Contents

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
  - SOR and Red/Black SOR

- Partitioning and parallelization

- Implementing stencil with shift

- Stencil and cache optimizations

- Stencil and communication optimizations

- Recurrence
Contents

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
  - SOR and Red/Black SOR
- Partitioning and parallelization
- 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 for all parallel operations.
Serial Stencil Example (part 1)

```cpp
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)

```java
    // 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?
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 multi-dimensional 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
- Stencil patterns derive from the numerical solution requirements of particular computational problems
  - Partial differential equations are a main type
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)

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

24-point stencil
(25-point stencil)
Stencil Example

- Here is our array, A

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 & 0 \\
0 & 6 & 4 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
Stencil Example

- Here is our array $A$
- Assume an output array $B$
  - Initialize to all 0
- Apply a stencil operation to the inner square of the form:
  \[
  B(i,j) = \text{avg}( A(i,j), A(i-1,j), A(i+1,j), A(i,j-1), A(i,j+1) )
  \]
- What is the stencil?
1) Average all blue squares
Stencil Pattern Procedure

1) Average all blue squares
2) Store result in B
**Stencil Pattern Procedure**

1) Average all blue squares
2) Store result in B
3) Repeat Step 1 and Step 2 for all green squares
Practice!
Stencil Pattern Practice

A

B

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 \\
0 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 4.4 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]
Stencil Pattern Practice

A

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 \\
0 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

B

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 4.4 & 4.0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]
Stencil Pattern Practice

A

B

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4.4</td>
<td>4.0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Stencil Pattern Practice

**A**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>9</th>
<th>7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>7</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**B**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>4.4</th>
<th>4.0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4.4</td>
<td>4.0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3.8</td>
<td>3.4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**What about the boundaries?**

**How is this parallelized?**
Contents

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
  - SOR and Red/Black SOR
- Partitioning and parallelization
- 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)

```c
// 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] = 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
Stencil Pattern with In Place Update

Input Array !!

Problems?
• Here is our array, A
Stencil Example

- Here is our array $A$
- Update $A$ in place
- Apply a stencil operation to the inner square of the form:
  \[
  A(i,j) = \text{avg}(A(i,j), A(i-1,j), A(i+1,j), A(i,j-1), A(i,j+1))
  \]
- What is the stencil?
1) Average all blue squares
**Stencil Pattern Procedure**

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Stencil Pattern Procedure

1) Average all blue squares
2) Store result in red square
3) Repeat Step 1 and Step 2 for all green squares
Practice!
Stencil Pattern Practice

A

\[ \begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 \\
0 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{array} \]
## Stencil Pattern Practice

![Stencil Pattern Diagram](image)

### Matrix A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4.4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
What is the stencil pattern?

A

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 4.4 & 7 & 0 \\
0 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
What is the stencil pattern?
What is the stencil pattern?

A

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 4.4 & 3.1 & 0 \\
0 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]
What is the stencil pattern?

\[
\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 4.4 & 3.1 & 0 \\
0 & 2.9 & 4 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]
What is the stencil pattern?

\[ A = \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 4.4 & 3.1 & 0 \\
0 & 2.9 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix} \]
What is the stencil pattern?

A

0 0 0 0
0 4.4 3.1 0
0 2.9 2.0 0
0 0 0 0
### Different Cases

<table>
<thead>
<tr>
<th>Separate output array</th>
<th>Updates occur in place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: 9</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Input: 9</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
Which is correct?

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 7</td>
<td>4.4 3.1</td>
</tr>
<tr>
<td>6 4</td>
<td>2.9 2.0</td>
</tr>
</tbody>
</table>

Is this output incorrect?
Contents

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
  - SOR and Red/Black SOR
- Partitioning and parallelization
- 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 commonly found in computer simulations of physical systems for science 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 (SOR)
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.
- Iterations in iterative codes are typically associated with steps forward in time.
  - Stencils are applied to compute a new value at a later time from old values at an earlier time.
Simple 2D Example

Consider the following code

```plaintext
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])
    }
}
}
```

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
- 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

Heat equation simulation

4-point stencil

Step 0  Step 200  Step 400  Step 600  Step 800  Step 1000
Successive Over Relaxation (SOR)

- SOR is an alternate method of solving partial differential equations
- While the Jacobi iteration scheme is very simple and parallelizable, its slow convergent rate renders it impractical for any "real world" applications
- One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation
- It also allows a method known as Red-Black SOR to be used to enable parallel updates in place
Red / Black SOR

Pass 1: Writing to red cells, reading from black

Pass 2: Writing to black cells, reading from red
Red / Black SOR
Contents

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

- Data is divided into
  - Non-overlapping regions (avoid write conflicts, data races)
  - Equal-sized regions (improve load balancing)
Partitioning

- Data is divided into
  - Non-overlapping regions (avoid write conflicts, data races)
  - Equal-sized regions (improve load balancing)
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 multi-dimensional 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

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

m*sizeof(cacheLine)

Height of array
Contents

- Partitioning
- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
  - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence
But first... Conway’s Game of Life

- The **Game of Life** is a cellular automaton created by John Conway in 1970.
- The evolution of the game is entirely based on the input state – zero player game.
- To play: create initial state, observe how the system evolves over successive time steps.

2D landscape
Conway’s Game of Life

- Typical rules for the Game of Life
  - Infinite 2D grid of square cells, each cell is either “alive” or “dead”
  - Each cell will interact with all 8 of its neighbors
    - Any cell with < 2 live neighbors dies (under-population)
    - Any cell with 2 or 3 live neighbors lives to next gen.
    - Any cell with > 3 live neighbors dies (overcrowding)
    - Any dead cell with 3 live neighbors becomes a live cell
Conway’s Game of Life: Examples
Conway’s Game of Life

- The Game of Life computation can easily fit into the stencil pattern!
- Each larger, black box is owned by a thread
- What will happen at the boundaries?
Conway’s Game of Life

- We need some way to preserve information from the previous iteration without overwriting it
- *Ghost Cells* are one solution to the boundary and update issues of a stencil computation
- Each thread keeps a copy of neighbors’ data to use in its local computations
- These ghost cells must be updated after each iteration of the stencil
Conway’s Game of Life

- Working with ghost cells
Conway’s Game of Life

- Working with ghost cells
Conway’s Game of Life

- Working with ghost cells

Compute the new value for this cell
Conway’s Game of Life

- Working with ghost cells

Five of its eight neighbors already belong to this thread.

But three of its neighbors belong to a different thread.
Conway’s Game of Life

- Working with ghost cells

Before any updates are done in a new iteration, all threads must update their ghost cells.
Conway’s Game of Life

- Working with ghost cells

Data this thread can use (including ghost cells from neighbors)
Conway’s Game of Life

- Working with ghost cells

Updated cells
Conway’s Game of Life

☐ Things to consider…

○ What might happen to our ghost cells as we increase the number of threads?
  ◆ ghost cells to total cells ratio will rapidly increase causing a greater demand on memory

○ What would be the benefits of using a larger number of ghost cells per thread? Negatives?
  ◆ in the Game of Life example, we could double or triple our ghost cell boundary, allowing us to perform several iterations without stopping for a ghost cell update
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


**Seismic Raytracing**


---

Calculate shortest travel time paths to sensors

818 points!

---
Table of Contents

- What is the stencil pattern?
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence
Recurrence

- What if we have several nested loops with data dependencies between them when doing a stencil computation?
Recurrence

```c
void my_recurrence(
    size_t v,  // number of elements vertically
    size_t h,  // number of elements horizontally
    const float a[v][h],  // input 2D array
    float b[v][h]  // output 2D array (boundaries already initialized)
) {
    for (int i=1; i<v; ++i)
        for (int j=1; j<h; ++j)
            b[i][j] = f(b[i-1][j], b[i][j-1], a[i][j]);
}
```
Recurrence

```
void my_recurrence(
    size_t v, // number of elements vertically
    size_t h, // number of elements horizontally
    const float a[v][h], // input 2D array
    float b[v][h] // output 2D array (boundaries already initialized )
) {
    for (int i=1; i<v; ++i)
        for (int j=1; j<h; ++j)
            b[i][j] = f(b[i-1][j], b[i][j-1], a[i][j]);
```

Data dependencies between loops
Recurrence

- This can still be parallelized!
- Trick: find a plane that cuts through grid of intermediate results
  - Previously computed values on one side of plane
  - Values to still be computed on other side of plane
  - Computation proceeds perpendicular to plane through time (this is known as a sweep)
- This plane is called a *separating hyperplane*
Recurrence

Hyperplanes

Iteration 1
Iteration 2
Iteration 3
Recurrence

- Same grid of intermediate results
- Each level corresponds to a loop iteration
- Computation proceeds downward
Conclusion

- Examined the stencil and recurrence pattern
  - Both have a regular pattern
    - of communication
    - of data access
- In both patterns we can convert a set of offset memory accesses to shifts
- Stencils can use strip-mining to optimize cache use
- Ghost cells should be considered when stencil data is distributed across different memory spaces