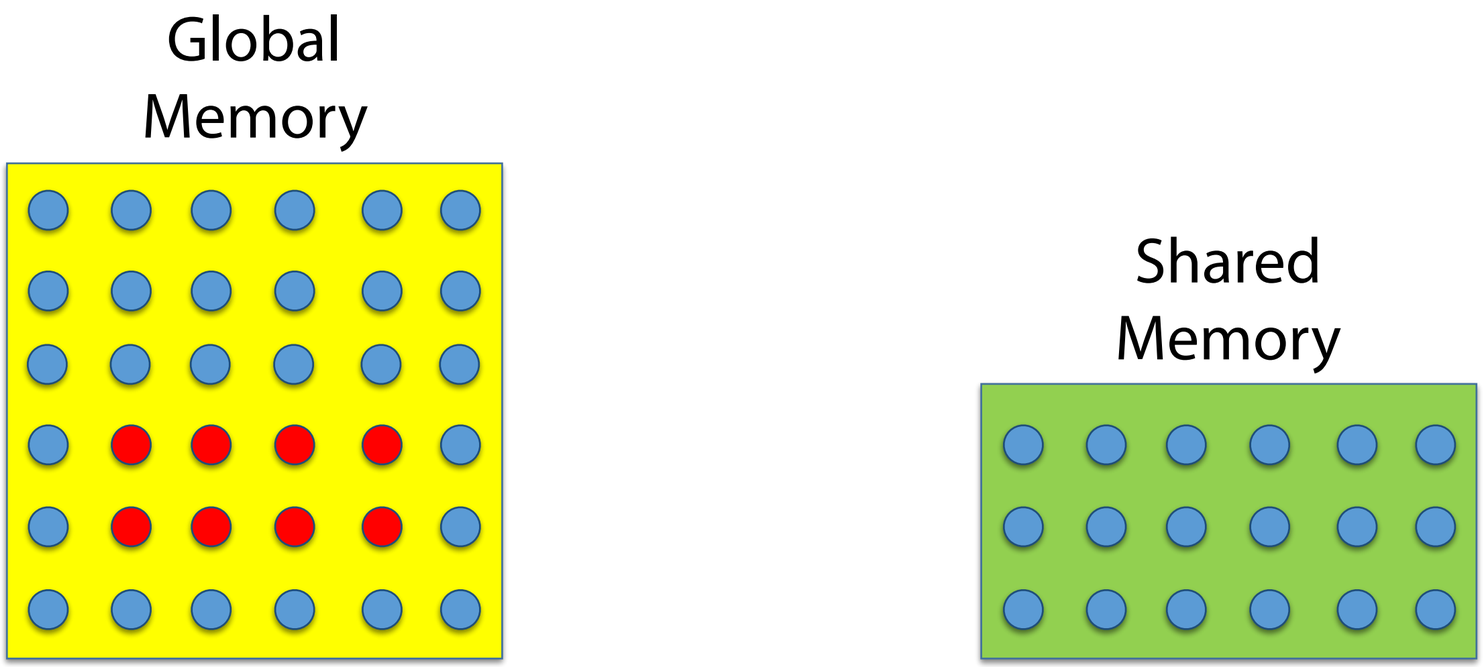
Load next row into shared memory

Optimization #3: March through domain plane-by-plane



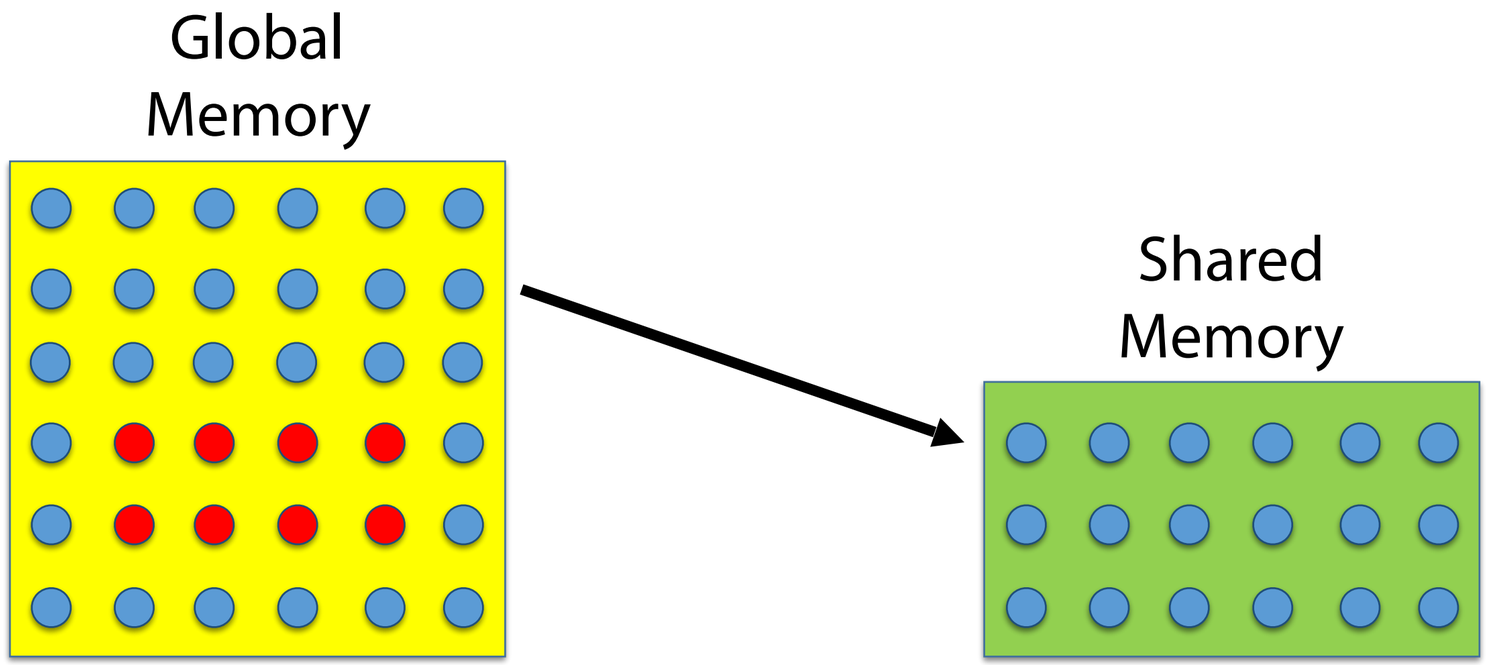
Load next row into shared memory

Optimization #3: March through domain plane-by-plane



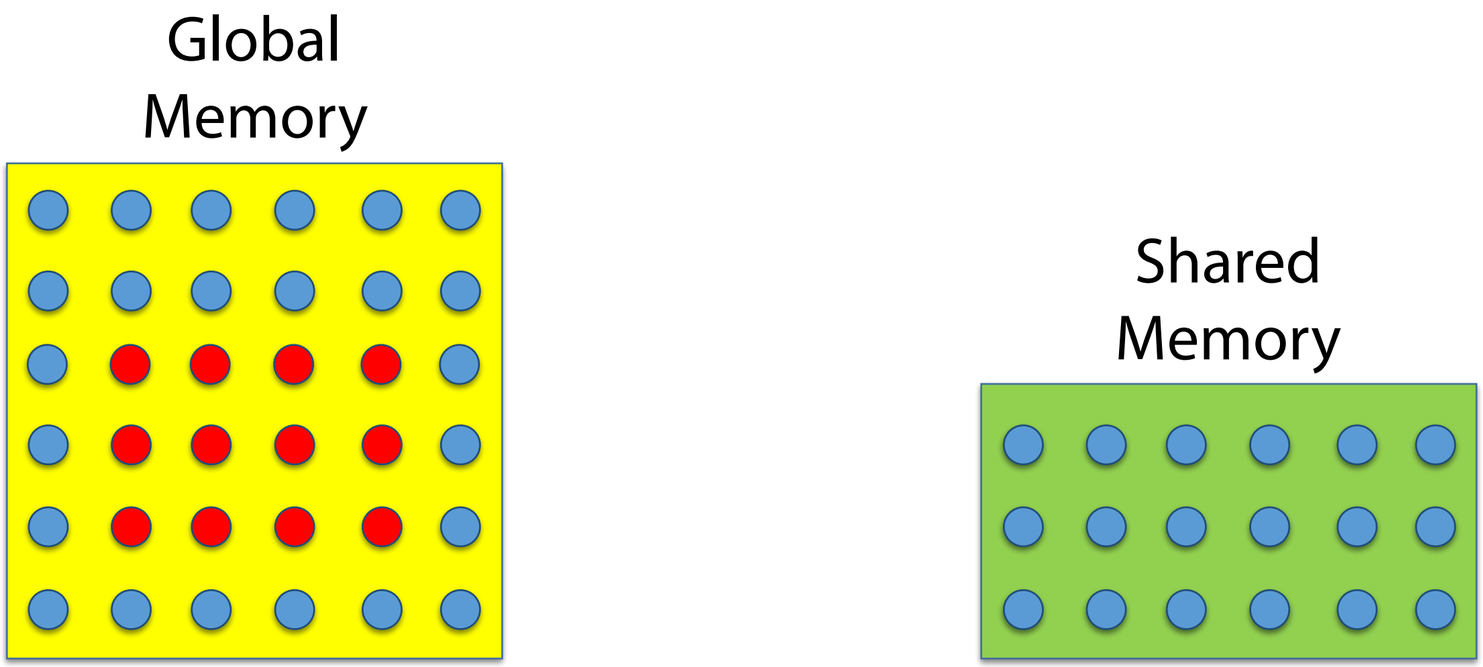
Load next row into shared memory

Optimization #3: March through domain plane-by-plane



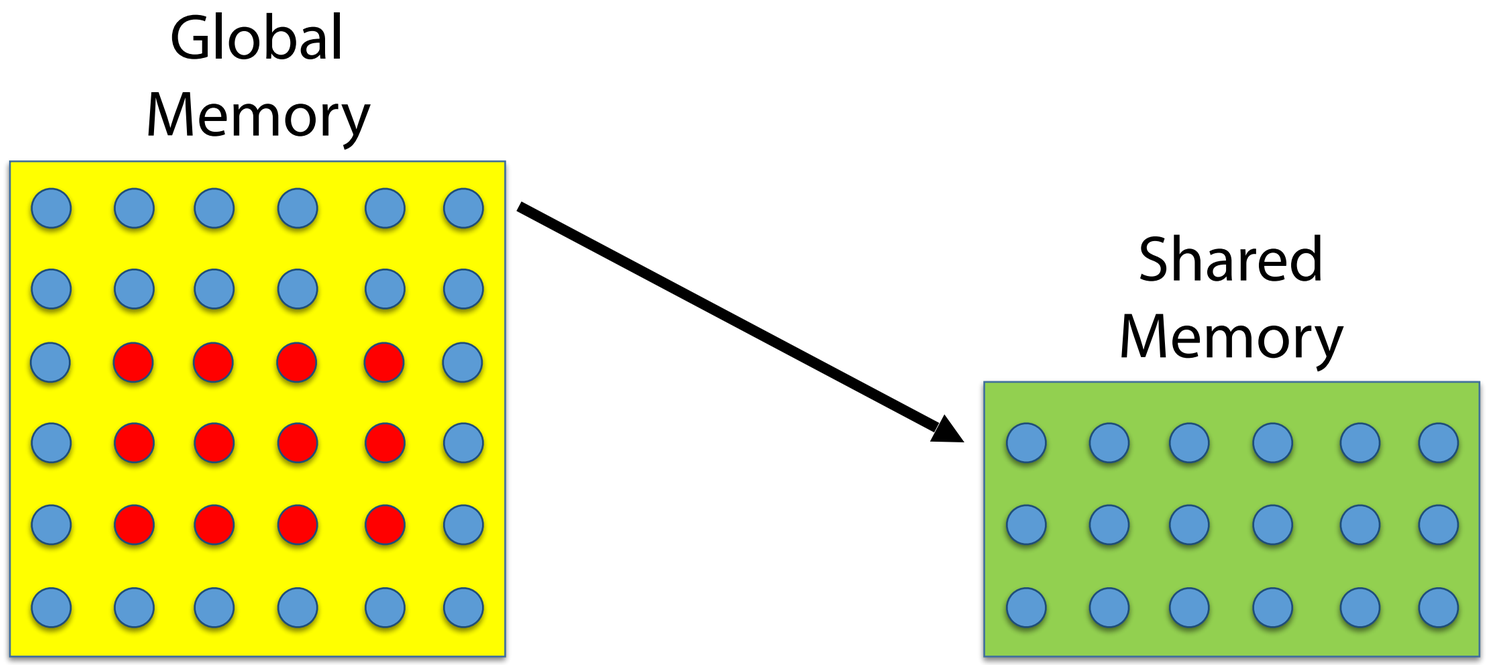
Load next row into shared memory

Optimization #3: March through domain plane-by-plane



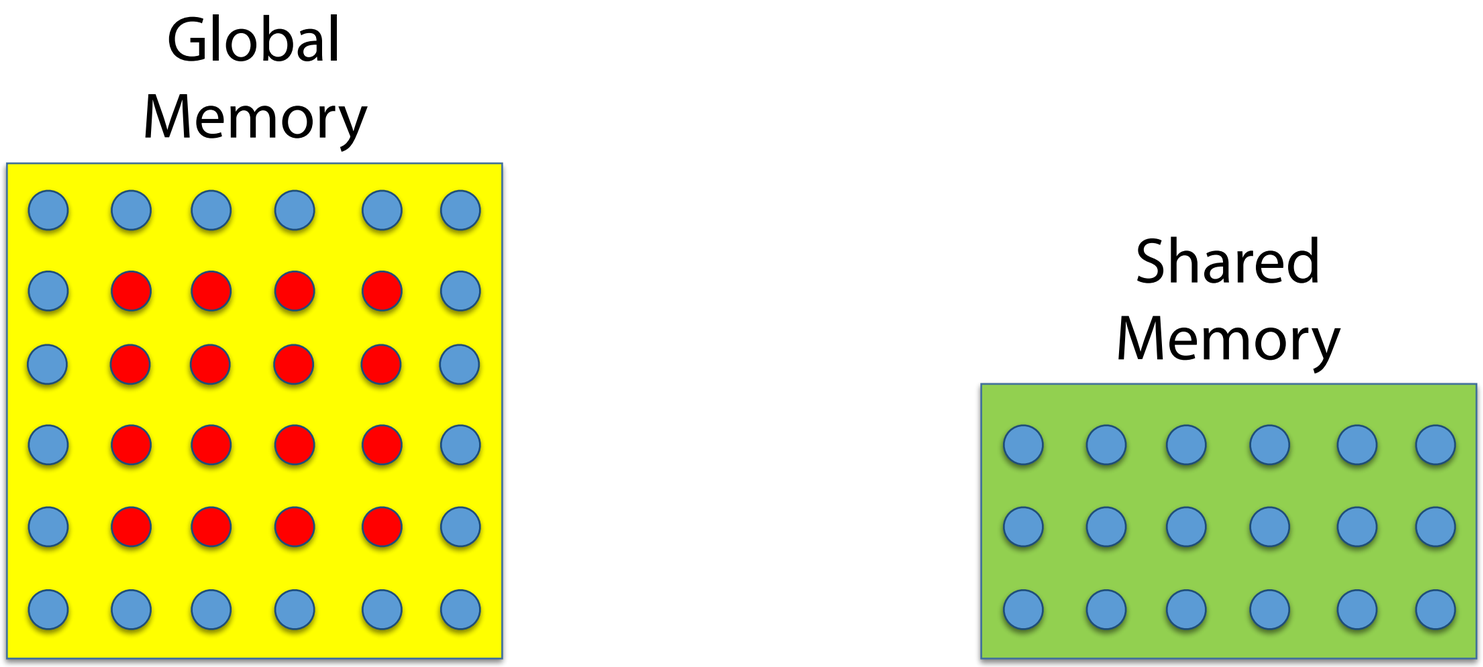
Load next row into shared memory

Optimization #3: March through domain plane-by-plane



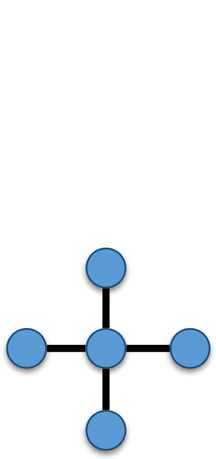
Load next row into shared memory

Optimization #3: March through domain plane-by-plane

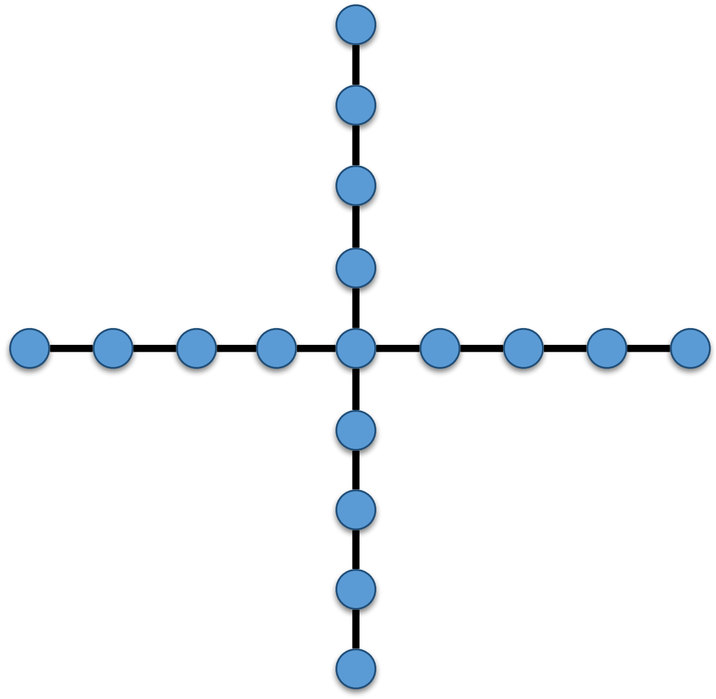


Load next row into shared memory

Optimization #4: High-order stencils

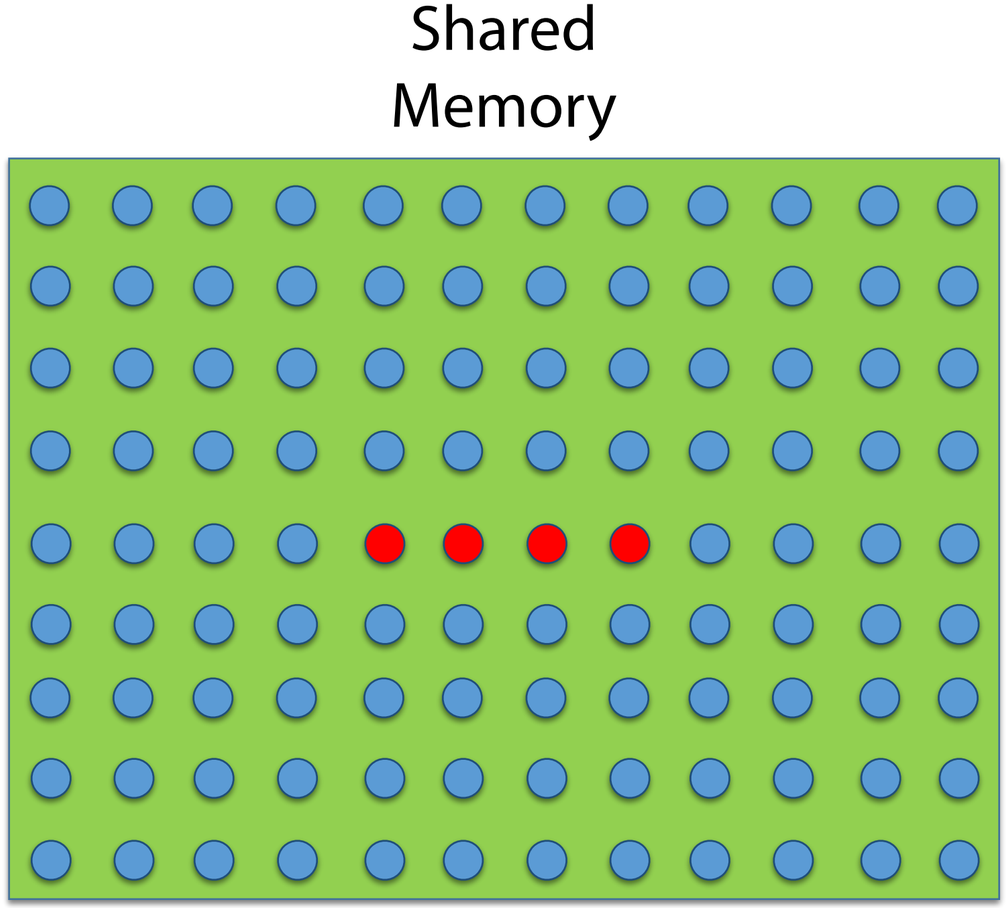


Second Order



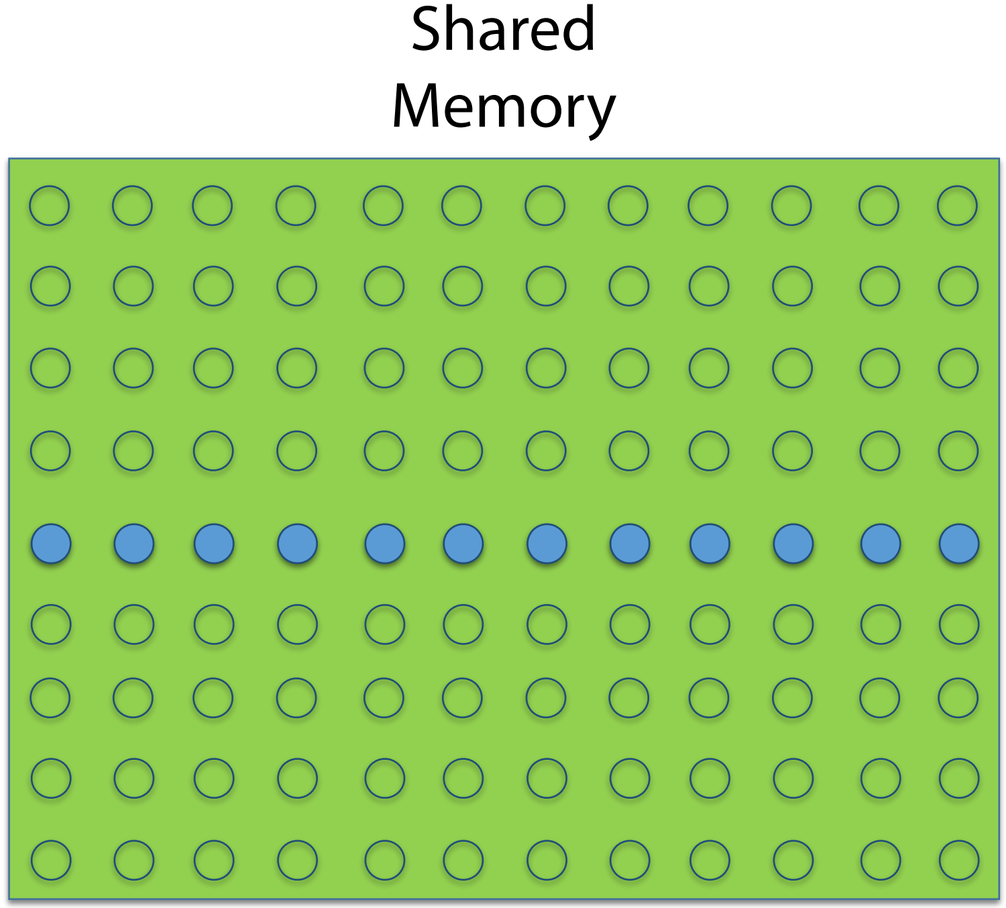
Eighth Order

Optimization #4: High-order stencils



Need 9 planes to compute 1

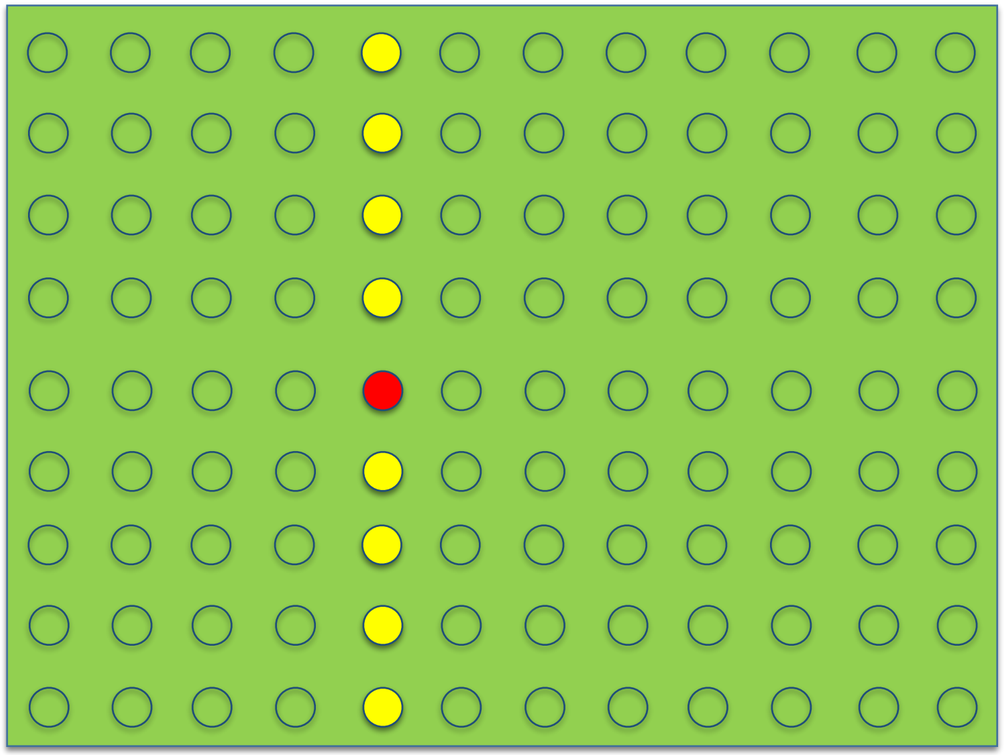
Optimization #4: High-order stencils



Only the grid nodes in one plane need to be shared to evaluate the stencil – store these in shared memory

Optimization #4: High-order stencils

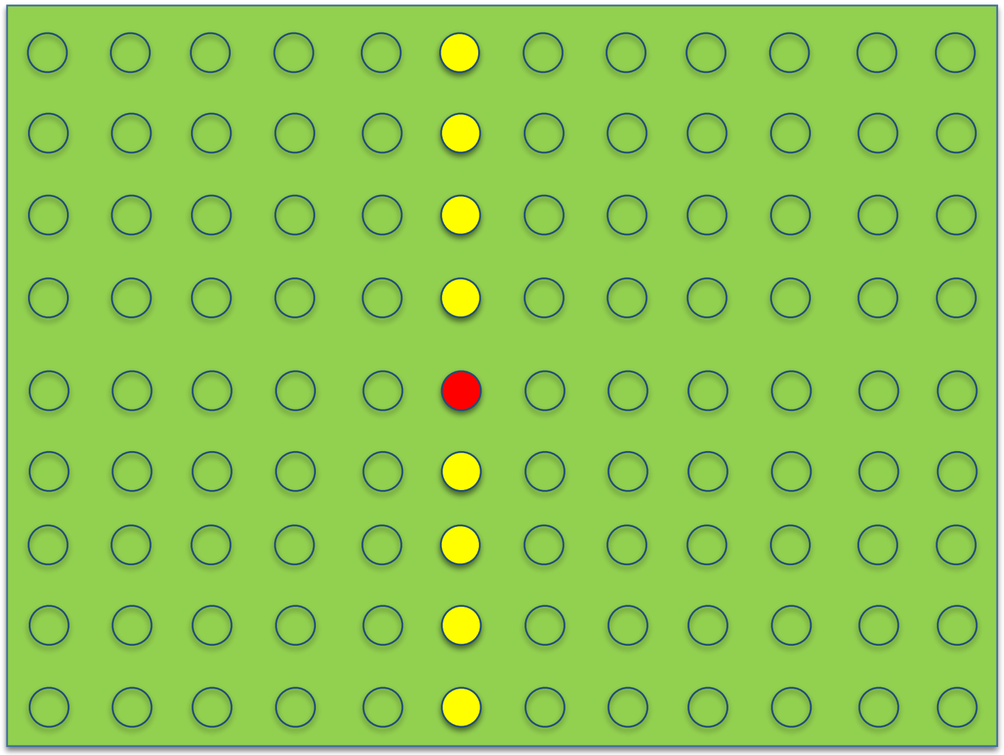
Registers



The other grid elements can be stored in registers for each thread

Optimization #4: High-order stencils

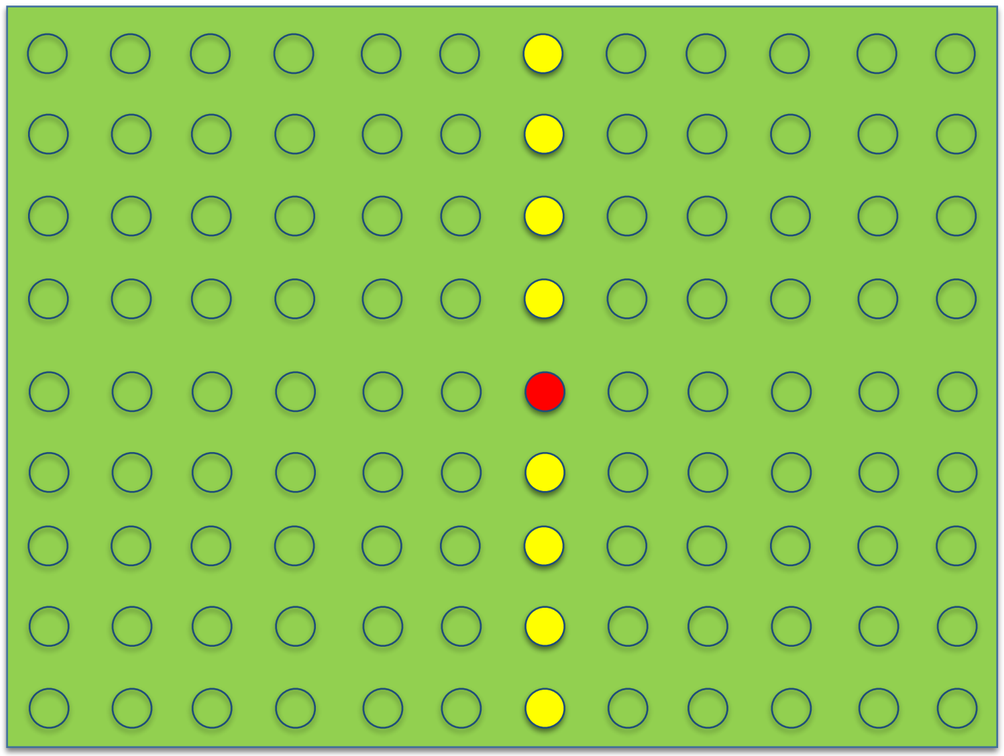
Registers



The other grid elements can be stored in registers for each thread

Optimization #4: High-order stencils

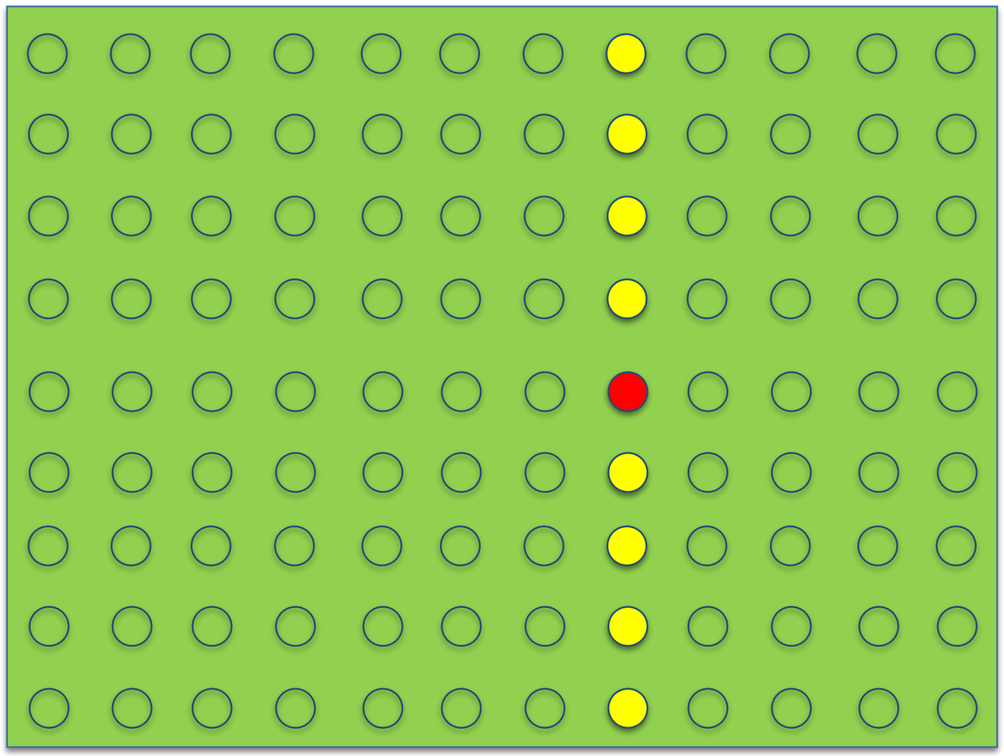
Registers



The other grid elements can be stored in registers for each thread

Optimization #4: High-order stencils

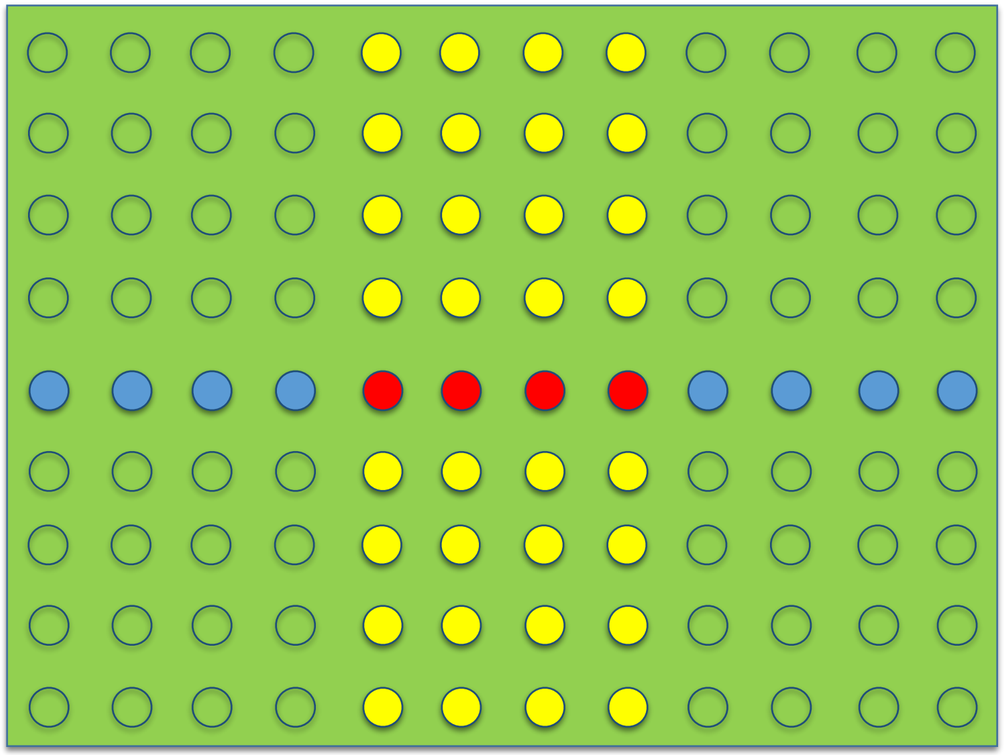
Registers



The other grid elements can be stored in registers for each thread

Optimization #4: High-order stencils

Shared Memory and
Registers



Final mix of shared memory and register storage

Application: Computational Fluid Dynamics (CFD)

- Uses numerical analysis and algorithms to solve and analyze fluid flow problems

- Solve system of partial differential equations (PDE) to determine flow

- Often use iterative solvers like Gauss-Seidel or Jacobi

https://en.wikipedia.org/wiki/Computational_fluid_dynamics

Application: Computational Fluid Dynamics (CFD)

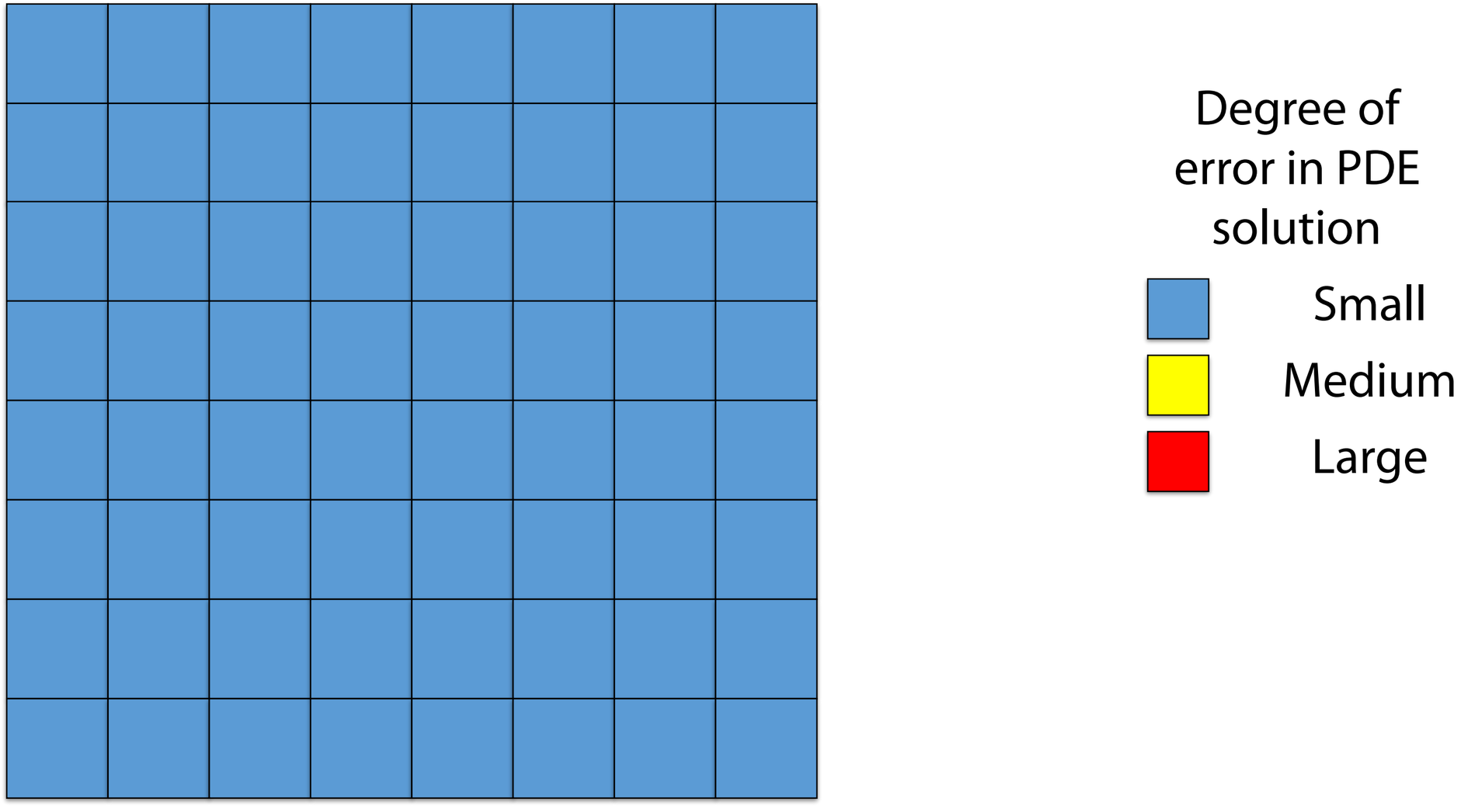
- Start with initial fluid state in each cell

- At each timestep, solve the PDEs
  - These are (basically) stencil operations

- Once PDEs converge, begin next timestep

https://en.wikipedia.org/wiki/Computational_fluid_dynamics

Application: Computational Fluid Dynamics (CFD)



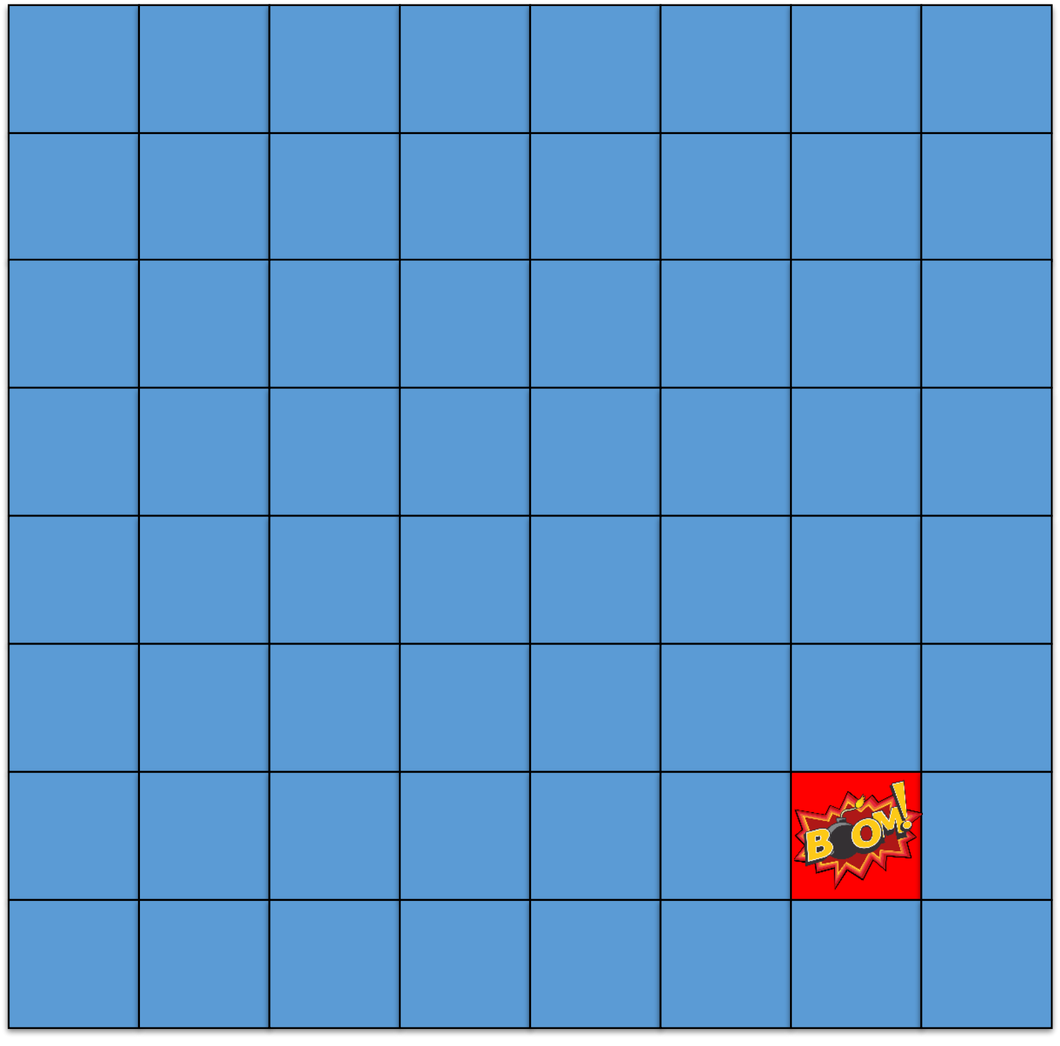
Degree of error in PDE solution

Small

Medium

Large

Application: Computational Fluid Dynamics (CFD)



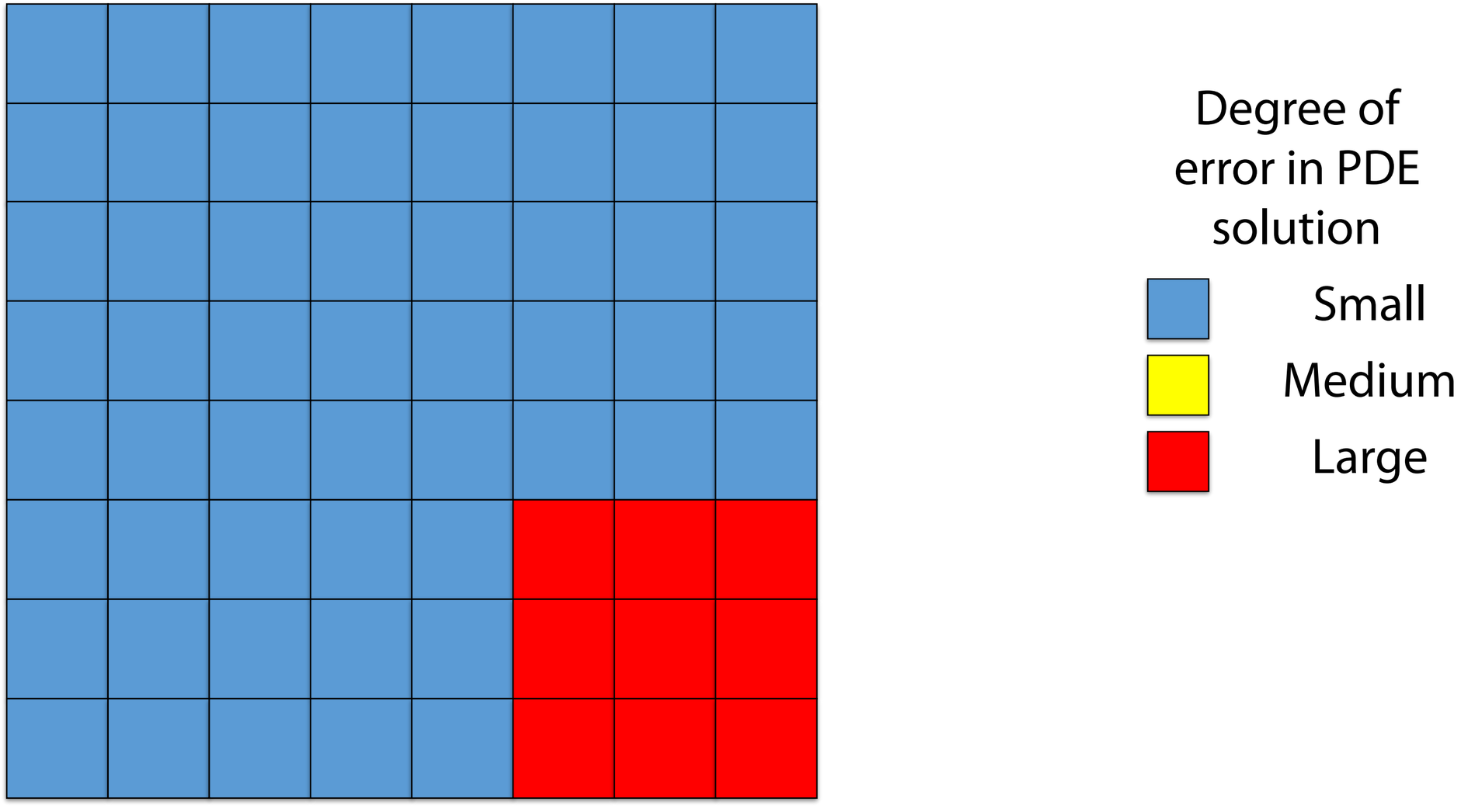
Degree of error in PDE solution

Small

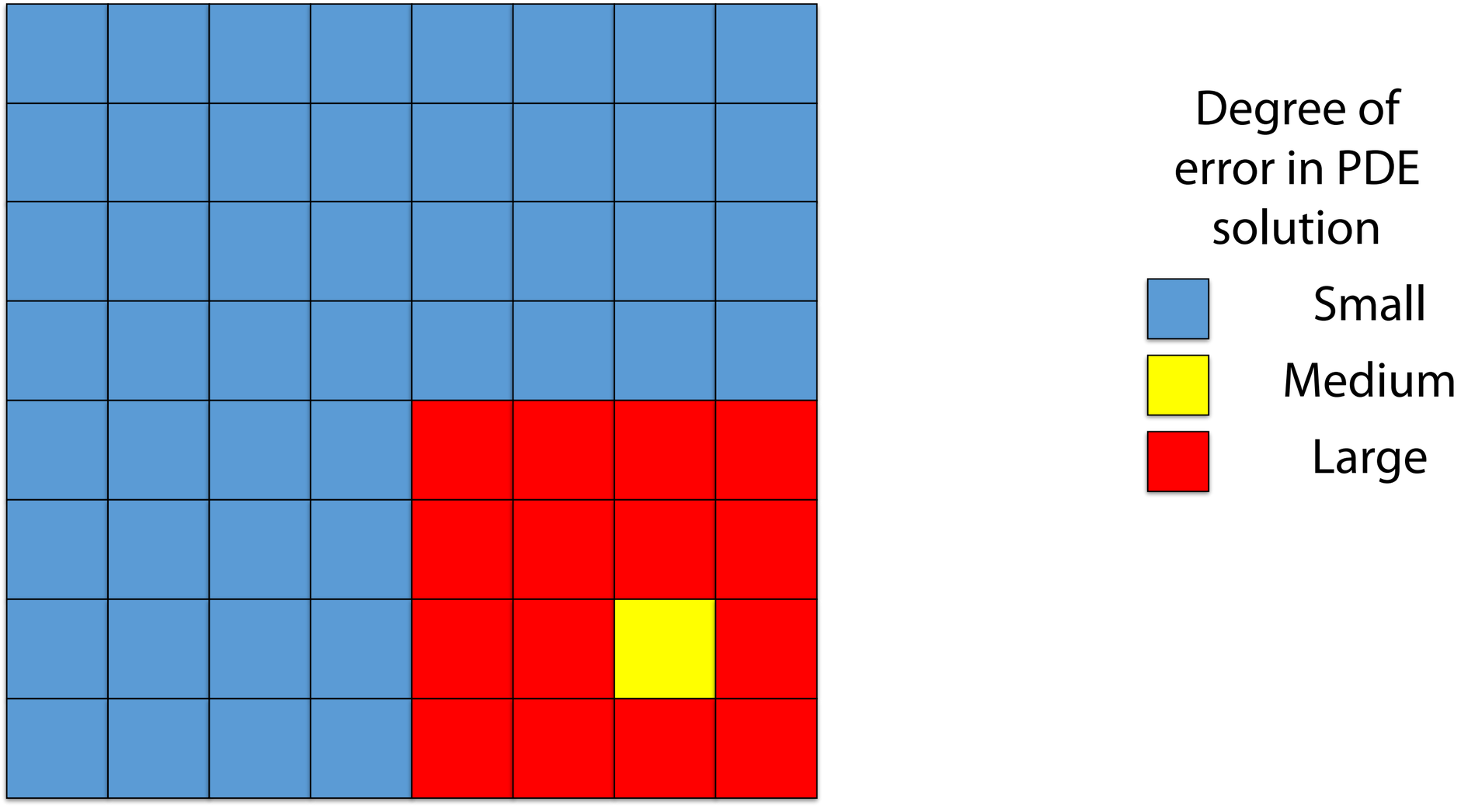
Medium

Large

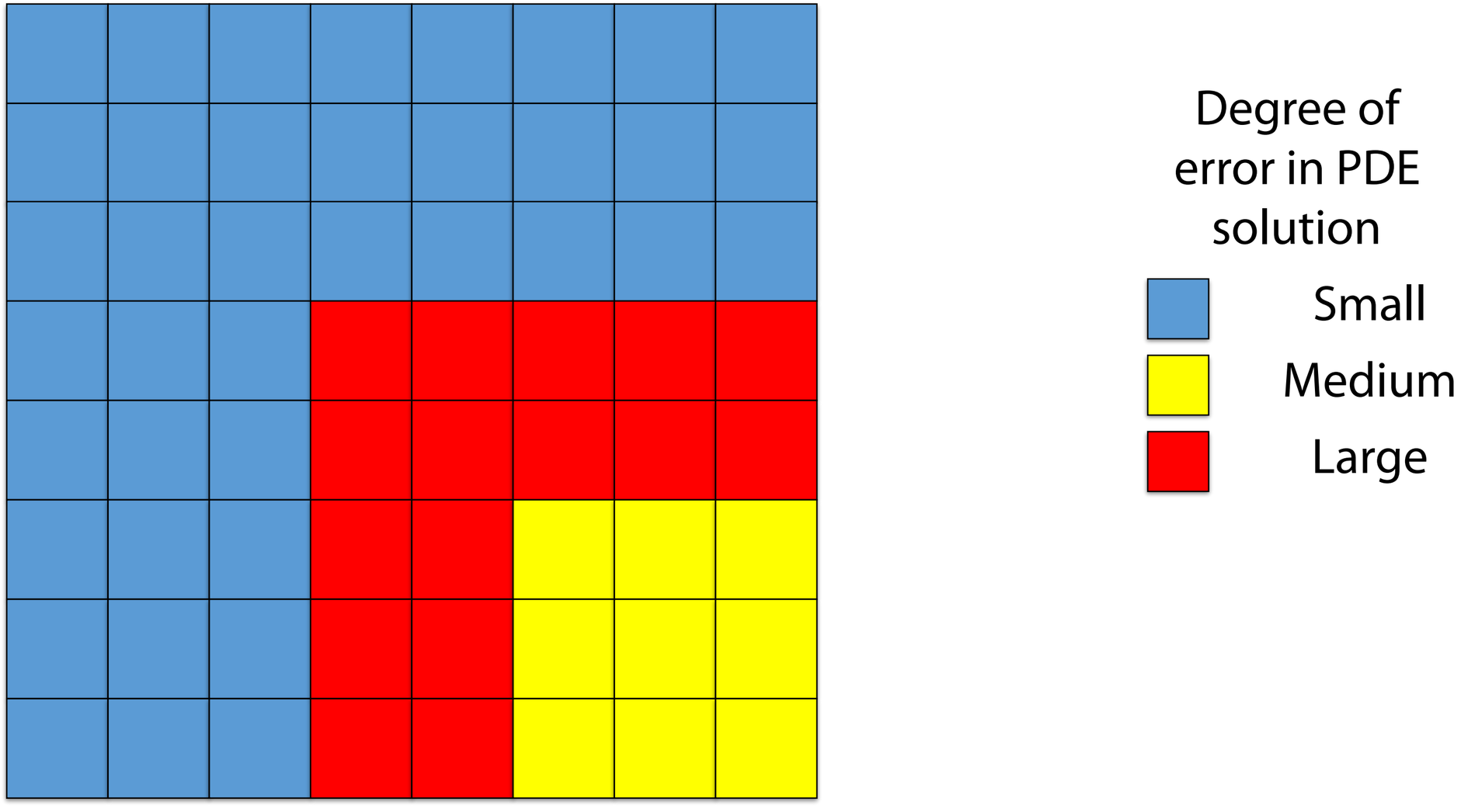
Application: Computational Fluid Dynamics (CFD)



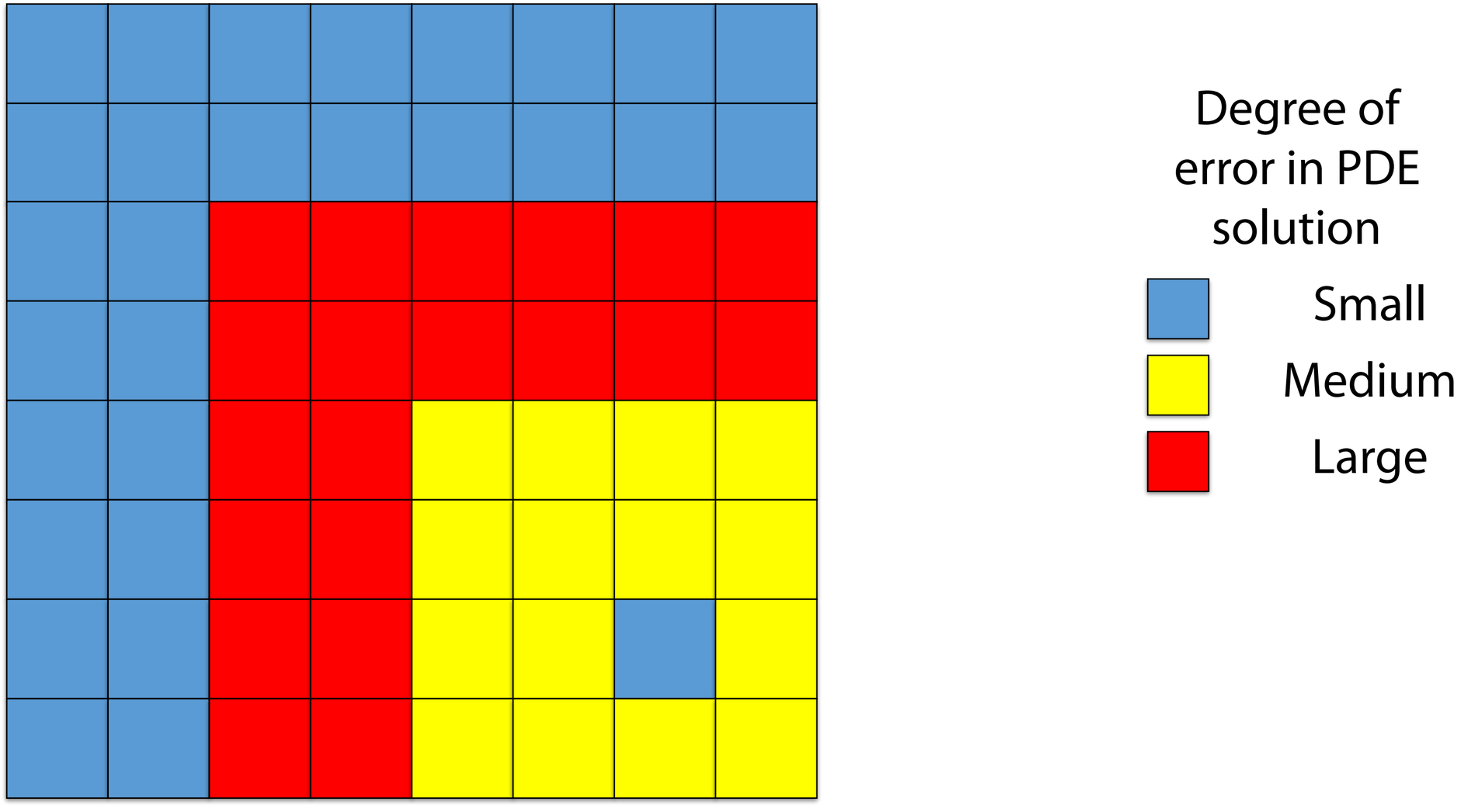
Application: Computational Fluid Dynamics (CFD)



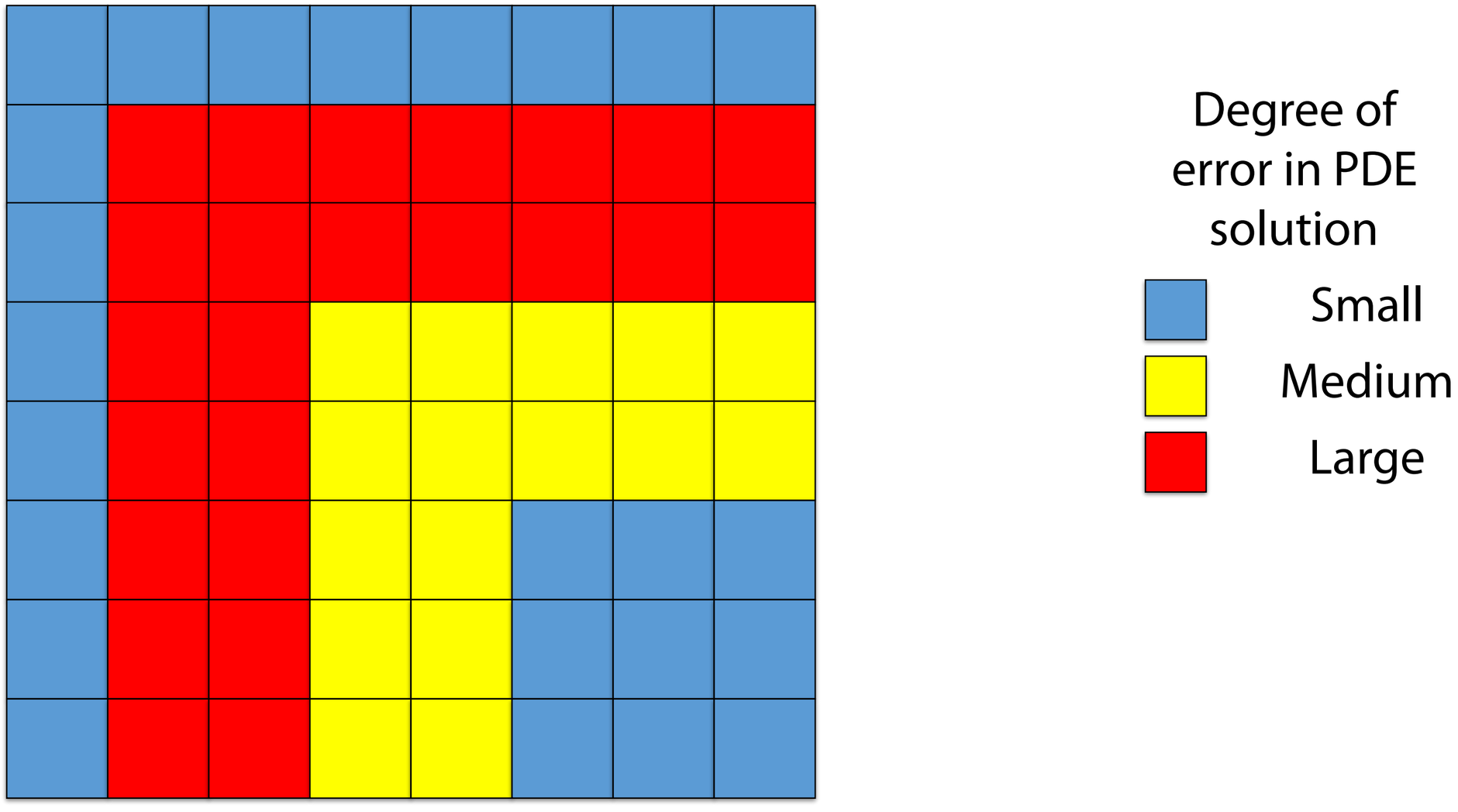
Application: Computational Fluid Dynamics (CFD)



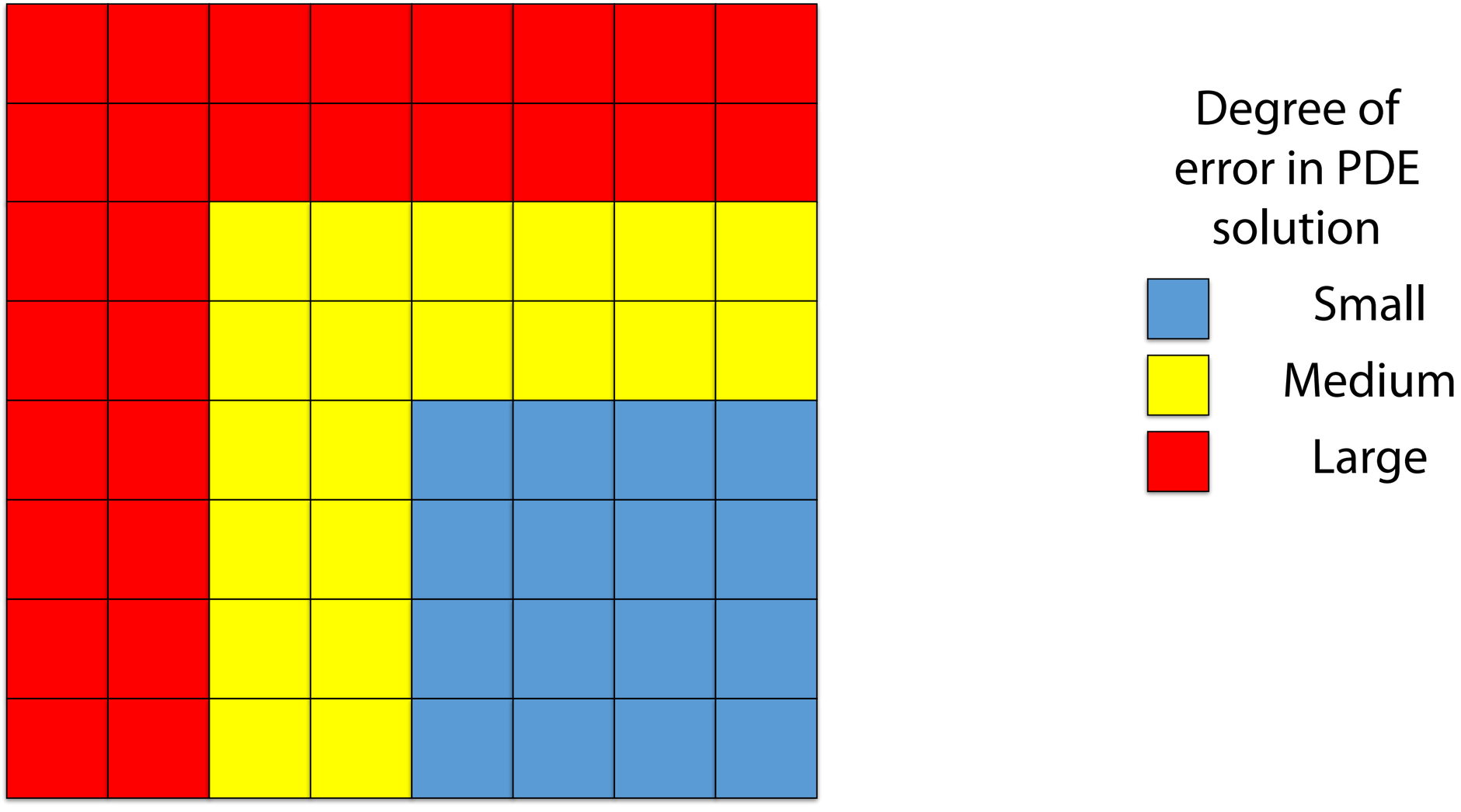
Application: Computational Fluid Dynamics (CFD)



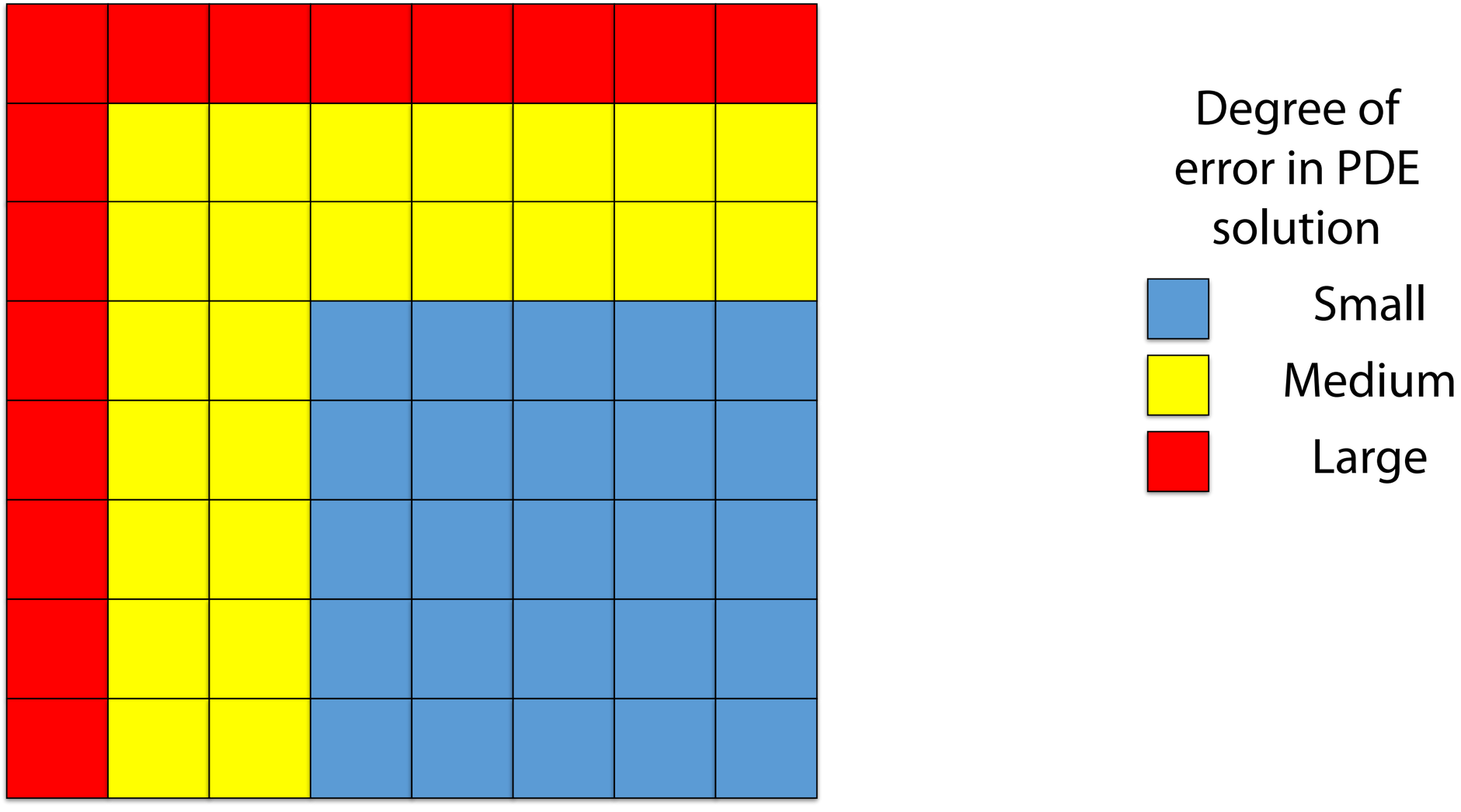
Application: Computational Fluid Dynamics (CFD)



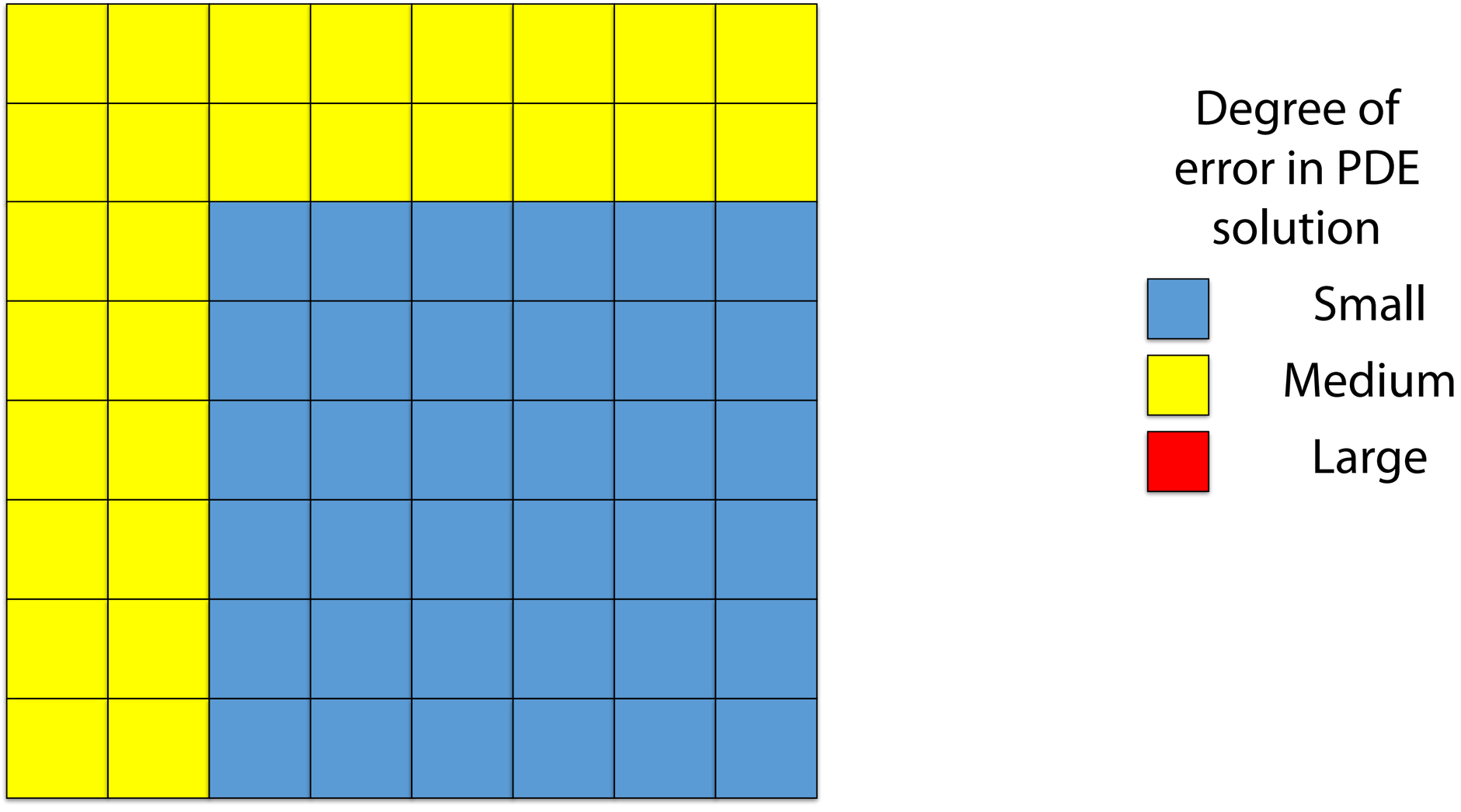
Application: Computational Fluid Dynamics (CFD)



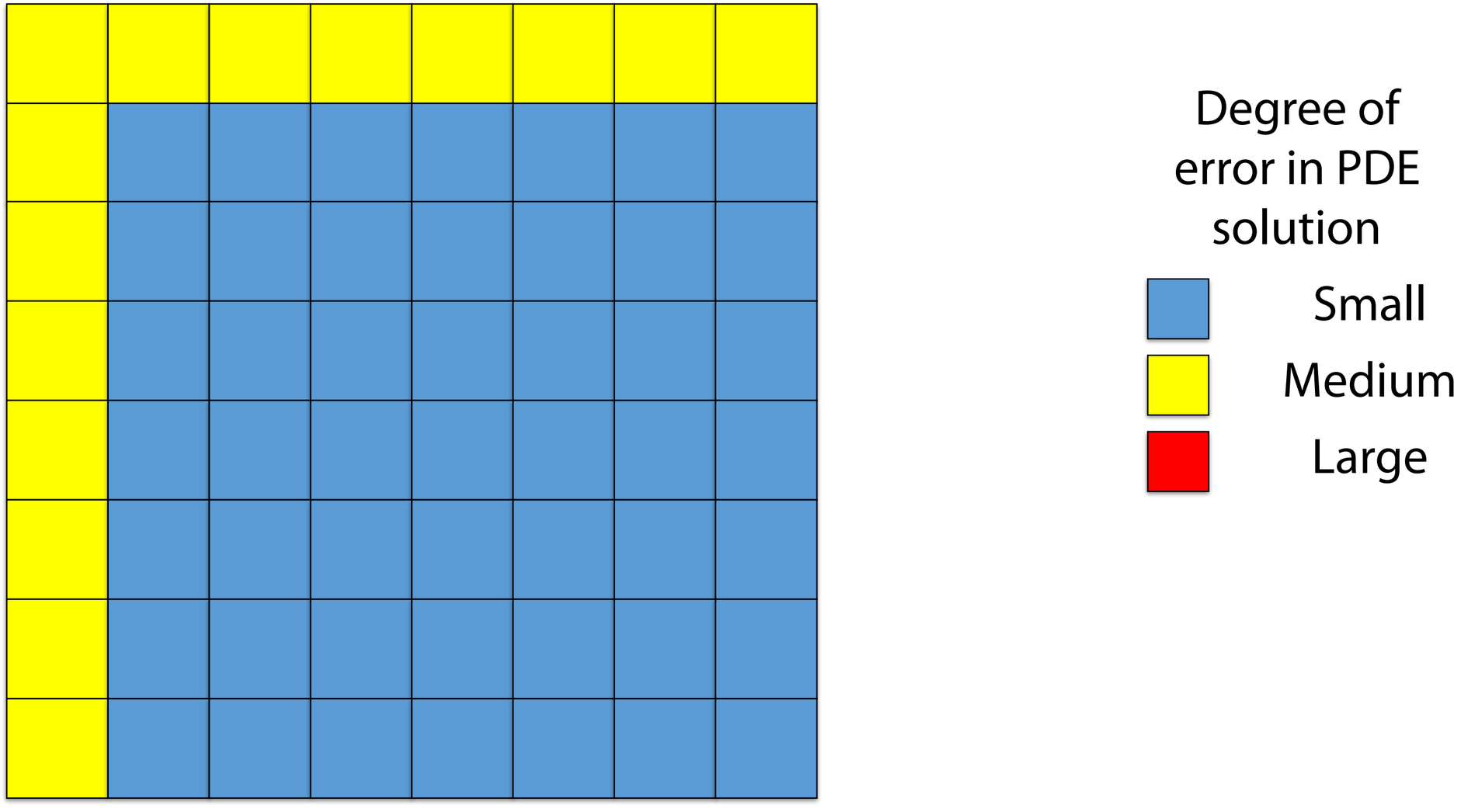
Application: Computational Fluid Dynamics (CFD)



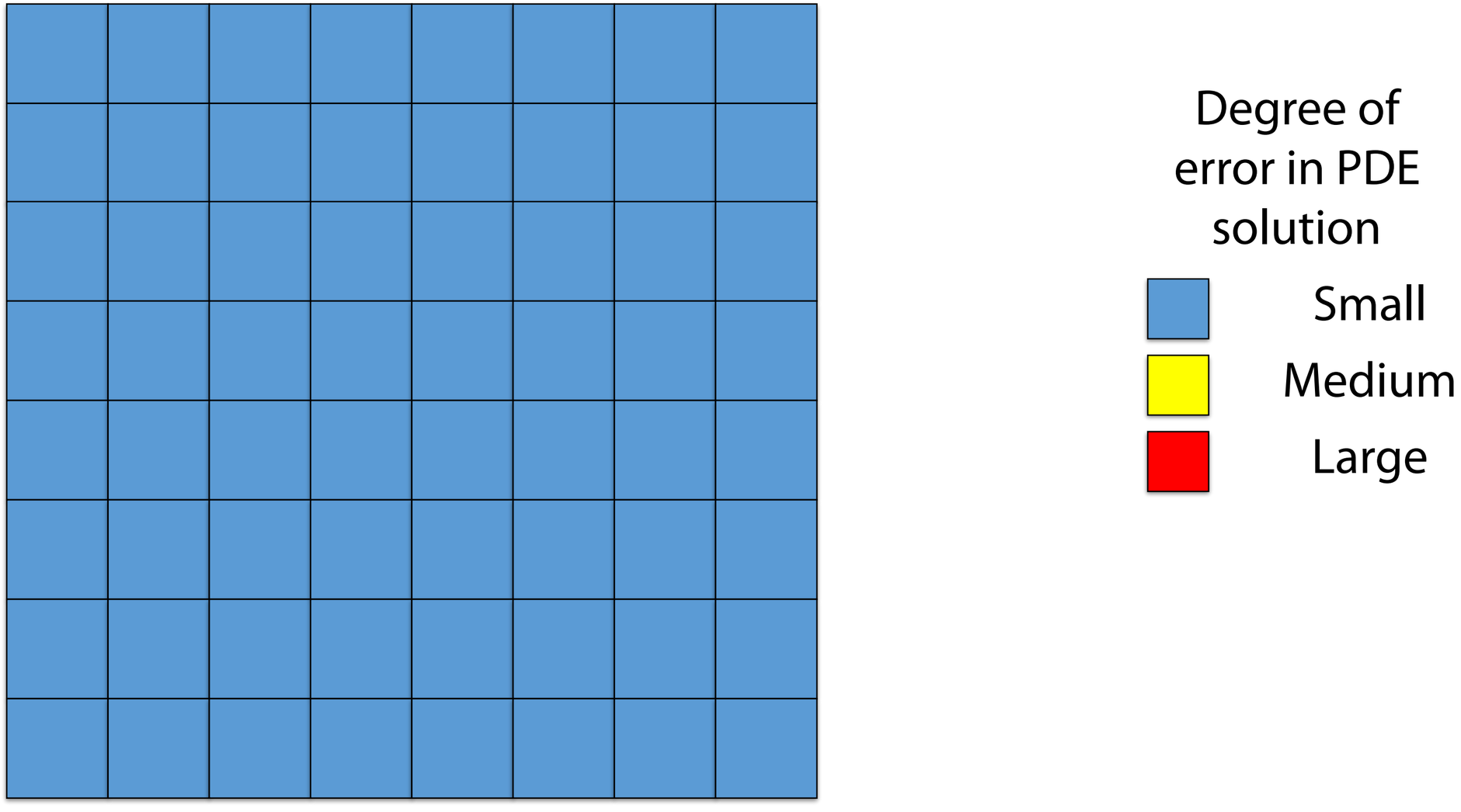
Application: Computational Fluid Dynamics (CFD)



Application: Computational Fluid Dynamics (CFD)



Application: Computational Fluid Dynamics (CFD)



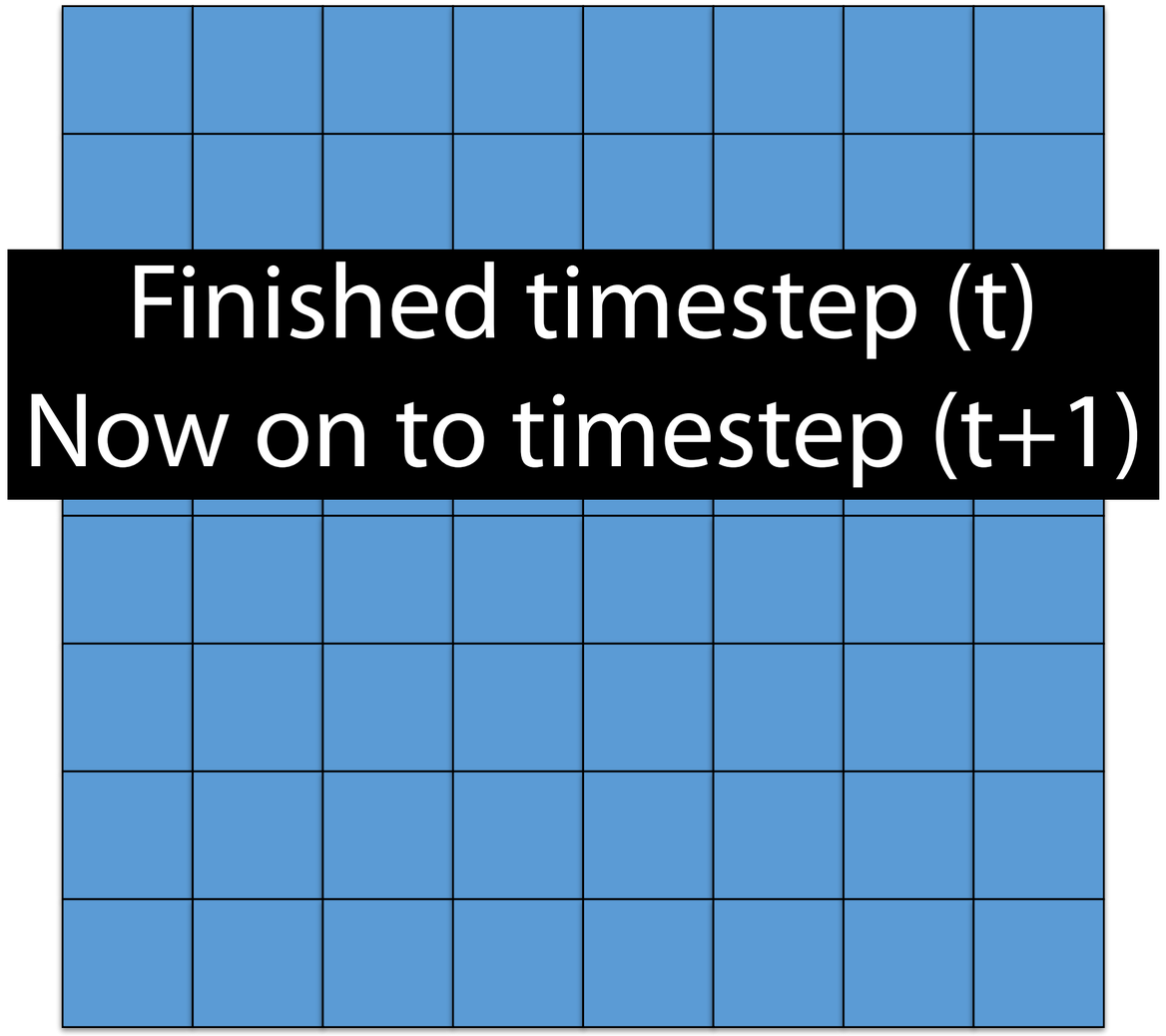
Degree of error in PDE solution

Small

Medium

Large

Application: Computational Fluid Dynamics (CFD)



Degree of error in PDE solution

Small

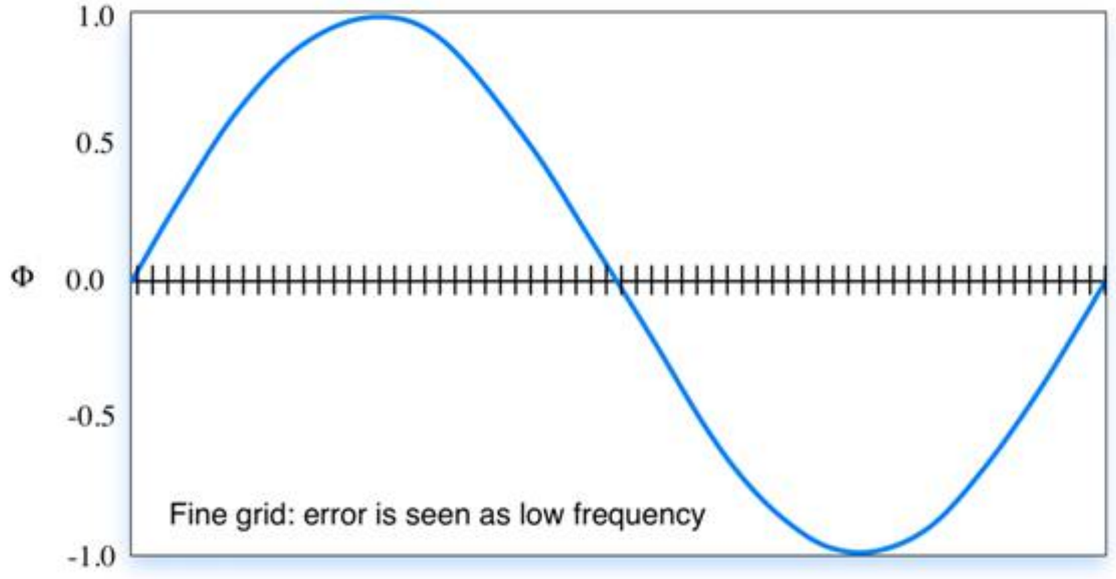
Medium

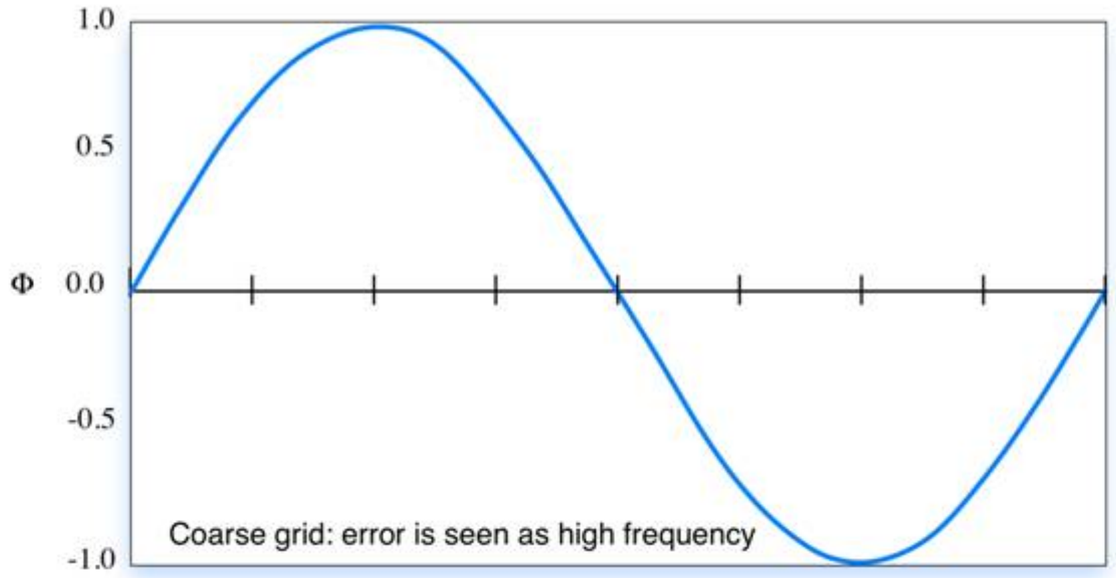
Large

Multigrid

- Iterative solvers can take a long time to converge
  - Information must propagate, one cell at a time, to the entire grid

- Coarser grids converge more quickly
  - Information travels to entire grid in fewer iterations





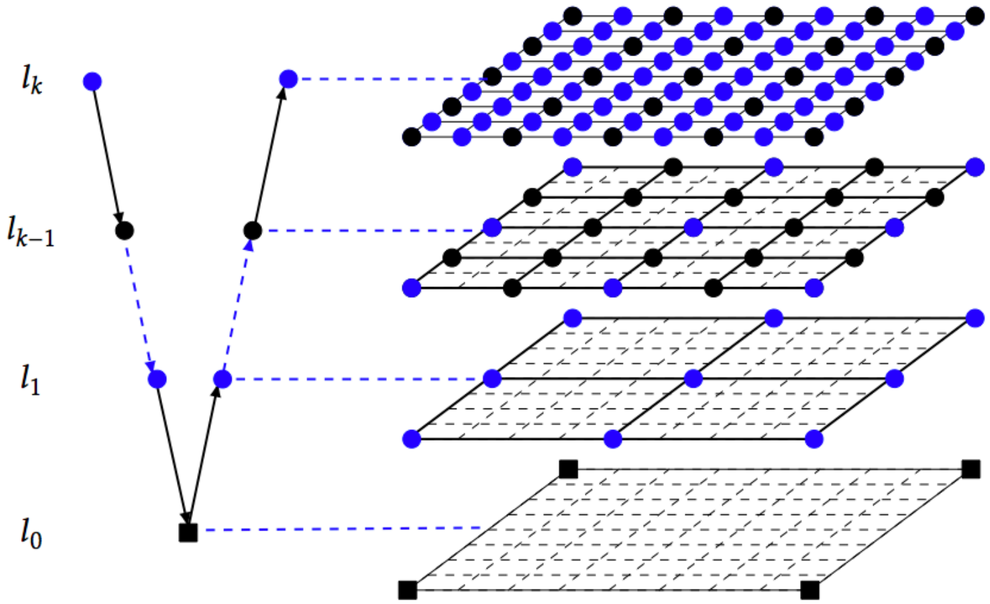
http://www.cfd-online.com/Wiki/Multigrid_methods

Three Main Steps of Multigrid

- **Smoothing – reduce high frequency errors using a few iterations of a solver (e.g., Gauss-Seidel)**

- **Restriction – downsample error to a coarser grid**

- **Interpolation (prolongation) – interpolate coarse grid error to finer grid**



http://web.utk.edu/~wfeng1/style/mggrid.png