Efficient stencil computations in Accelerate

Josh Meredith, UNSW

Stencil computations

- Like map operation, but including neighbouring elements

Input:

1	2	3	4
5	6	7	8
9	10	11	12

Map:

f(1)	f(2)	f(3)	f(4)
f(5)	f(6)	f(7)	f(8)
f(9)	f(10)	f(11)	f(12)

f = function of current element

Stencil computations

- Like map operation, but including neighbouring elements

Input:

1	2	3	4
5	6	7	8
9	10	11	12

Map:

f(1)	f(2)	f(3)	f(4)
f(5)	f(6)	f(7)	f(8)
f(9)	f(10)	f(11)	f(12)

f = function of current element

Stencil:

…	…	…	…
…	g(…)	…	…
…	…	…	…

g = function of neighbouring elements

Stencil computations

- Why are they useful?
  - Image processing (e.g. photoshop filters)
  - Scientific applications (numerical simulations)

How do we implement these efficiently?

Benchmark example

High level Accelerate (Haskell) stencil:

```
benchmark :: Stencil3x3 Float -> Exp Float
benchmark = ((x,t,y)
            ,(l,c,r)
            ,(z,b,w)) = 4 * c + x - t + y + l + r + z - b + w
```

	1	-1	1		
	1	4	1		
	1	-1	1		

Benchmarks (teaser)

	Time (ms)	Runtime vs C++ (x slower)
Accelerate (current)	3052	28x
C++ (hand optimised)	110	1x
This work	112	1x

Problem #1: nested loops

- Accelerate currently uses general, but slow, 1-dimensional indices
- We want an efficient implementation for 2D stencils, which are very common

Problem #1: nested loops

1-dimensional:

0	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**

i = [0, width * height)

x = i `mod` width

y = i `div` width

- Loop over i
- When we need x and y, calculate them with mod and div

Problem #1: nested loops

2-dimensional:

(0,0)	**(0,1)**	**(0,2)**	**(0,3)**
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**
(2,0)	**(2,1)**	**(2,2)**	**(2,3)**

x = [0, width)

y = [0, height)

i = y * width + x

- Loop over x and y
- When we need i, calculate it based on the x, y and width

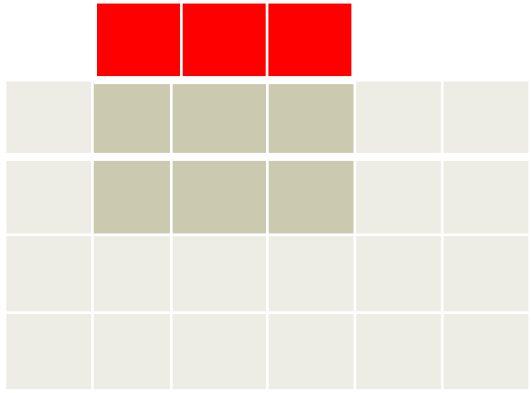
Summary problem #1

	Time (ms)	Speedup (step)	Speedup (total)
1D loop	3242	-	1

Summary problem #1

	Time (ms)	Speedup (step)	Speedup (total)
1D loop	3242	-	1
2D loop	1153	2.8	2.8

Problem #2: bounds checking

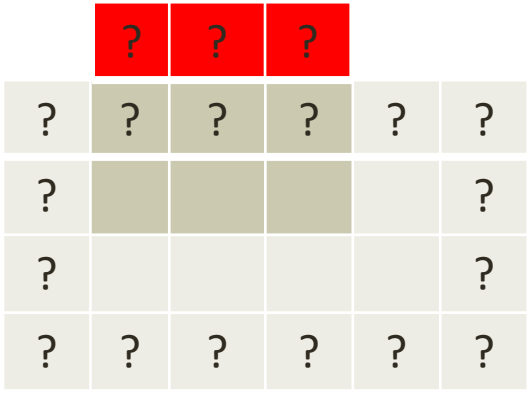


What do we do at the edges?

Problem #2: bounds checking

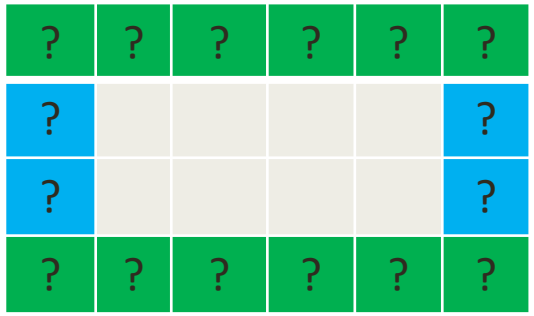
Naive solution: check and apply the boundary condition on all reads

Problem #2: bounds checking



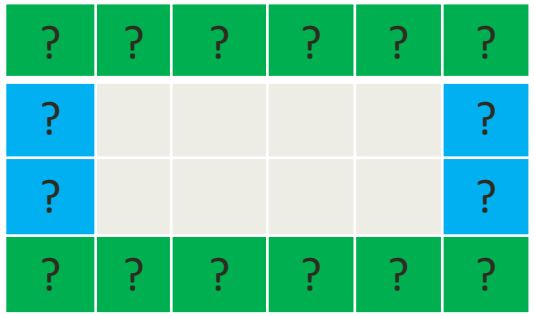
Better solution: only check the boundary condition when computing border elements

Problem #2: bounds checking



Better solution: only check the boundary condition when computing border elements

Bounds checking: corners



- Accelerate arrays are row major
- So we should include the corners with the top and bottom

Summary problem #2

	Time (ms)	Speedup (step)	Speedup (total)
1D loop	3242	-	1
2D loop	1153	2.8	2.8
Bounds checking	144	8	22.5

Problem #3: Tiling

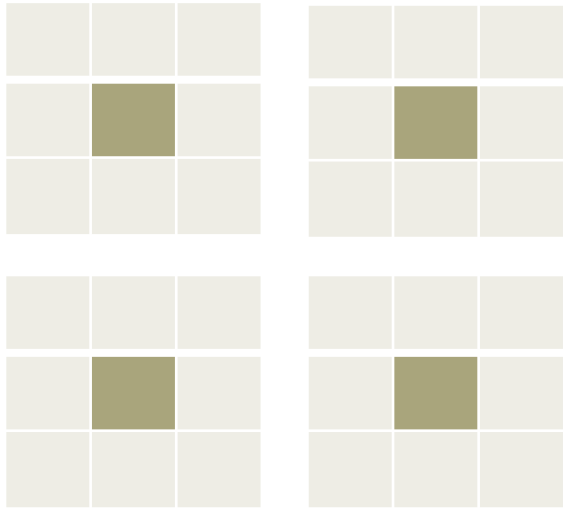
- Reading neighbouring elements means we have a lot of duplicated reads

Solution: 1x1 tiles

- Does LLVM make tiling unnecessary?
- Let’s compare

Solution: 2x2 tiles

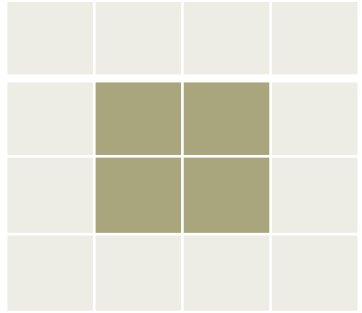
Before tiling:



36 reads

4 writes

After tiling:

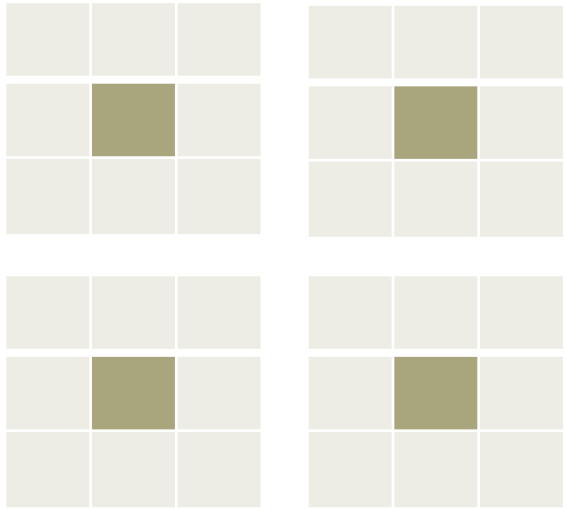


16 reads

4 writes

Solution: 2x2 tiles

Before tiling:

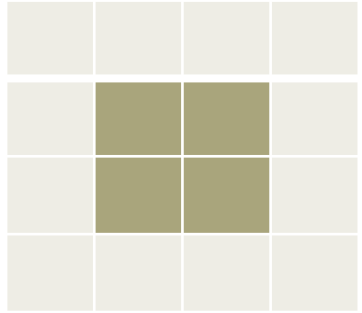


36 reads

4 writes

144ms

After tiling:



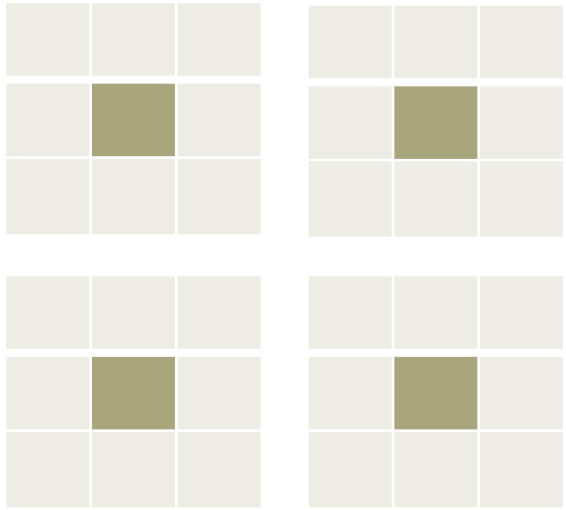
16 reads

4 writes

350ms

Solution: 1x4 tiles

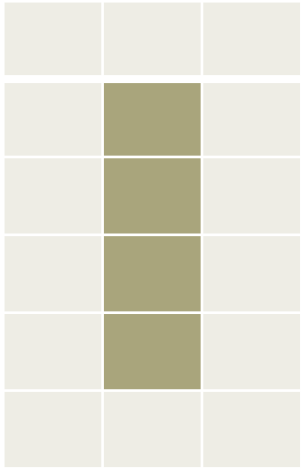
Before tiling:



36 reads

4 writes

After tiling:

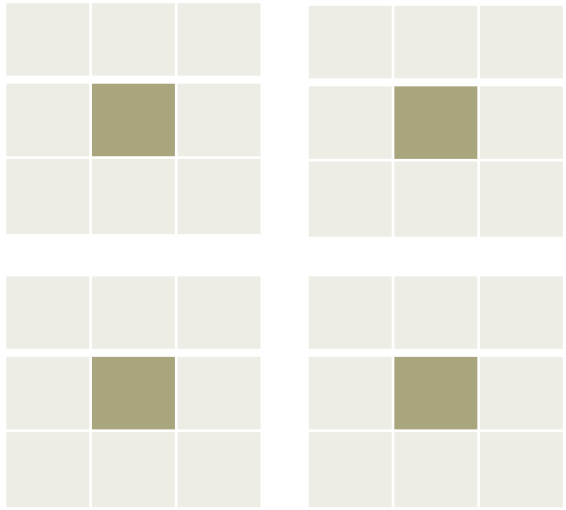


18 reads

4 writes

Solution: 1x4 tiles

Before tiling:

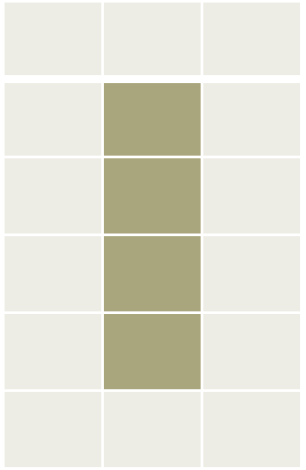


36 reads

4 writes

144ms

After tiling:



18 reads

4 writes

110ms

Summary problem #3

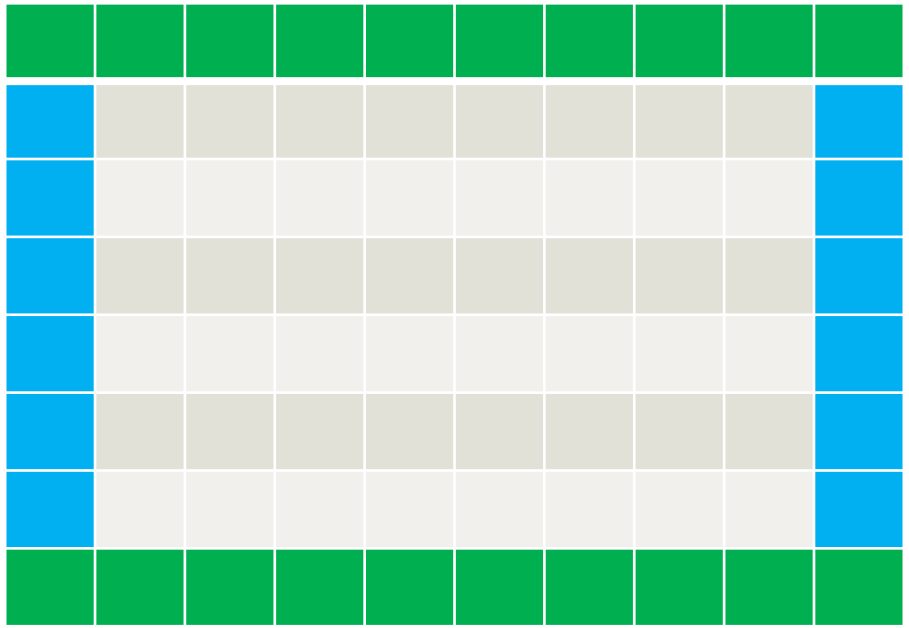
	Time (ms)	Speedup (step)	Speedup (total)
1D loop	3242	-	1
2D loop	1153	2.8	2.8
Bounds checking	144	8	22.5
1x4 tiles	110	1.3	29.5

Problem #4: Parallelisation

- How should we evaluate this in parallel?

Problem #4: Parallelisation

- Evaluate the edge regions sequentially
- Then, evaluate the main region in parallel using Accelerate's work stealer



Problem #4: Parallelisation

- No performance gains for this stencil
- Our example is very memory bottlenecked
- Many common stencils are similarly memory bottlenecked

Summary evaluation strategy

- Evaluate in several regions:
  - Top & bottom, including corners, with bounds checking
  - Left & right, excluding corners, with bounds checking
  - Middle, without bounds checking
    - Tile several elements at once on the outer axis
    - Also parallelise across the outer (y) axis
    - Vectorise across the inner (x) axis with LLVM