cuNumeric

CS315B

Lecture 3

Overview

- An important special case of programming involves ``bulk'' operations on collections of data

- Examples
  - Find all the words in a document
  - Adjust the color of every pixel of an image
  - Analyze every sale of a product
  - Update every entry in a spreadsheet

Array Programming

- Often the data types involved in such bulk operations are arrays:
  - An image is an array of pixels
  - A column of a database/spreadsheet is an array of some type
  - A document is an array of characters

- *Array programming languages* focus on whole-array operations

- Advantages:
  - Conciseness: Bulk operations over the entire array
    - No explicit looping
    - Iteration/recursion is “baked in” to the operations
  - Performance: Leave the details of the implementation the underlying system
    - Might be very different for different hardware, e.g., CPUs or GPUs

History

- First array programming language was APL
  - Designed by Ken Iverson in the 1960’s
- Designed for expressing pipelines of operations on bulk data
  - Basic data type is the multidimensional array
- Trivia: Special APL keyboards accommodated the many 1 character built-ins
  - APL programs can be unreadable strings of Greek letters
- Highly influential
  - On functional programming (several languages)
  - And array programming (Matlab, R, NumPy)

```
{(+⌿ω)÷≢ω}
```

NumPy

- The most popular array programming interface today is NumPy

- A library of array operations
  - And also syntax for defining *views* of arrays

- Python has many other, non-array features
  - So NumPy programs tend to mix styles, including using variables, state, etc.

A Brief NumPy Tutorial

A short overview of NumPy arrays

- Defining
- Shape
- Views
- Filters

Using NumPy

```
# This line will always appear in a NumPy program
import numpy as np
```

Defining an Array

```
import numpy as np

# initialize an array A of 10 elements with the integers 0..9
A = np.arange(0,10)
```

Example: Adding Arrays

```
import numpy as np
A = np.arange(0,10)

# addition is pointwise if the dimensions match
np.add(A,A)
```

Reshaping

```
import numpy as np
A = np.arange(0,10)

# Reshaping is a general operation that changes array dimensions.
# Normally defines a view: creates a new way of naming the array but does
# not make a copy.

# view the elements of A as a 2x5 array
A.reshape(2,5)

# view the elements of A as a 10x1 (column) array
A.reshape(10,1)
```

Example: Outer Product

```
import numpy as np
A = np.arange(0,10)

# We can use a combination of reshape and broadcast to define a
# concise outer product.
# Broadcasting duplicates elements to make array arguments match

np.multiply(A,A.reshape(10,1))
```

Broadcasting

- Broadcasting is used to make two (or more) array arguments to a NumPy operator conformable

- In general, if the dimensions of two arrays do not match and the smaller dimension is 1, then that dimension in the smaller array is copied to match the dimension of the larger array

```
A = np.array([1, 2, 3])
B = 4
A * B
```

Slicing

```
import numpy as np
A = np.arange(0,10)

# slicing defines views of subsets of an array
A[3:]     # slice of 4th element to the end of the array
A[:-3]    # slice up to the 4th element from the end of the array
A[1:-1]  # slice of all but the first and last elements of the array
A.reshape(2,5)[:,1:3]     # slicing in multiple dimensions
A.reshape(2,5)[0:2,1:3] # same slice written a different way
```

Example: Moving Average

```
import numpy as np
A = np.arange(0,10)

# cumulative sum is one of many NumPy built-in array functions
B = np.cumsum(A)

# moving average of A with a window of size 3
(B[3:] – B[:-3] )  / 3.0
```

Masks

```
import numpy as np
A = np.arange(0,10)

# Using an array in a predicate returns an array of Boolean results
# Here broadcasting promotes 5 to a 1D array of 5's
A > 5
A <= 5
(2 * A) == (A ** 2)
```

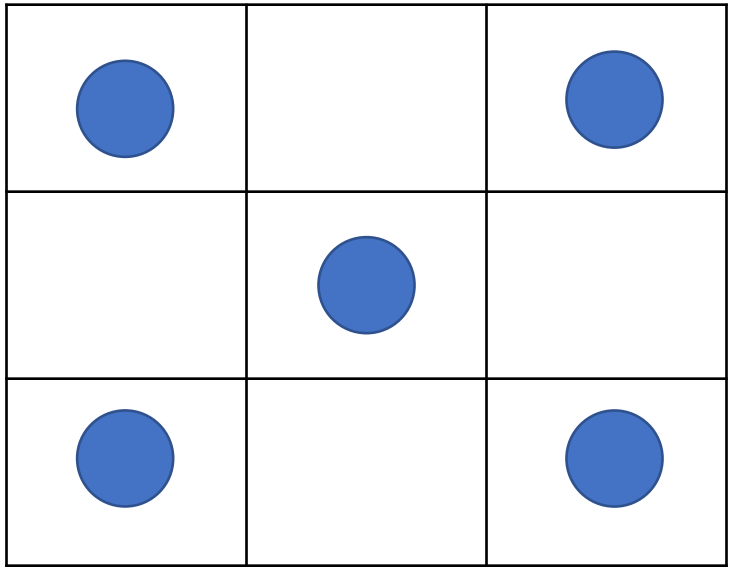
Filters

```
import numpy as np
A = np.arange(0,10)

# Boolean arrays can be used as array indices to filter arrays
A[A > 5]                  # elements of A that are > 5
A[A <= 5]                 # elements of A that are <= 5
A[(2 * A) == (A ** 2)]    # elements x of A where 2*x == x ** 2
```

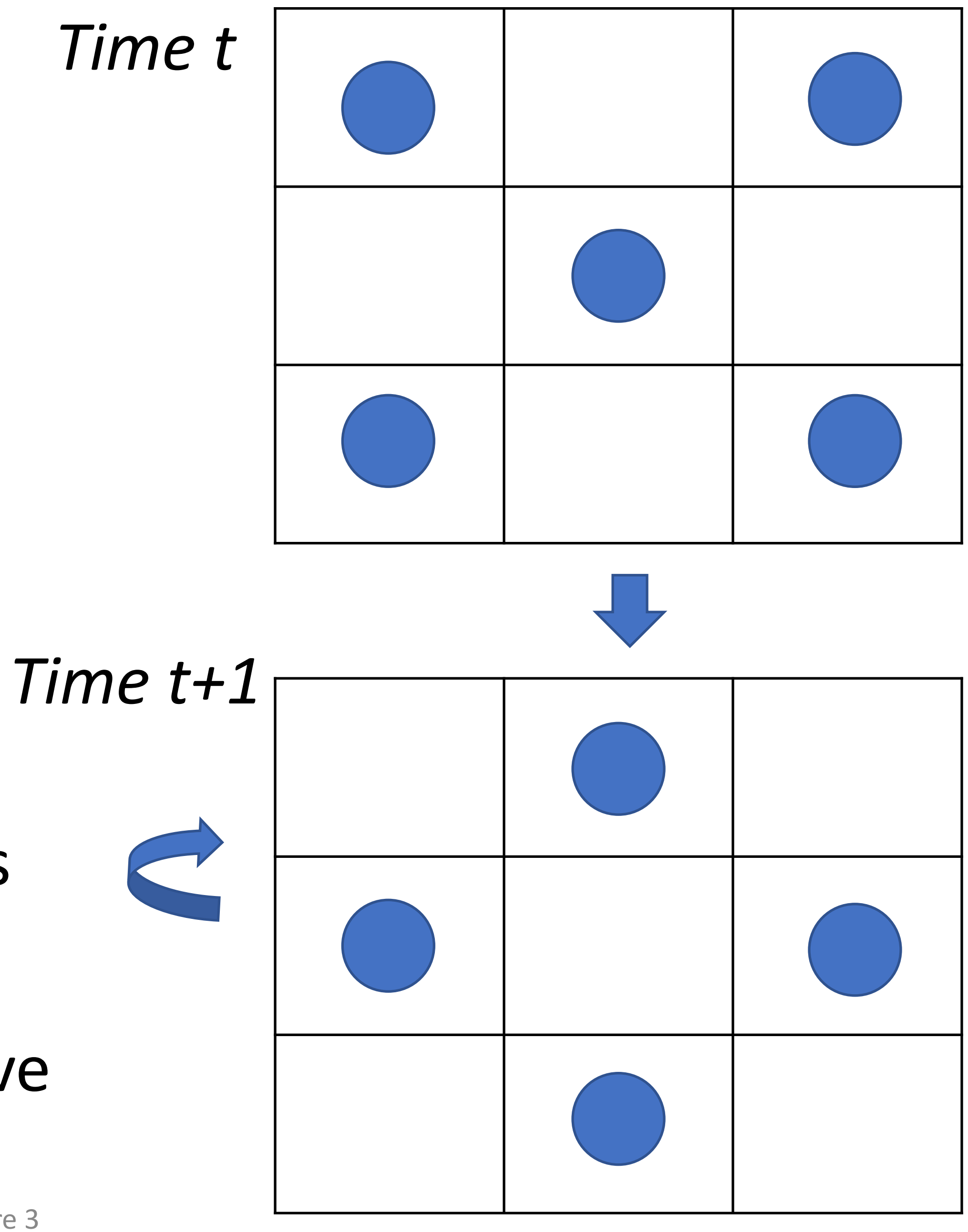
A Bigger Example: The Game of Life

- The Game of Life is played on 2D grid in time steps
- Grid cells are either *live* or *dead*
- A cell is live or dead at time *t+1* based on its neighbors at time *t*
  - Cells at the world's edge are always dead
- Defined by George Conway in 1969
  - An early example of cellular automata



Rules

- A live cell with < 2 neighbors dies
  - From loneliness
- A live cell with > 3 neighbors dies
  - From overcrowding
- A live cell with 2 or 3 neighbors survives
- A dead cell with 3 neighbors becomes live



The Game of Life

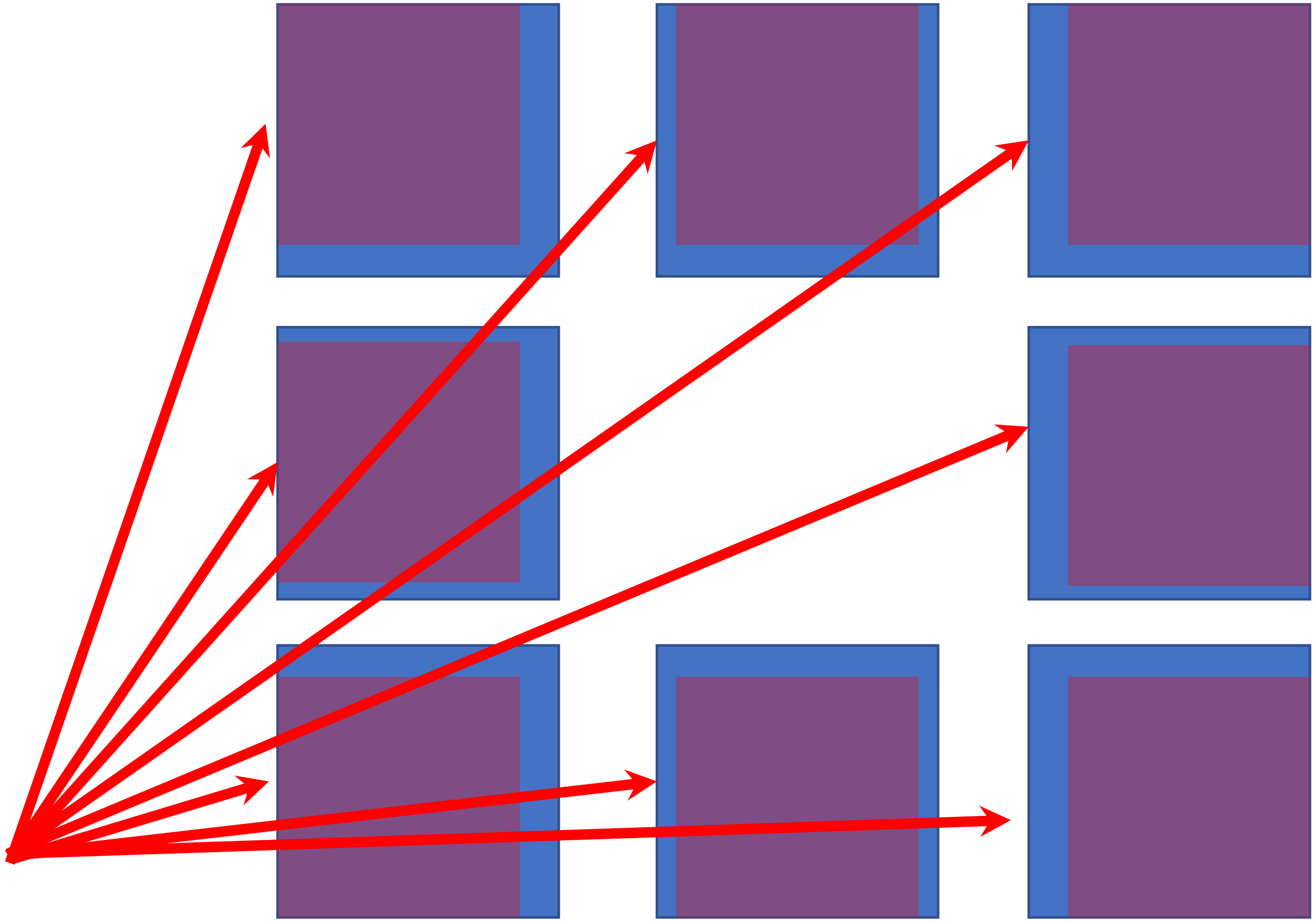
```
import numpy as np
Z = np.zeros((300, 600))
Z[1:-1,1:-1] = np.random.randint(0,2,np.shape(Z[1:-1,1:-1]))      # 0 is dead, 1 is live

while True:
   N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
        Z[1:-1, 0:-2]                 + Z[1:-1, 2:] +
        Z[2:  , 0:-2]  + Z[2:  , 1:-1]  + Z[2:  , 2:])
   birth = (N == 3) & (Z[1:-1, 1:-1] == 0)
   survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)
   Z[:,:] = 0
   Z[1:-1, 1:-1][birth | survive] = 1
```

Picture

N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
Z[1:-1, 0:-2] + Z[1:-1, 2:] +
Z[2: , 0:-2] + Z[2: , 1:-1] + Z[2: , 2:])

Summing these 8 subarrays computes the number of live neighbors for each cell in the interior of the space.



Explanation

```
…
    # N is a 2D array of the number of neighbors of each cell
    # birth is a 2D Boolean array; a cell is true if it is has 3 neighbors and is dead
    birth = (N == 3) & (Z[1:-1, 1:-1] == 0)

    # survive is a 2D Boolean array; a cell is true if it is has 2 or 3 neighbors and is live
    survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)

    # create a new generation
    # the interior cells of Z are live if they are born or survive the previous time step
    Z[:,:] = 0
    Z[1:-1, 1:-1][birth | survive] = 1
```

The Game of Life

```
import numpy as np
Z = np.zeros((300, 600))
Z[1:-1,1:-1] = np.random.randint(0,2,np.shape(Z[1:-1,1:-1]))      # 0 is dead, 1 is live

while True:
    N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
         Z[1:-1, 0:-2]                + Z[1:-1, 2:] +
         Z[2:   , 0:-2]  + Z[2:   , 1:-1]  + Z[2:   , 2:])
    birth = (N == 3) & (Z[1:-1, 1:-1] == 0)
    survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)
    Z[:,:] = 0
    Z[1:-1, 1:-1][birth | survive] = 1
```

CuNumeric

- CuNumeric is an implementation of the NumPy interface

- To use CuNumeric, replace

```
import numpy as np
by
import cunumeric as np
```

- Automatically runs NumPy programs in parallel
  - On multiple GPUs
  - Across large clusters

- Your homework
  - Write a cuNumeric image processing program
  - In general: Use bulk NumPy operators as much as possible – the more work in a single operation, the better!

- Next time: How cuNumeric works