POLITECNICO DI MILANO

Parallel Patterns 4th part

058165 – Parallel Computing

Fabrizio Ferrandi

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
fabrizio.ferrandi@polimi.it

References

- “Structured Parallel Programming: Patterns for Efficient Computation,” Michael McCool, Arch Robinson, James Reinders, 1st edition, Morgan Kaufmann, ISBN: 978-0-12-415993-8, 2012

Outline

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Stencil Pattern

- A stencil pattern is a map where each output depends on a "neighborhood" of inputs
- These inputs are a set of fixed offsets relative to the output position
- A stencil output is a function of a "neighborhood" of elements in an input collection
  - Applies the stencil to select the inputs
- Data access patterns of stencils are regular
  - Stencil is the "shape" of "neighborhood"
  - Stencil remains the same

Serial Stencil Example (part 1)

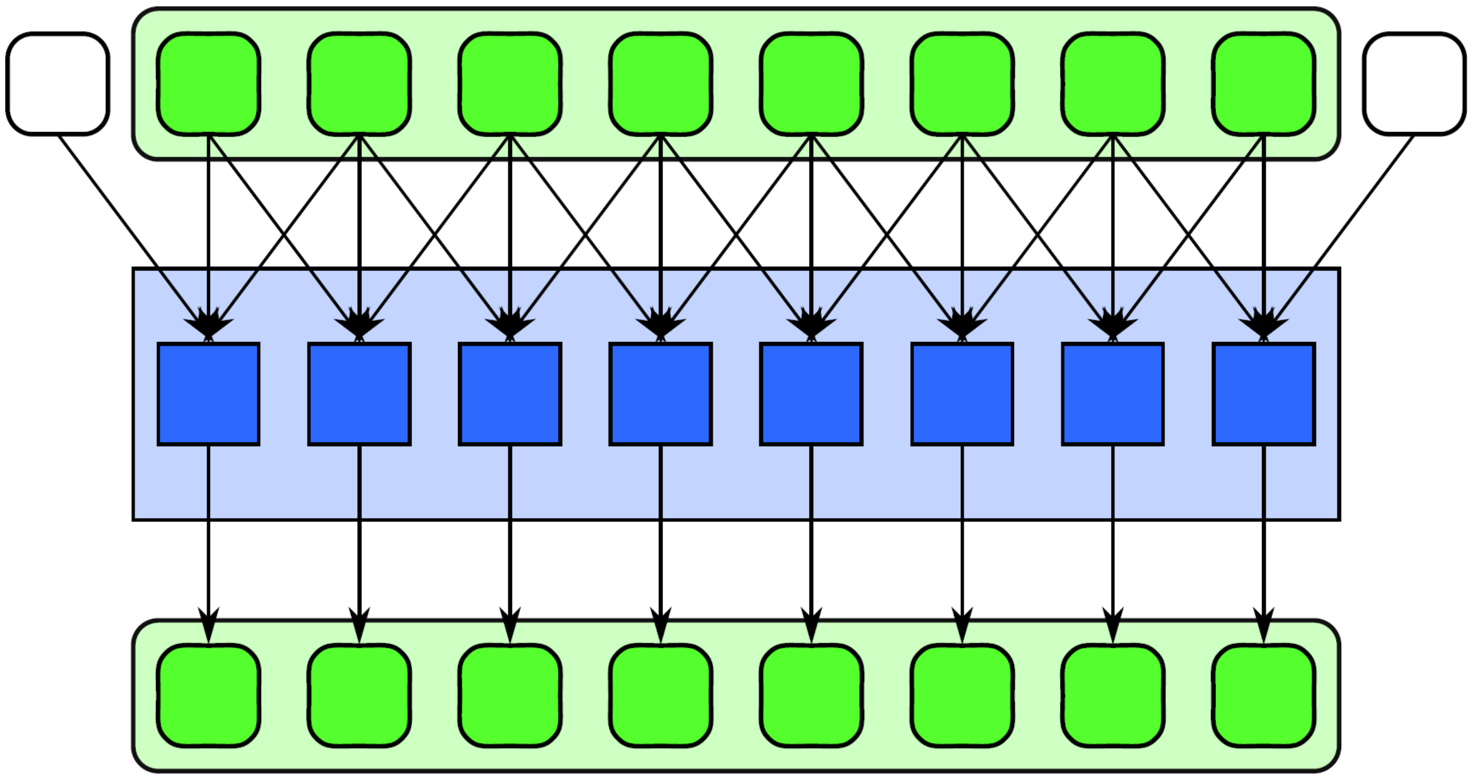
```
template<
    int NumOff,      // number of offsets
    typename In,     // type of input locations
    typename Out,    // type of output locations
    typename F       // type of function/functor
>
void stencil(
    int n,        // number of elements in data collection
    const In a[], // input data collection (n elements)
    Out r[],      // output data collection (n elements)
    In b,         // boundary value
    F func,       // function/functor from neighborhood inputs to output
    const int offsets[] // offsets (NumOffsets elements)
) {
```

Serial Stencil Example (part 2)


```
    // array to hold neighbors
    In neighborhood[NumOff];
    // loop over all output locations
    for (int i = 0; i < n; ++i) {
        // loop over all offsets and gather neighborhood
        for (int j = 0; j < NumOff; ++j) {
            // get index of jth input location
            int k = i+offsets[j];
            if (0 <= k && k < n) {
                // read input location
                neighborhood[j] = a[k];
            } else {
                // handle boundary case
                neighborhood[j] = b;
            }
        }
        // compute output value from input neighborhood
        r[i] = func(neighborhood);
    }
}
```

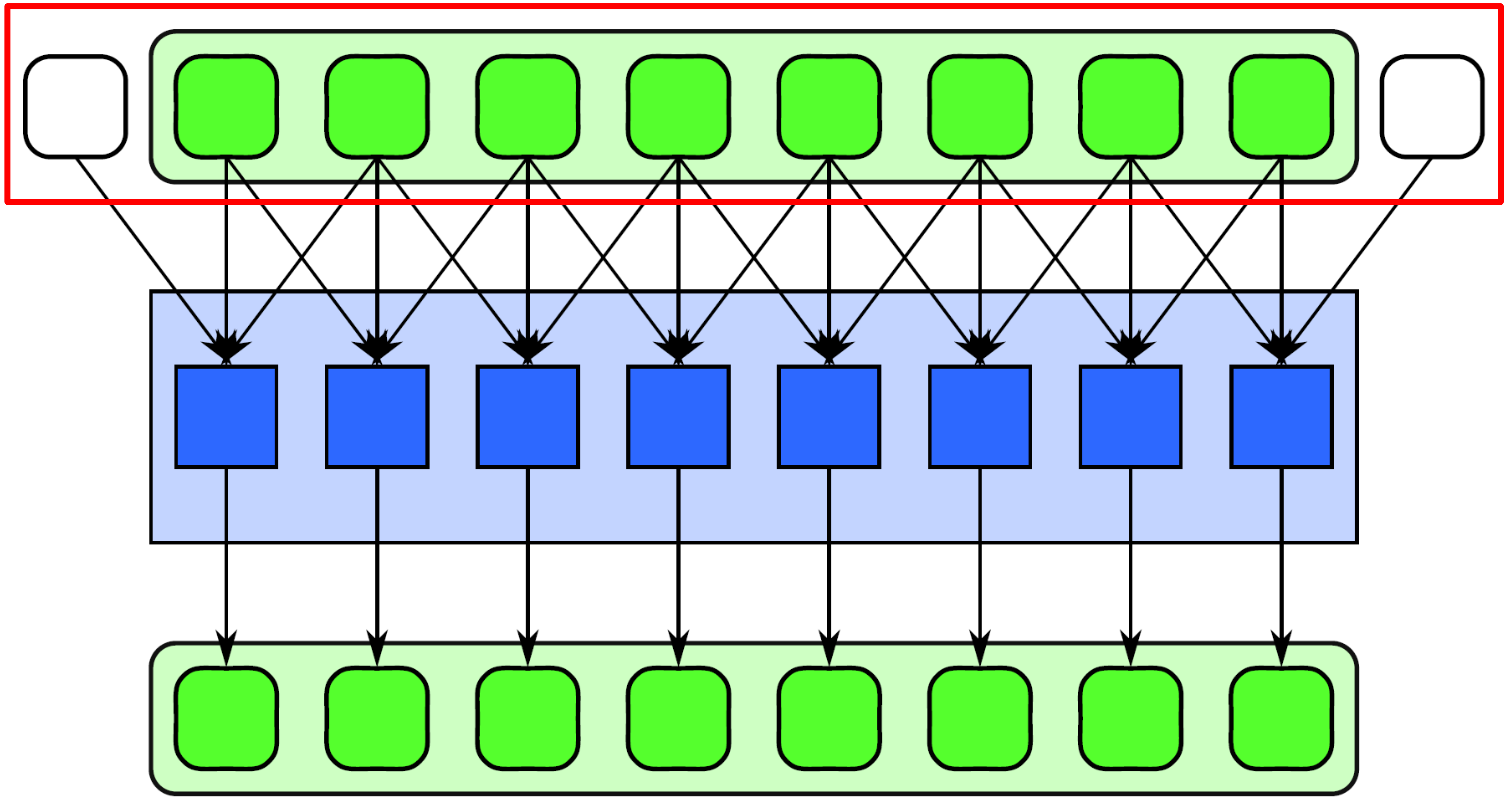
How would we parallelize this?

What is the stencil pattern?



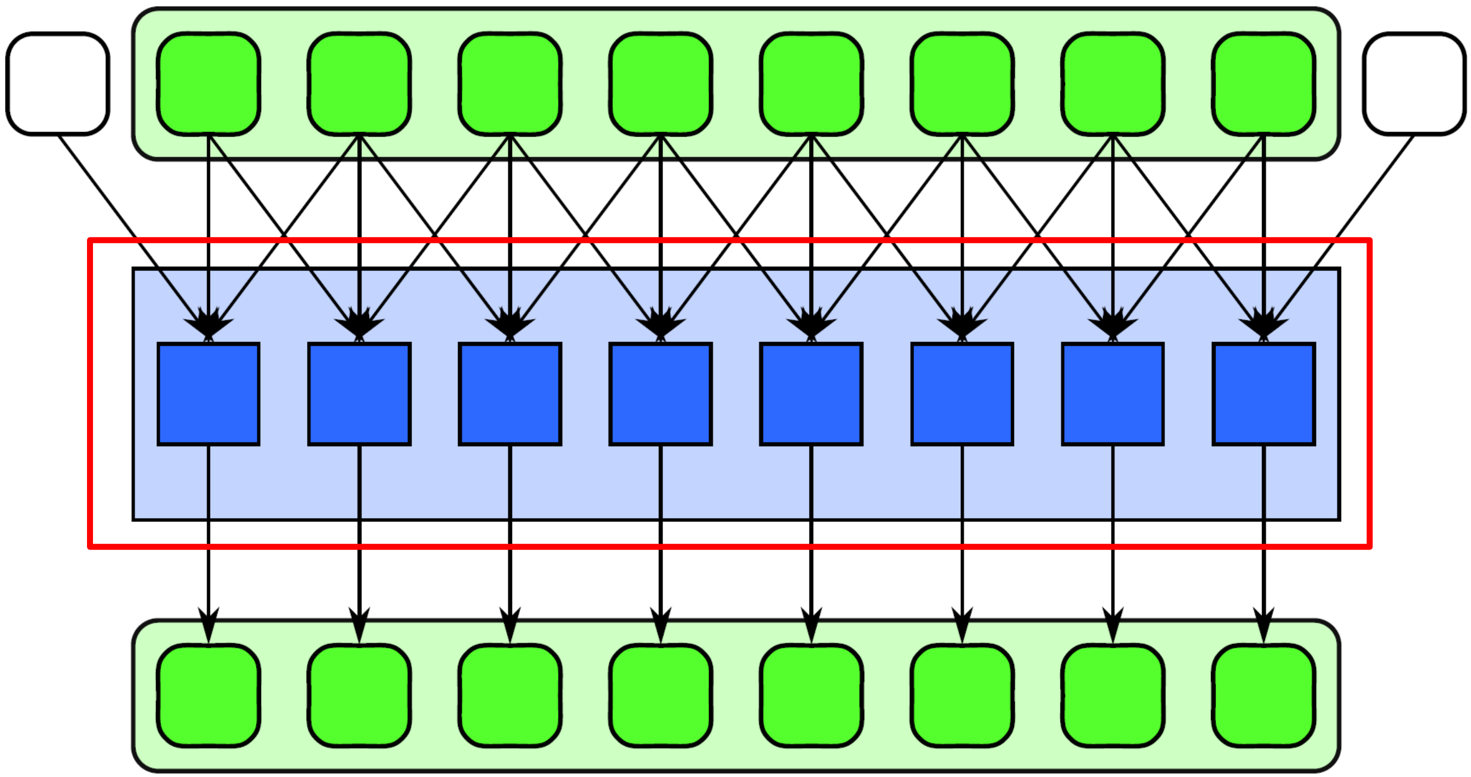
What is the stencil pattern?

Input array

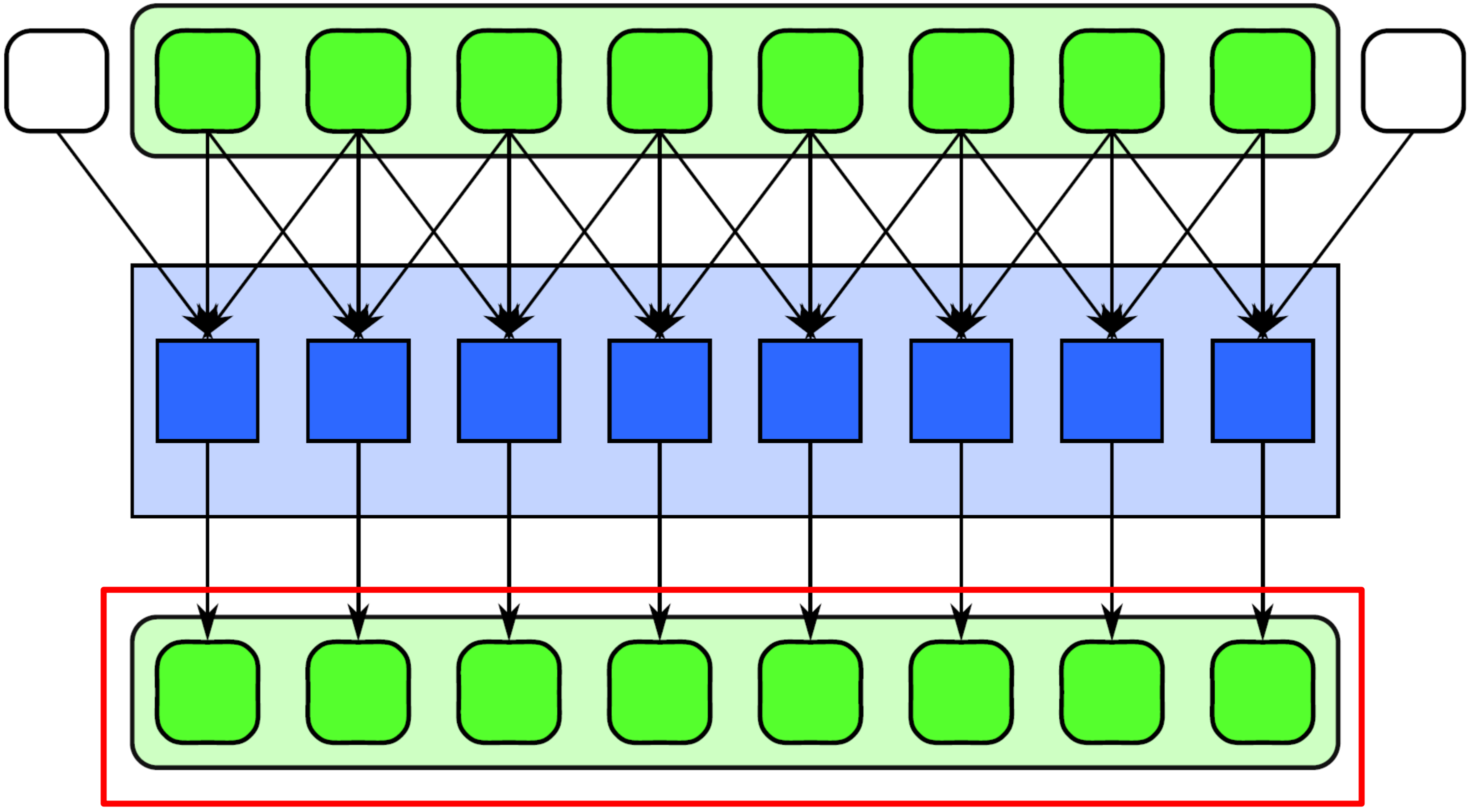


What is the stencil pattern?

Function

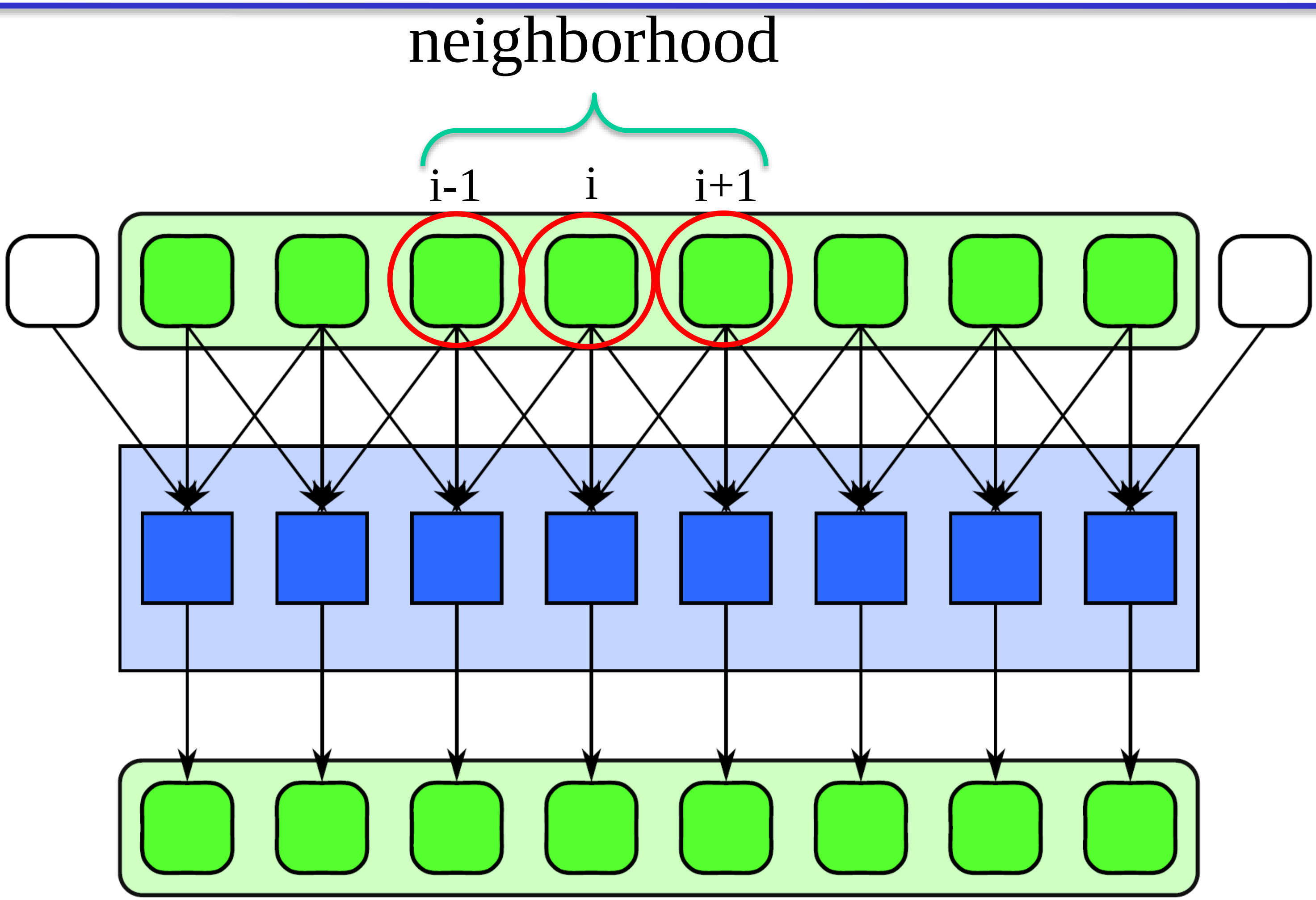


What is the stencil pattern?



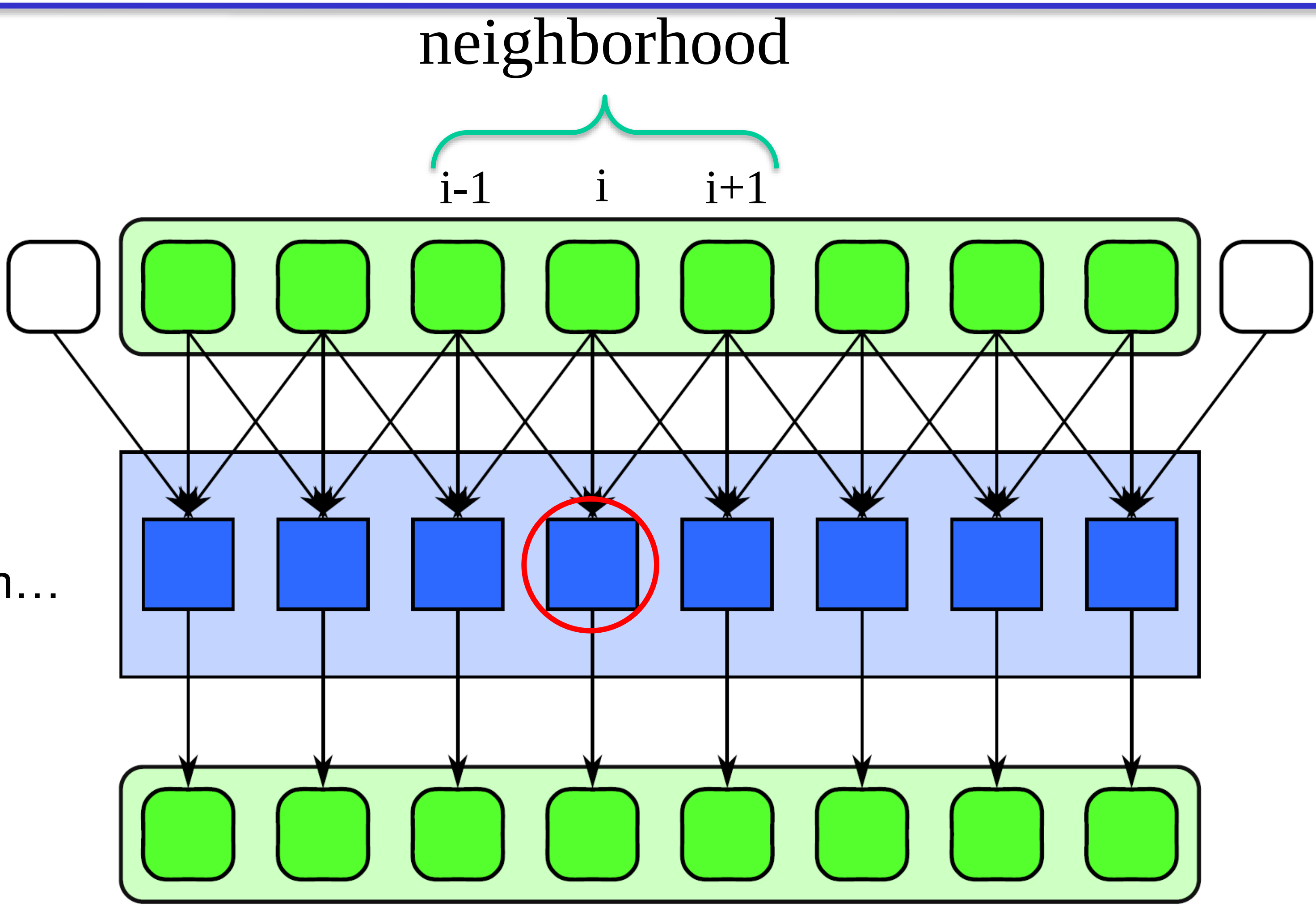
Output Array

What is the stencil pattern?



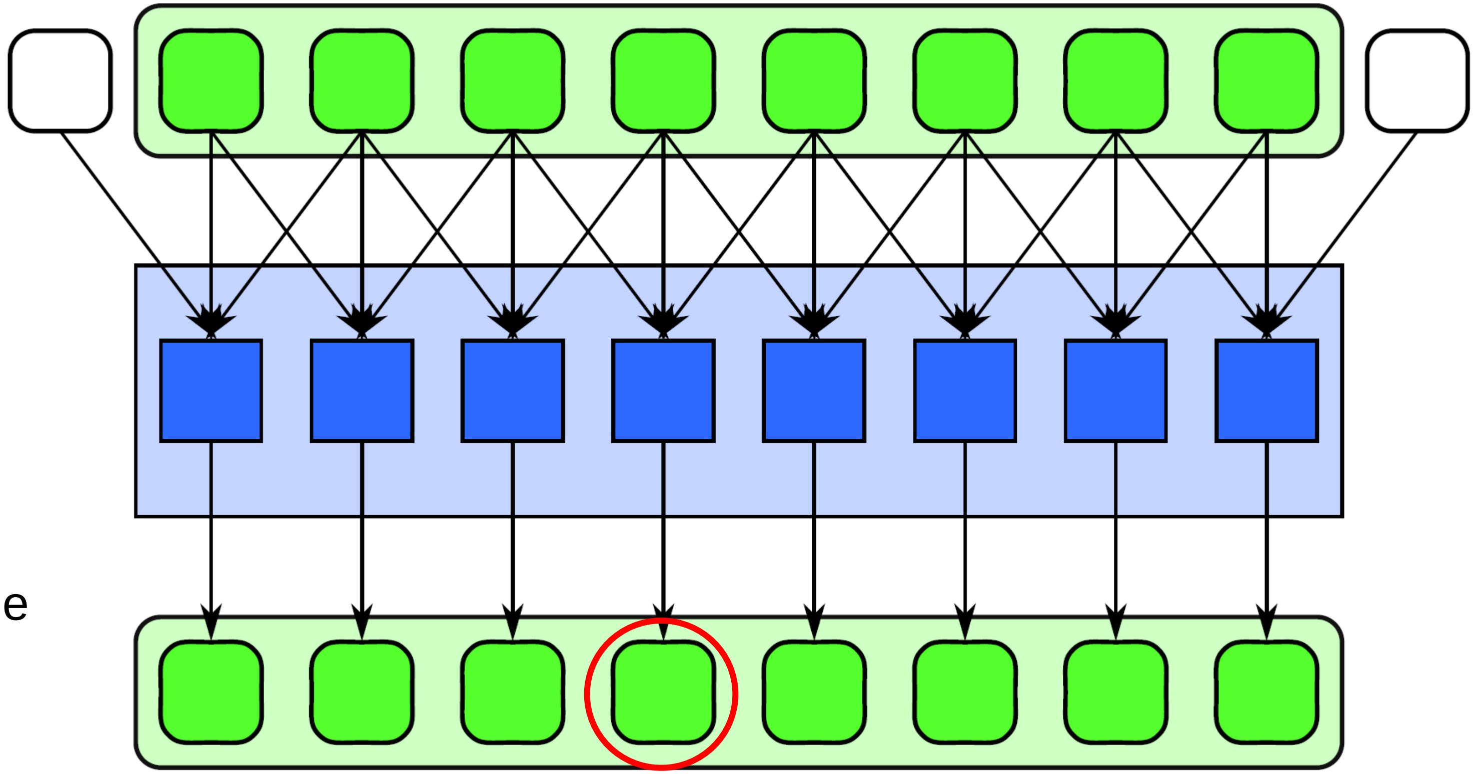
This stencil has 3 elements in the neighborhood: i-1, i, i+1

What is the stencil pattern?



Applies some function to them…

What is the stencil pattern?

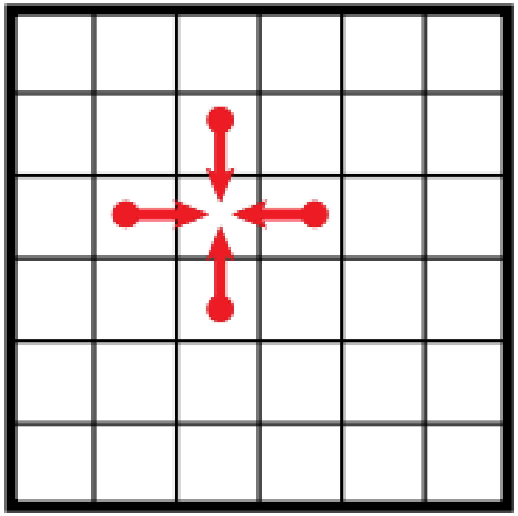


And outputs to the i^{th} position of the output array

Stencil Patterns

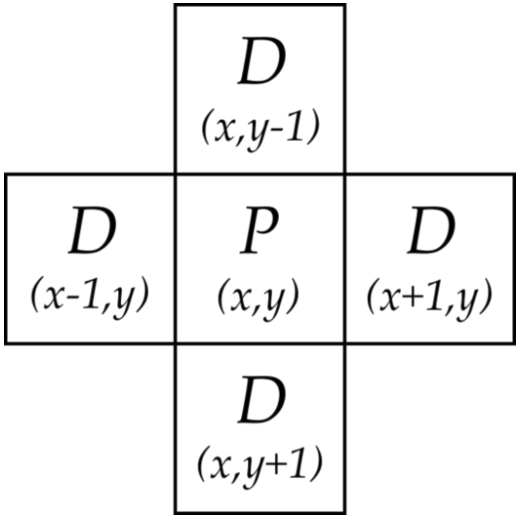
- Stencils can operate on one dimensional and multidimensional data
- Stencil neighborhoods can range from compact to sparse, square to cube, and anything else!
- It is the pattern of the stencil that determines how the stencil operates in an application

2-Dimensional Stencils



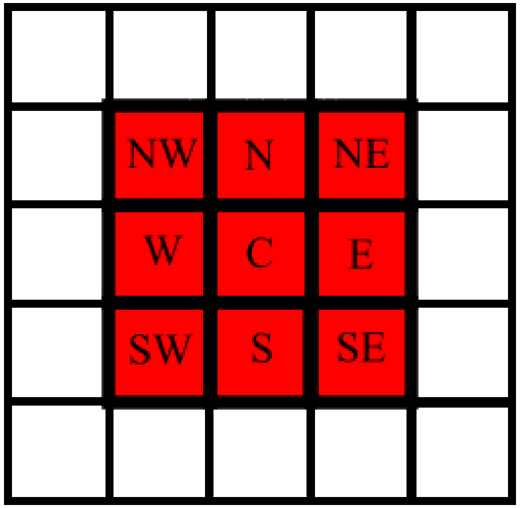
4-point stencil

Center cell (P) is not used



5-point stencil

Center cell (P) is used as well

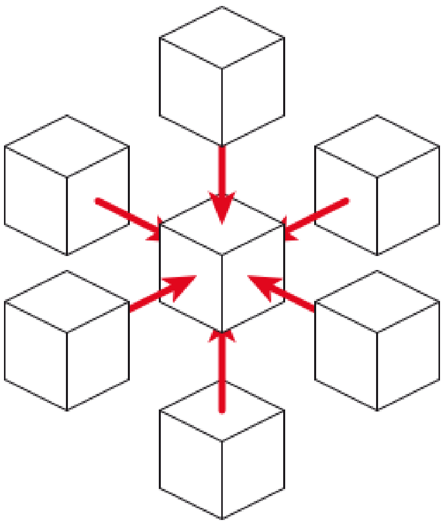


9-point stencil

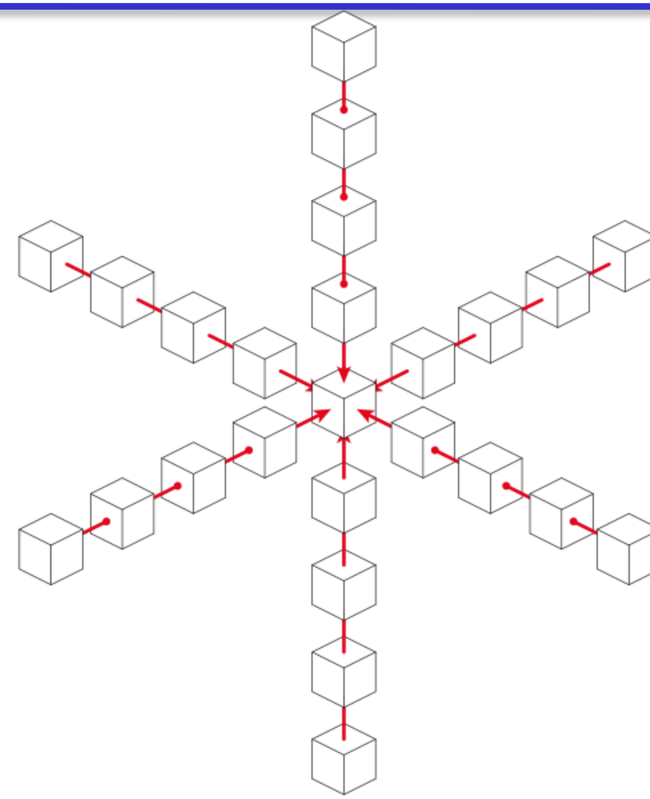
Center cell (C) is used as well

Source: http://en.wikipedia.org/wiki/Stencil_code

3-Dimensional Stencils



6-point stencil
(7-point stencil)



24-point stencil
(25-point stencil)

Source: http://en.wikipedia.org/wiki/Stencil_code

Stencil Example

- Here is our array, A

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

Stencil Example

- Here is our array **A**
- **B** is the output array
  - Initialize to all 0
- Apply a stencil operation to the inner square of the form:

B(i,j) = avg(A(i,j),
A(i-1,j), A(i+1,j),
A(i,j-1), A(i,j+1)
)

What is the stencil?

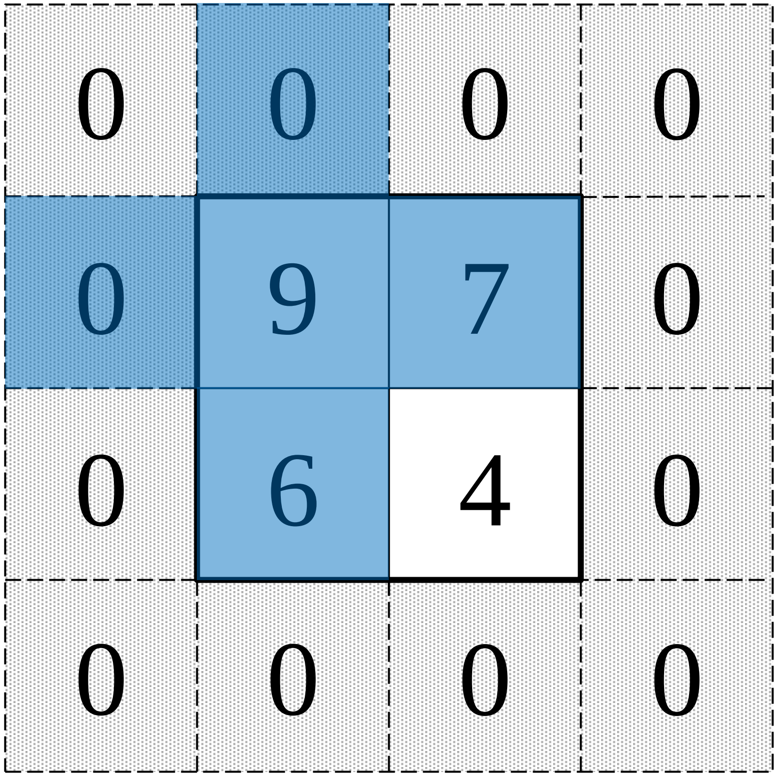
A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

Stencil Pattern Procedure

1) Average all blue squares

A



Stencil Pattern Procedure

1) Average all blue squares
2) Store result in B

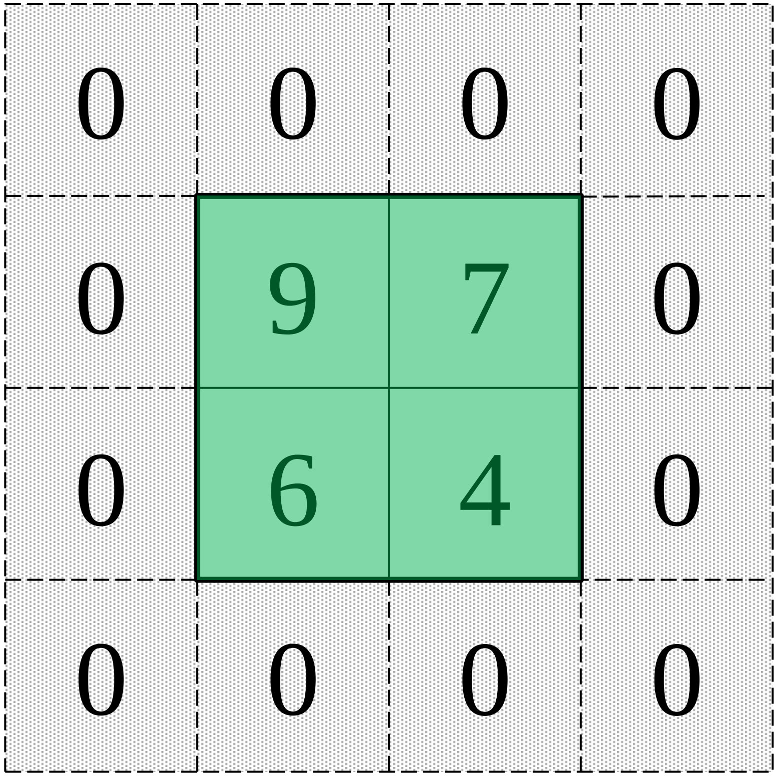
B

0	0	0	0
0	4.4	0	0
0	0	0	0
0	0	0	0

Stencil Pattern Procedure

1) Average all blue squares
2) Store result in B
3) Repeat 1 and 2 for all green squares

A



Practice!

Stencil Pattern Practice

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	0	0
0	0	0	0
0	0	0	0

Stencil Pattern Practice

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	4.0	0
0	0	0	0
0	0	0	0

Stencil Pattern Practice

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	4.0	0
0	3.8	0	0
0	0	0	0

Stencil Pattern Practice

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	4.0	0
0	3.8	3.4	0
0	0	0	0

Outline

- Partitioning
- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Serial Stencil Example (part 1)


```
template<
    int NumOff,      // number of  offsets
    typename In,     // type  of  input  locations
    typename Out,    // type  of  output  locations
    typename F       // type  of  function / functor
>
void stencil(
    int n,         // number of elements  in  data  collection
    const In a[], // input data collection  (n  elements)
    Out r[],       // output data  collection  (n elements)
    In b,          // boundary value
    F func,        // function / functor  from neighborhood inputs  to  output
    const int offsets[] // offsets (NumOffsets elements)
) {
```

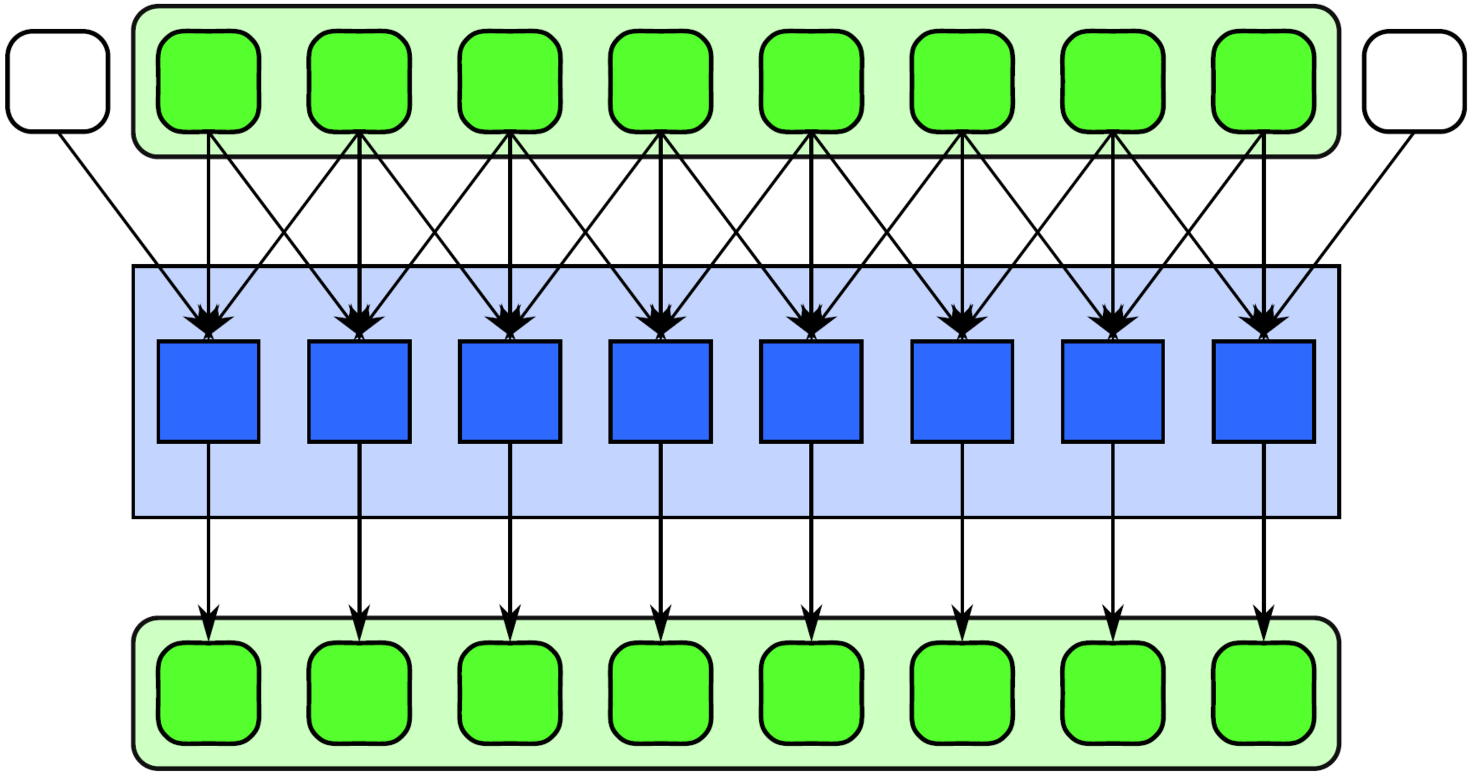
Serial Stencil Example (part 2)

```
    // array to hold neighbors
    In neighborhood[NumOff];
    // loop over all output locations
    for (int i = 0; i < n; ++i) {
        // loop over all offsets and gather neighborhood
        for (int j = 0; j < NumOff; ++j) {
            // get index of jth input location
            int k = i+offsets[j];
            if (0 <= k && k < n) {
                // read input location
                neighborhood[j] = a[k];
            } else {
                // handle boundary case
                neighborhood[j] = b;
            }
        }
        // compute output value from input neighborhood
a[i] r[i] = func(neighborhood);
    }
}
```

How would we parallelize this?

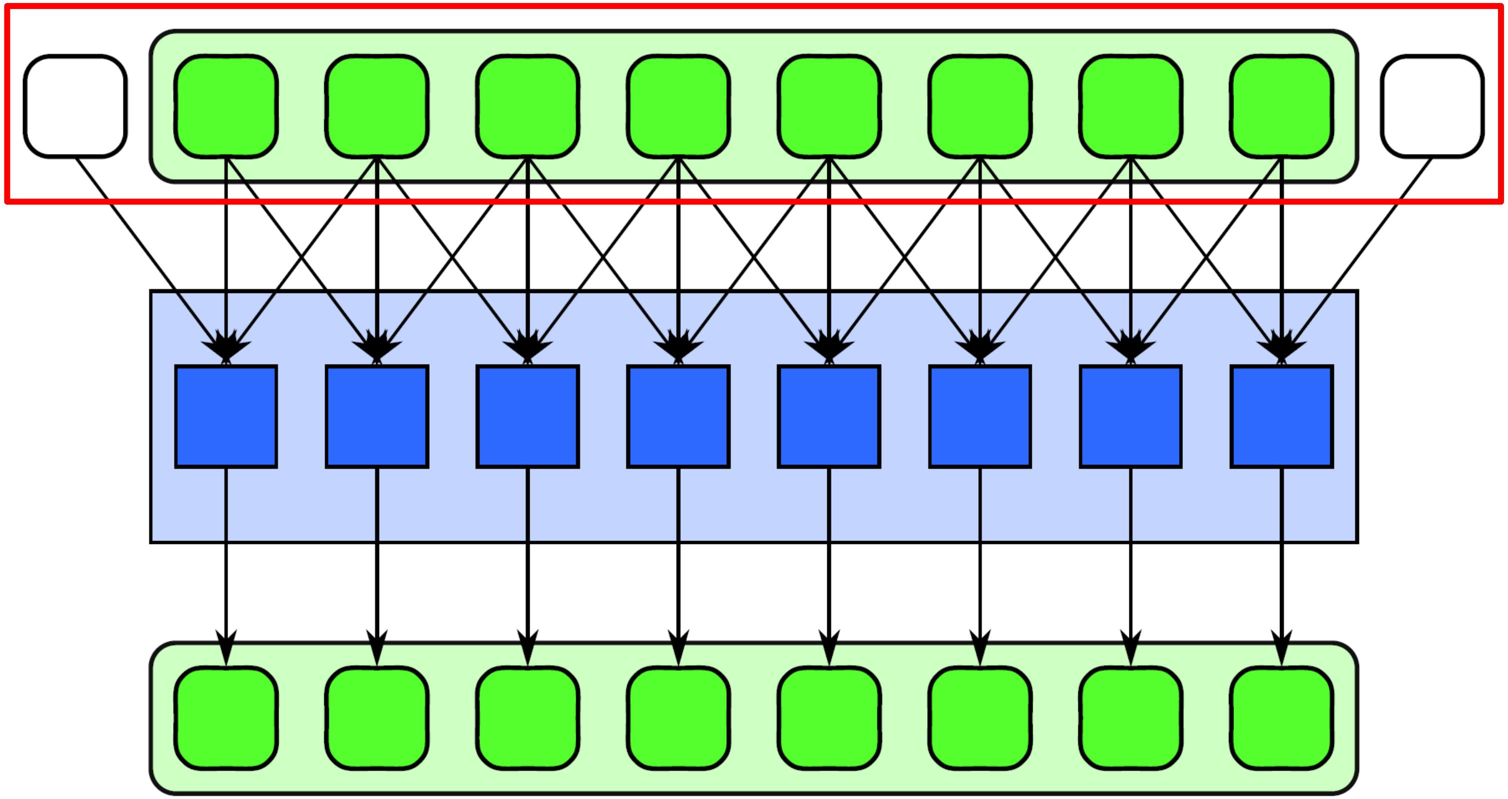
Updates occur in place!!!

Stencil Pattern with In Place Update



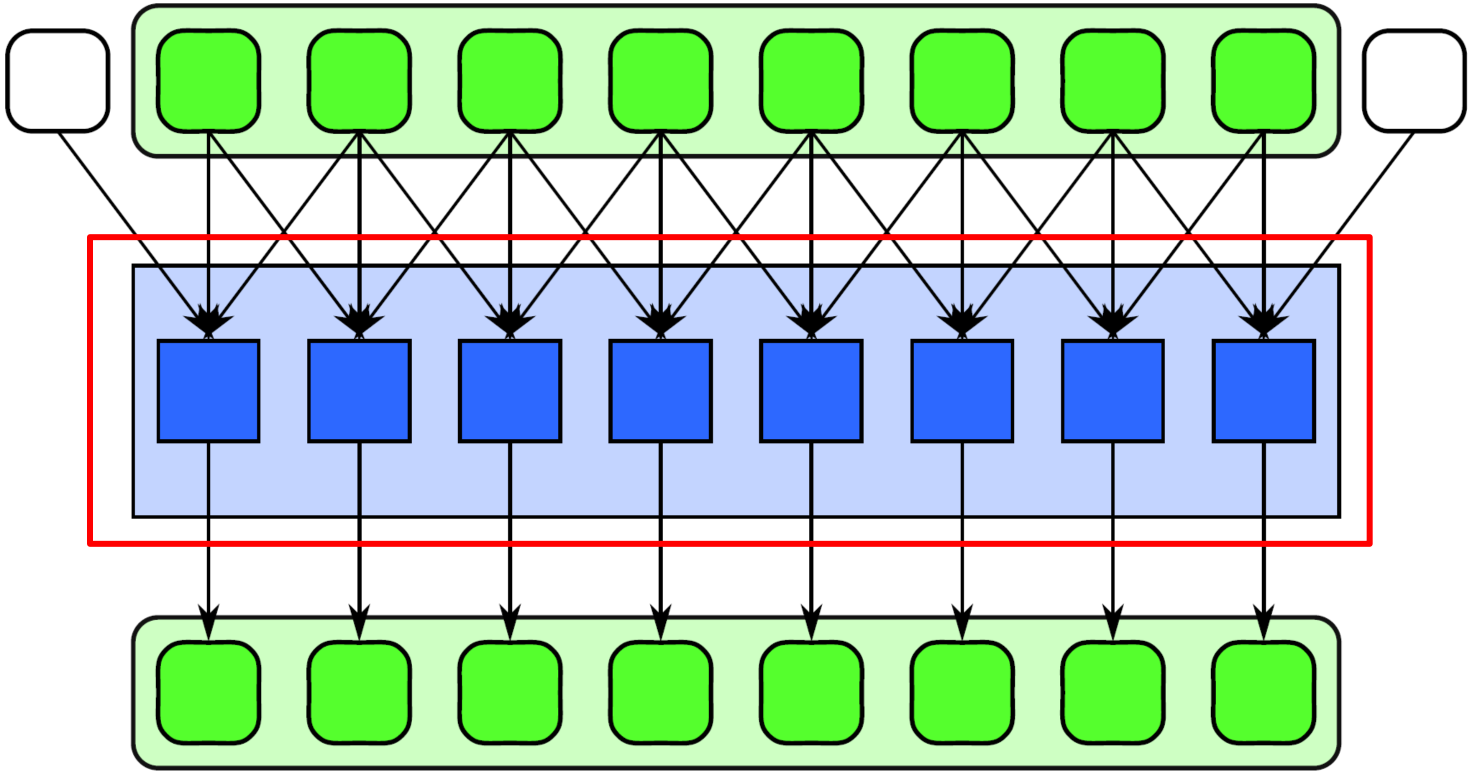
Stencil Pattern with In Place Update

Input array

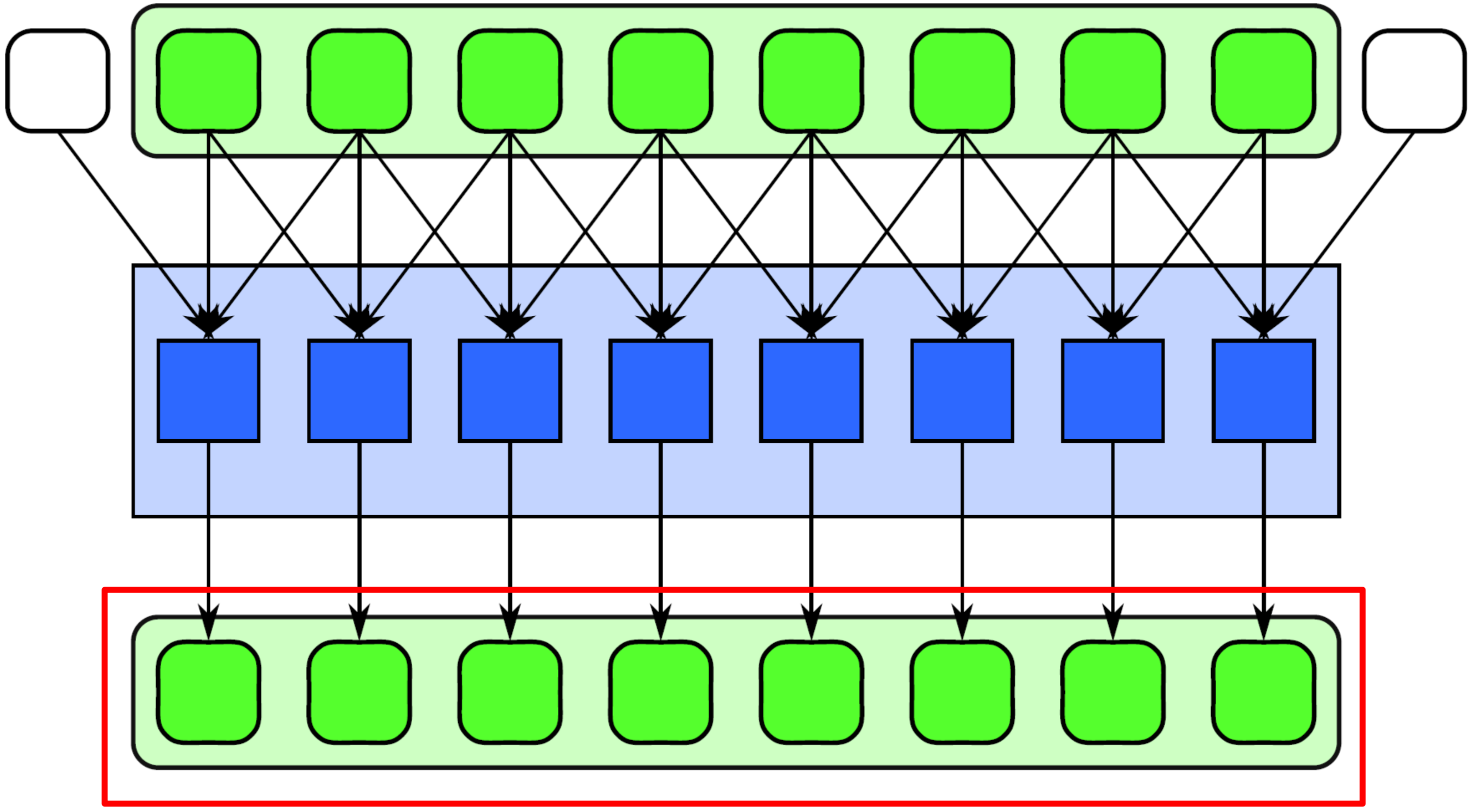


Stencil Pattern with In Place Update

Function



Stencil Pattern with In Place Update



Input Array !!!

Stencil Example

- Here is our array, A

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

Stencil Example

- Here is our array **A**
- Update A in place
- Apply a stencil operation to the inner square of the form:

```
A(i,j) = avg( A(i,j),
              A(i-1,j), A(i+1,j),
              A(i,j-1), A(i,j+1)
        )
```

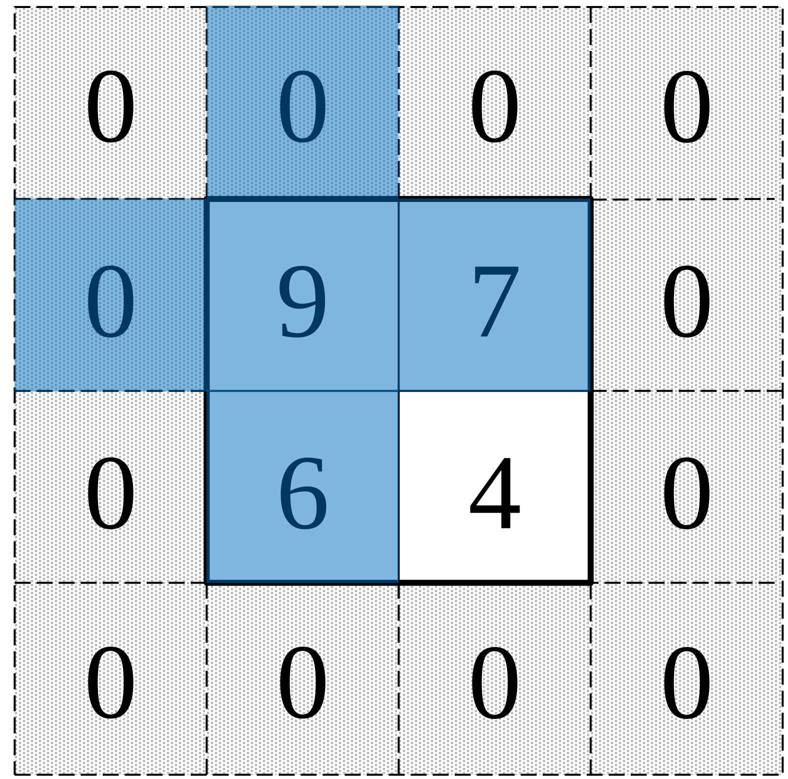
A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

What is the stencil?

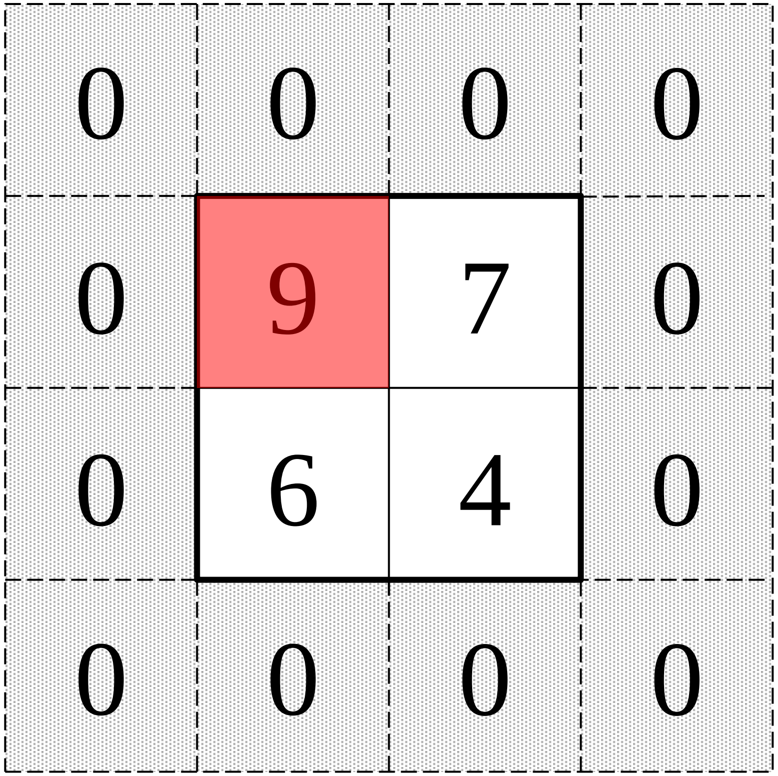
Stencil Pattern Procedure

1) Average all blue squares



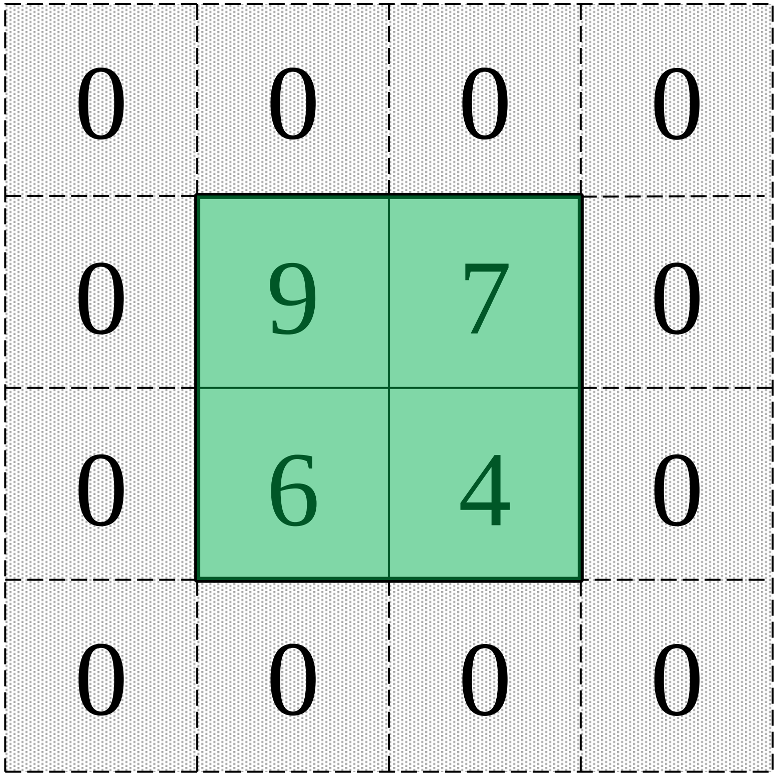
Stencil Pattern Procedure

1) Average all blue squares
2) Store result in red square



Stencil Pattern Procedure

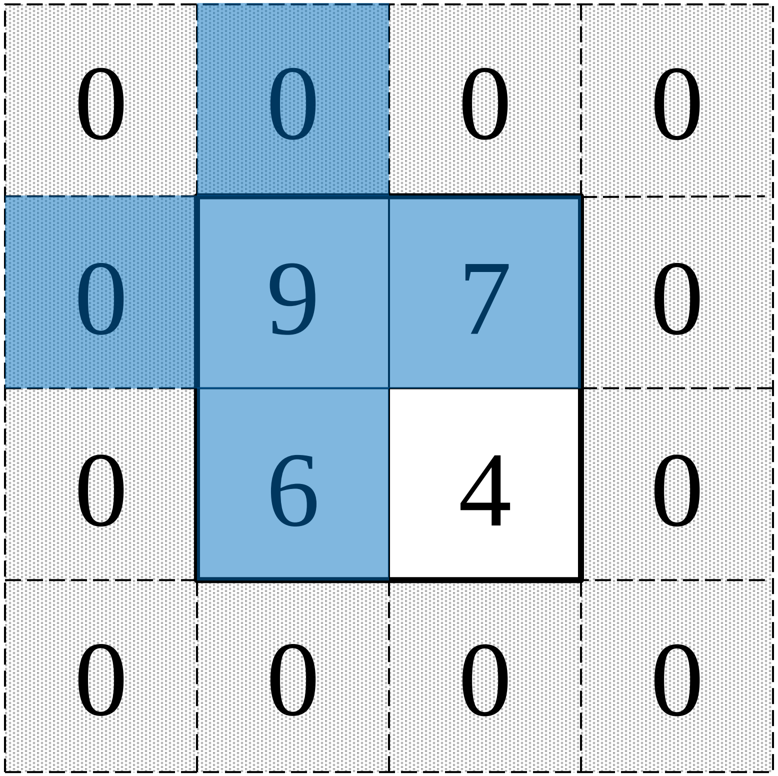
1) Average all blue squares
2) Store result in red square
3) Repeat 1 and 2 for all green squares



Practice!

Stencil Pattern Practice

A



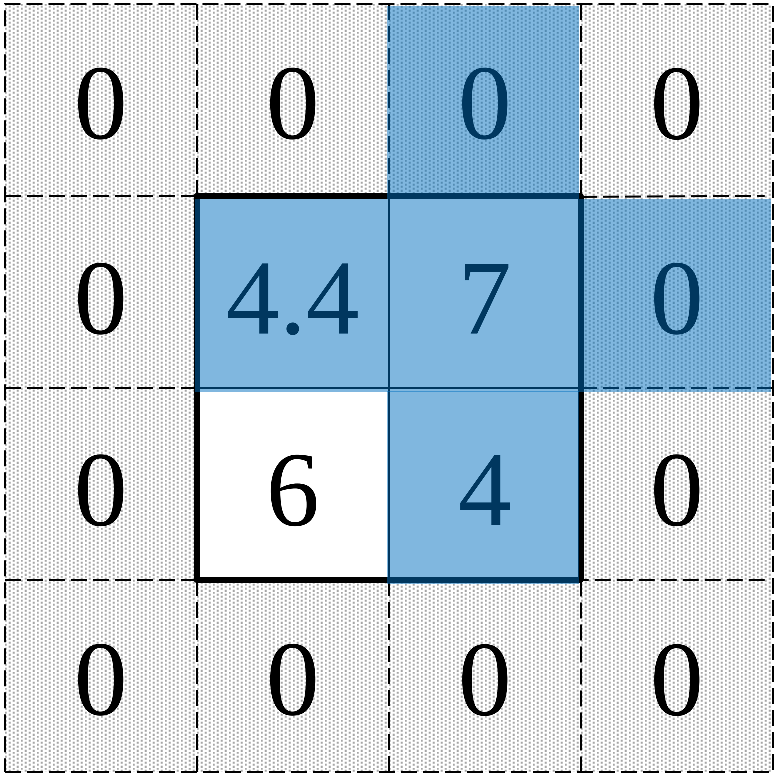
Stencil Pattern Practice

A

0	0	0	0
0	4.4	7	0
0	6	4	0
0	0	0	0

What is the stencil pattern?

A



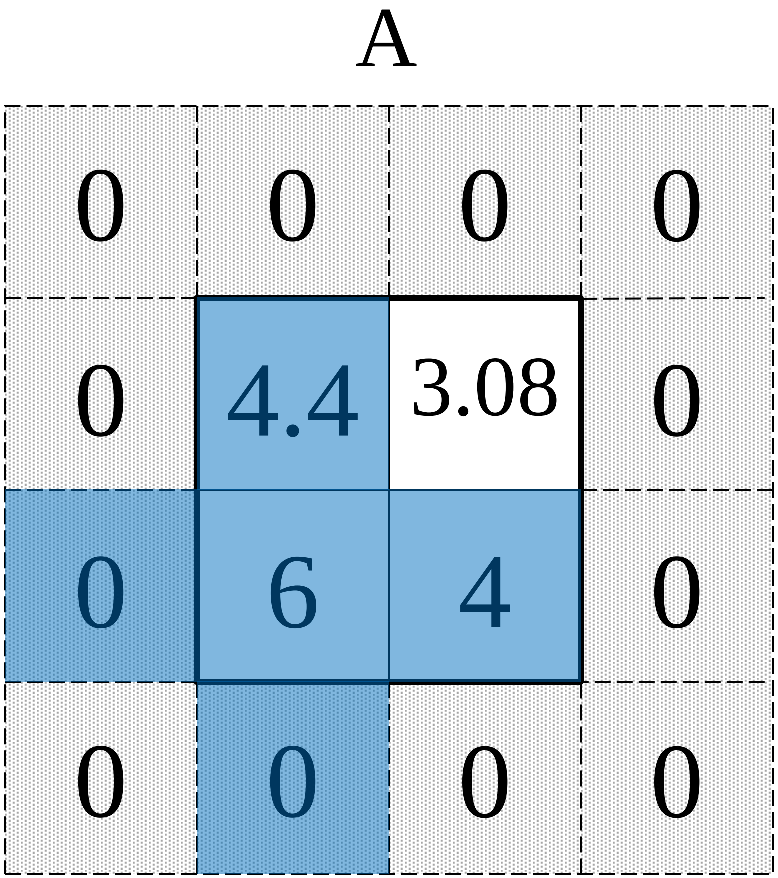
What is the stencil pattern?

A

0	0	0	0
0	4.4	3.08	0
0	6	4	0
0	0	0	0

43

What is the stencil pattern?



What is the stencil pattern?

A

0	0	0	0
0	4.4	3.08	0
0	2.88	4	0
0	0	0	0

What is the stencil pattern?

A

0	0	0	0
0	4.4	3.08	0
0	2.88	4	0
0	0	0	0

What is the stencil pattern?

A

0	0	0	0
0	4.4	3.08	0
0	2.88	1.992	0
0	0	0	0

Different Cases

Separate output array

Input

9	7
6	4

→

Output

4.4	4.0
3.8	3.4

Updates occur in place

Input

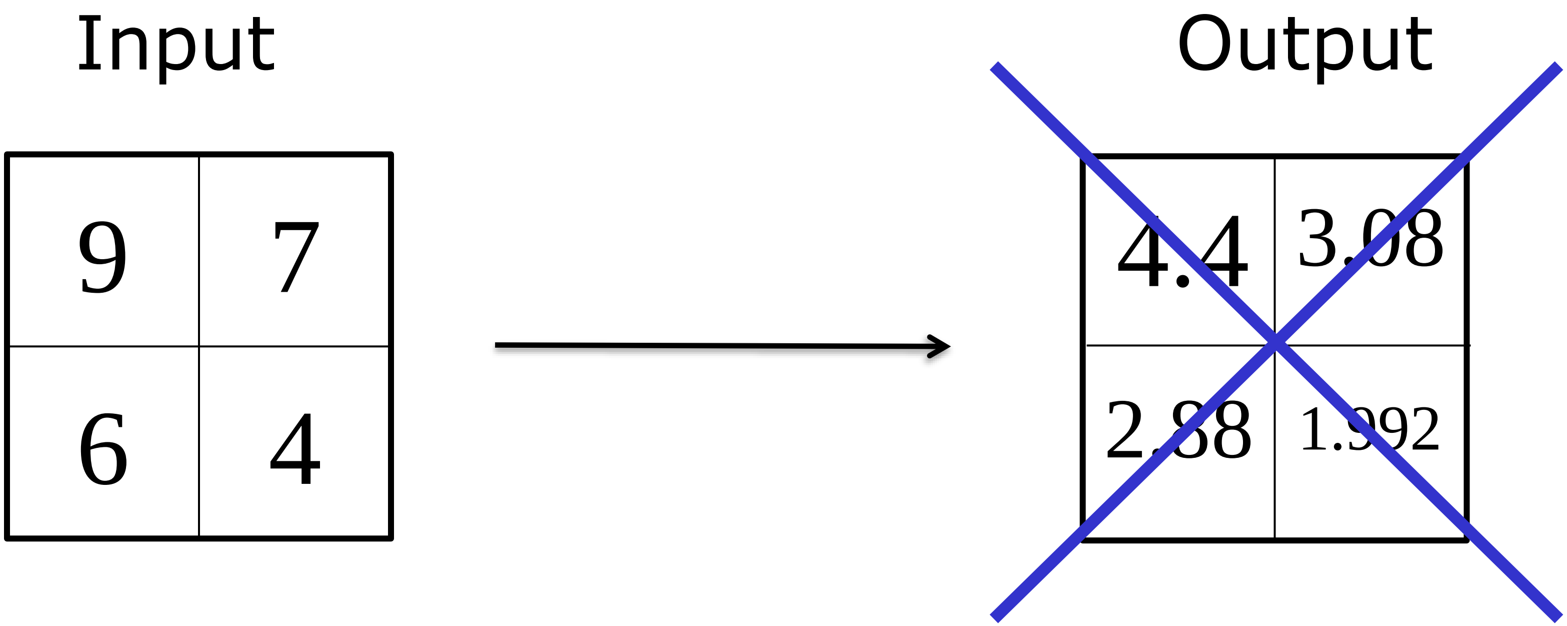
9	7
6	4

→

Output

4.4	3.08
2.88	1.992

Which is correct?



Is this output incorrect?

Outline

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Iterative Codes

- Iterative codes are ones that update their data in steps
  - At each step, a new value of an element is computed using a formula based on other elements
  - Once all elements are updated, the computation proceeds to the next step or completes
- Iterative codes are most commonly found in computer simulations of physical systems for scientific and engineering applications
  - Computational fluid dynamics
  - Electromagnetics modeling
- They are often applied to solve partial differential equations
  - Jacobi iteration
  - Gauss-Seidel iteration
  - Successive over relaxation

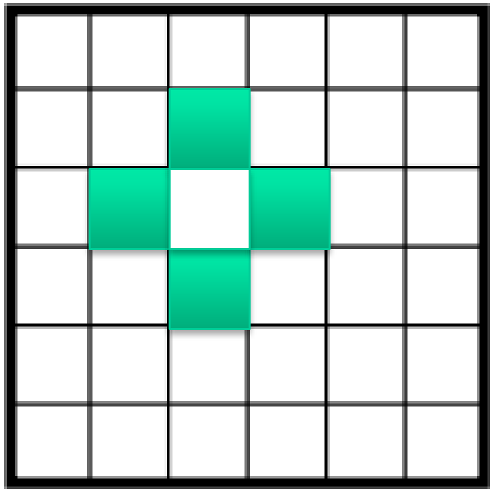
Iterative Codes and Stencils

- Stencils essentially define which elements are used in the update formula
- Because the data is organized in a regular manner, stencils can be applied across the data uniformly

Simple 2D Example

- Consider the following code

```
for k=1, 1000
  for i=1, N-2
    for j = 1, N-2
      a[i][j] = 0.25 * (a[i][j]+ a[i-1][j]
                        + a[i+1][j]
                        + a[i][j-1]
                        + a[i][j+1])
    }
  }
}
```



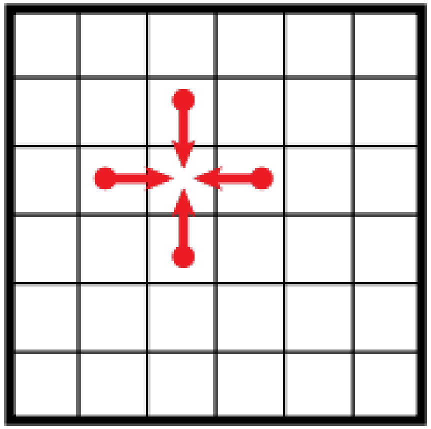
5-point stencil

Do you see anything interesting?

How would you parallelize?

2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge
- Here we are using a 4-point stencil

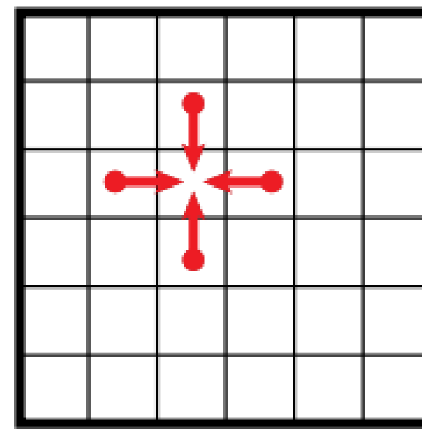


4-point stencil

- It is different from before because we want to update all array elements simultaneously … How?

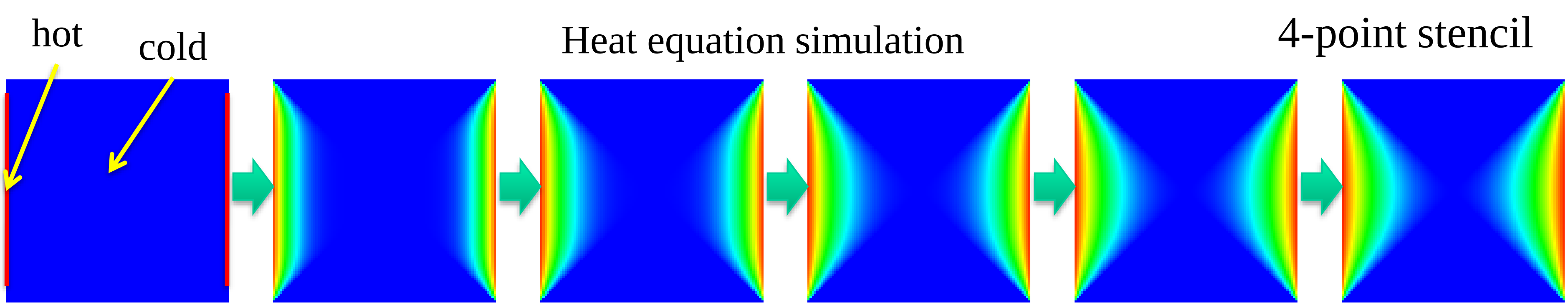
2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge



4-point stencil

Heat equation simulation



Step 0 | Step 200 | Step 400 | Step 600 | Step 800 | Step 1000

Outline

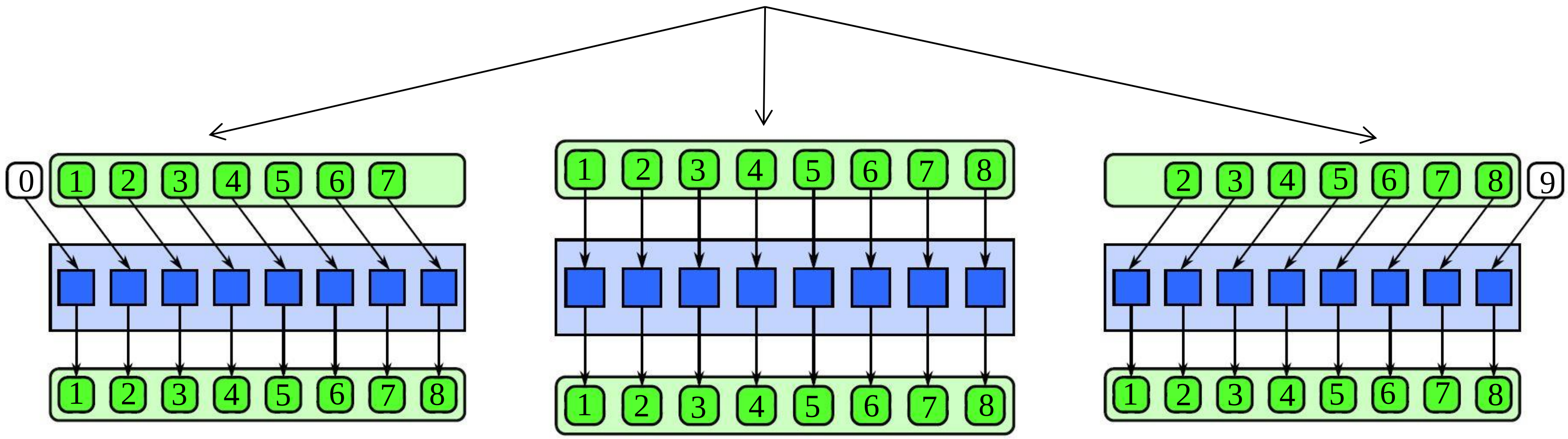
- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Implementing Stencil with Shift

- One possible implementation of the stencil pattern includes shifting the input data
- For each offset in the stencil, we gather a new input vector by **shifting** the original input by the offset amount

Implementing Stencil with Shift

All input arrays are derived from the same original input array



Implementing Stencil with Shift

- This implementation is only beneficial for one dimensional stencils or the memory-contiguous dimension of a multidimensional stencil

- Memory traffic to external memory is not reduced with shifts

- But, shifts allow vectorization of the data reads, which may reduce the total number of instructions

Contents

Stencil and Cache Optimizations

- Assuming 2D array where rows are contiguous in memory…
  - Horizontally related data will tend to belong to the same cache line
  - Vertical offset accesses will most likely result in cache misses

Stencil and Cache Optimizations

- Assigning rows to cores:
  - Maximizes horizontal data locality
  - Assuming vertical offsets in stencil, this will create redundant reads of adjacent rows from each core
- Assigning columns to cores:
  - Redundantly read data from same cache line
  - Create false sharing as cores write to same cache line

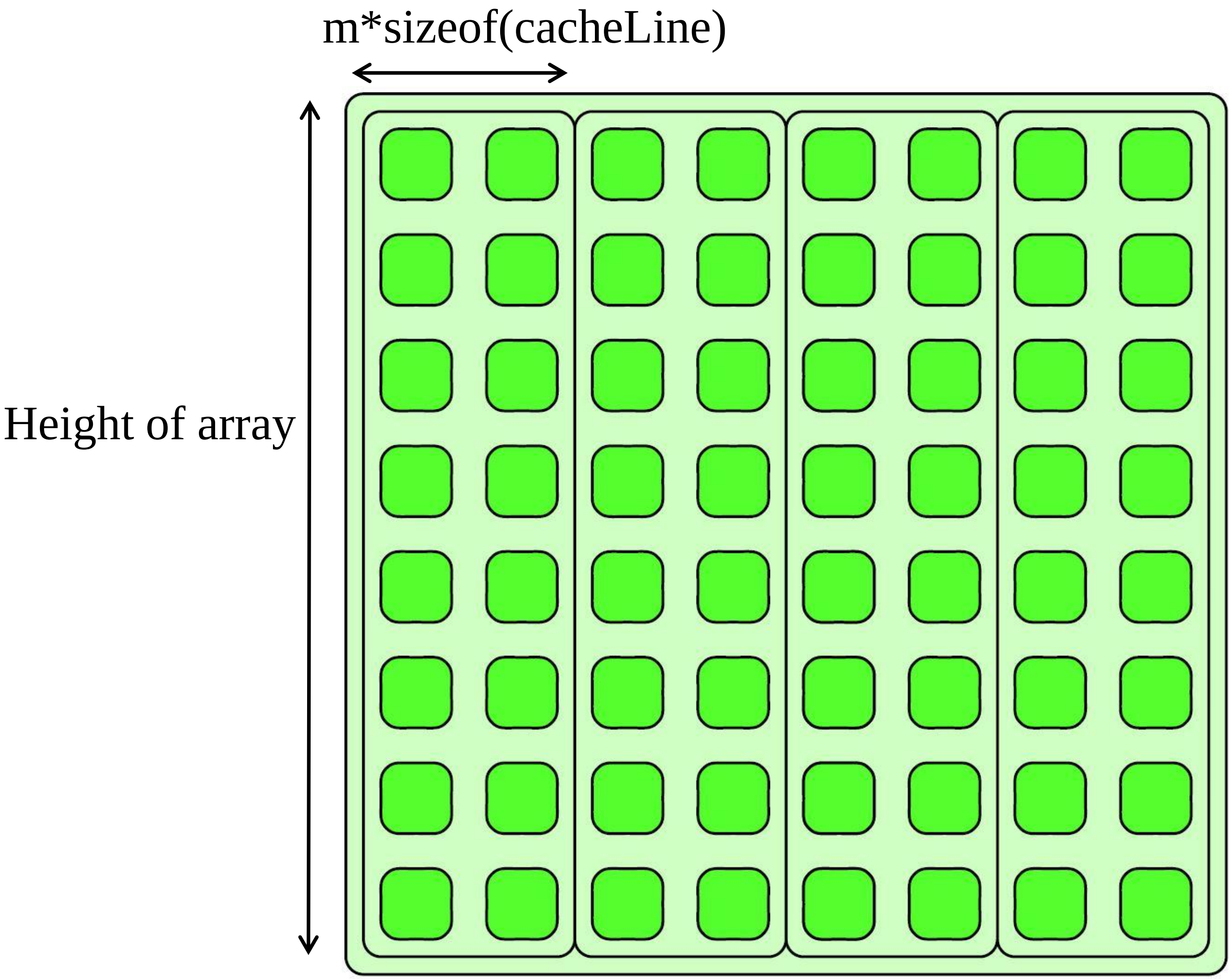
Stencil and Cache Optimizations

- Assigning "strips" to each core can be a better solution
- **Strip-mining**: an optimization in a stencil computation that groups elements in a way that avoids redundant memory accesses and aligns memory accesses with cache lines

Stencil and Cache Optimizations

- A strip's size is a multiple of a cache line in width, and the height of the 2D array
- Strip widths are in increments of the cache line size so as to avoid false sharing and redundant reads
- Each strip is processed serially from top to bottom within each core

Stencil and Cache Optimizations



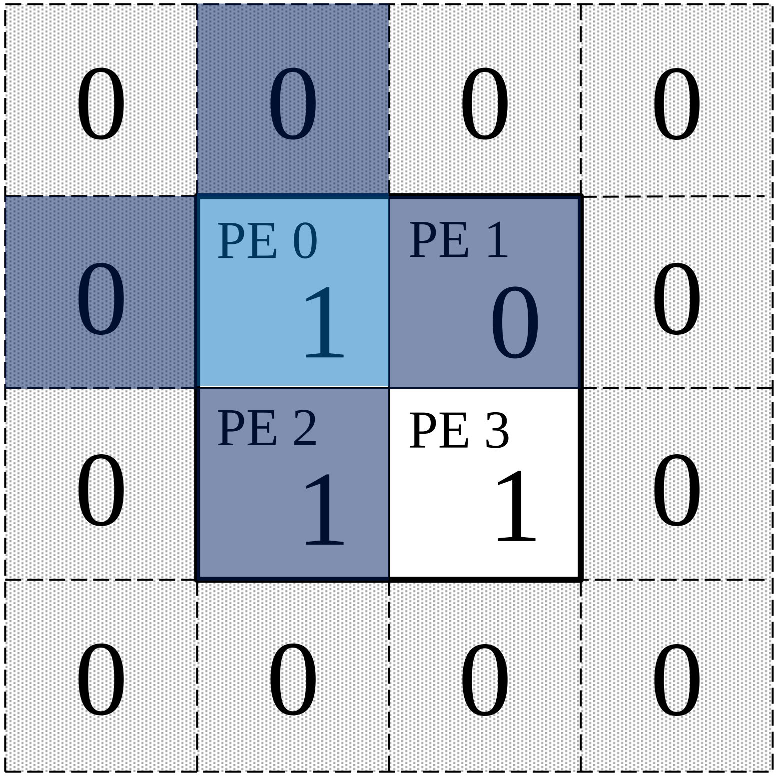
Outline

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Stencil and Communication Optimizations

- When data is distributed, ghost cells must be **explicitly** communicated among nodes between loop iterations

- Darker cells are PE 0's ghost cells
- After first iteration of stencil computation
  - PE 0 must request PE 1 & PE 2's stencil results
  - PE 0 can perform another iteration of stencil



Stencil and Communication Optimizations

- Generally better to replicate ghost cells in each local memory and swap after each iteration than to share memory
  - Fine-grained sharing can lead to increased communication cost

Stencil and Communication Optimizations

- **Halo**: set of all ghost cells
- Halo must contain all neighbors needed for one iteration
- Larger halo (**deep halo**)
  - Trade off
    - less communications and more independence, but…
    - more redundant computation and more memory used

- **Latency Hiding**: Compute interior of stencil while waiting for ghost cell updates