CS 6068
Parallel Computing
Fall 2014
Lecture 3 – Sept 15

Prof. Fred Annexstein   @proffreda
fred.annexstein@uc.edu
Office Hours: 11-12 MW or by appointment
Tel: 513-556-1807
Meeting: Mondays 6:00-8:50PM Baldwin

Week 3 : Plan

- Review – Basic Parallel Operations

  Scatter, Gather, Reduce, Prefix-Scan, Histogram
- More Common Communication Patterns

  Compact/Filter
- **Segmented Scan with Apps**

  Sparse Matrix Products

  CSR format
- Parallel Sorting
- Sorting Networks – **Bitonic Sort**
- **Homework #4 Sorting Arrays for redeye removal**

Compact / Filter

- returns a sequence consisting of those items from the sequence for which *predicate*(*item*) is true. If *sequence* is a string or tuple, the result will be of the same type; otherwise, it is always a list. For example, to compute some small primes:
- >>> def prd(x): return x % 2 != 0 and x % 3 != 0
- >>> compact(prd, range(2, 20))
- [5, 7, 11, 13, 17, 19]

Relation of Scatter to Compact

- We assume that the filter predicate can be applied to each element in the list in parallel constant (fast) time, resulting in a bit vector of Boolean values.
- 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
  0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1
- Let's look at the running sum of these Boolean values.... 000112222334444556

Scatter Operations

- Recall that Scatter is an data communication operation that moves or permutes data based on an array of location addresses
- The index of each element in the filtered or compacted list is the running sum, or more precisely, the exclusive sum scan.
- For example, we see that 17 is in the 4th list position since its running sum value is 5 which is 1 greater than its left neighbor which is 4.

Quiz

- Complexity: What is the Work and Step Complexity of the Compact Operation on arrays?

- Let’s consider Irregular Workloads….
- Can you generalize Compact so that elements can allocate space on a per filtered element basis? For example, suppose each thread needs to dynamically allocate 0-10 memory locations

Review Complexity Analysis of Scan

Recursive Version

```
def prefix(add,x):
 if len(x)== 1: return x
 else:
    firsthalf = prefix(add, x[0:n/2])
    secondhalf = prefix(add, x[n/2+1:n] )
    secondhalf = map(add(x[n/2]), secondhalf)
    res = firsthalf + secondhalf
    return res
```

- Is this work efficient??

Recursive Odd/Evens

- Suppose we extracted the odds and even indexed elements from an array and ran scan in parallel on both. Are we near done??
- Improved Complexity?

-
```
def extract_even_indexed (x):
  result = range(len(x)/2)
  indx = range(0,len(x)-1,2)
  for i in indx:
        result[i/2]=x[i]
 return result
```

A Work-efficient Solution

```
def prefix(f,x):
  if len(x) == 1:
    return x
  else: #begin parallel
    e = extract_even_indexed(x)
    o = extract_odd_indexed(x) #end parallel
    s = map(f,e,o)
    r1 = [0] + prefix(f,s)
    r2 = map(f,e+[0],r1)
    return interleave(r2[0:len(r2)-1],r1[1:])
```

Segmented Scan

- Indication of segments within array
- Apply scan operation to segments independently
- Work an example using both inclusive and exclusive scans.
- Next: application to sparse matrices

Sparse Matrix Dense Vector Products

- *Compressed Sparse Row* CSR format
- Value = nonzero data one array
- Column= identifies column for each
- Rowptr= pointers to location of each 1st data in each row

$$\begin{bmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} =$$

VALUE $[a \; b \; c \; d \; e \; f]$

COLUMN $[0 \; 2 \; 0 \; 1 \; 2 \; 1]$

ROWPTR $[0 \; 2 \; 5]$

VECTOR

$$\begin{array}{cc} \begin{bmatrix} x \\ y \\ z \end{bmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \end{array}$$

1 CREATE SEGMENTED REP'N FROM VALUE + ROWPTR

2 GATHER VECTOR VALUES USING COLUMN

3 PAIRWISE MULTIPLY 1·2

$$[a \; b \mid c \; d \; e \mid f]$$

$$[x \; z \; x \; y \; z \; y]$$

1. CREATE SEGMENTED REP'N FROM VALUE + ROWPTR
2. GATHER VECTOR VALUES USING COLUMN
3. PAIRWISE MULTIPLY 1·2
   (BACKWARDS)
4. EXCLUSIVE SEGMENTED SUM SCAN

$$[\, a \;\; b \mid c \;\; d \;\; e \mid f \,]$$

$$[\, x \;\; z \;\; x \;\; y \;\; z \;\; y \,]$$

$$[\, \underbrace{a\cdot x \;\; b\cdot z}_{+\ \to\ \mathrm{out}(0)} \mid \underbrace{c\cdot x \;\; d\cdot y \;\; e\cdot z}_{+\ \to\ \mathrm{out}(1)} \mid \underbrace{f\cdot y}_{\mathrm{out}(2)} \,]$$

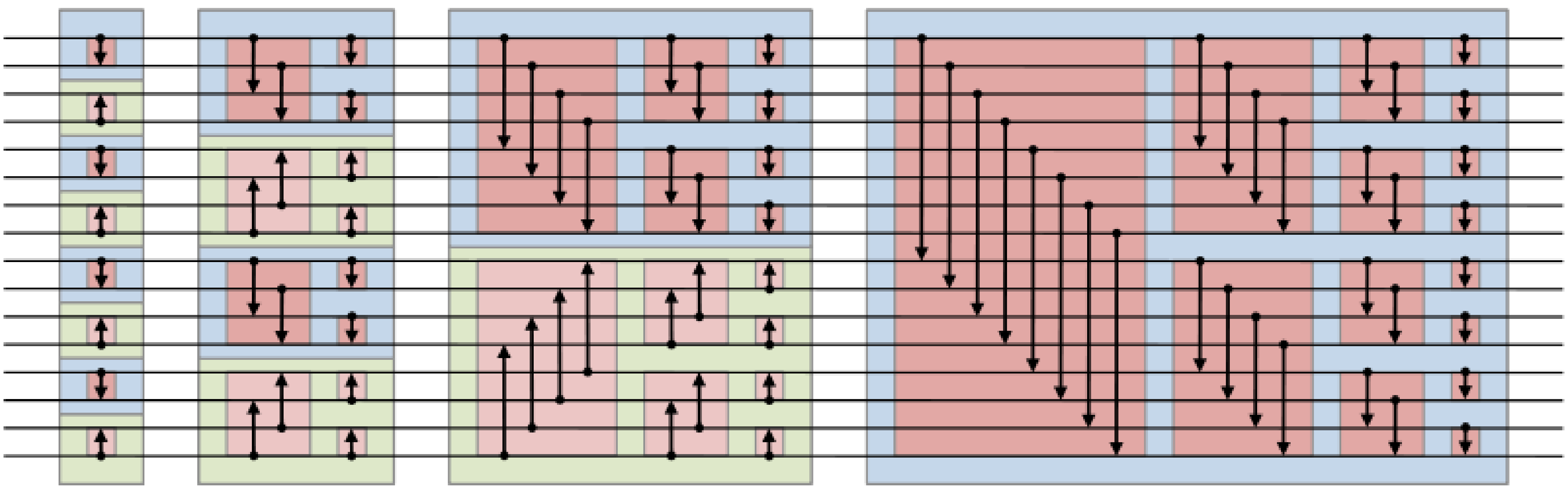
$$\begin{bmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + \cancel{0y} + bz \\ cx + dy + ez \\ \cancel{0x} + \cancel{0y} + fz \end{bmatrix}$$

Sorting

- Studied for classic patterns of data communication
- First look at favorite sequential algorithms and discuss parallelization
- BubbleSort
- QuickSort
- MergeSort
  - Three phase strategy

Sorting Networks

- Oblivious algorithms easily mapped to GPUs
- Bitonic Sorting
  - Ken Batcher (Kent State, Goodyear (Akron))
  - Bitonic Sequence (defined as a sequence of numbers with one direction change)



Recursive Bitonic Sorting

- def bitonic_sort(up, x):

```
if len(x) <= 1:
    return x
else:
   first = bitonic_sort(True, x[:len(x) / 2])
   second = bitonic_sort(False, x[len(x) / 2:])
   return bitonic_merge(up, first + second)
```

Bitonic Merge

- def bitonic_merge(up, x):

```
# assume input x is bitonic
# sorted list is returned
if len(x) == 1: return x
else:
  bitonic_compare(up, x)
  first = bitonic_merge(up, x[:len(x) / 2])
  second = bitonic_merge(up, x[len(x) / 2:])
  return first + second
```

- def bitonic_compare(up, x):

```
    dist = len(x) / 2
    for i in range(dist):
        if (x[i] > x[i + dist]) == up:
        x[i], x[i + dist] = x[i + dist], x[i]
```

Proof that Bitonic Sorting is correct

Assume given as input 0/1 sequence (applying Knuth's 0/1 Sorting Principle)

Assume that the length of the sequence is a power of 2

If the sequence is of length 1, do nothing

Otherwise, proceed as follows:

– Split the bitonic 0/1 sequence of length n into the first half and the second half i.e. 0000…01111…100000…0

– Perform $n/2$ compare interchange operations in parallel of the form $(i, i + n/2)$, $0 \cdot i < n/2$ (i.e., between corresponding items of the two halves)

– Claim: Either the first half is all 0's and the second half is bitonic, or the first half is bitonic and the second half is all 1's

– Therefore, it is sufficient to apply the same construction recursively on the two halves - Done!

Radix Sort – High Performance Sort

- Relies on numerical representation
- Example: Binary numbers
- Start with LSB move to MSB
- Each stage split into numbers into 2 sets
- Work Complexity
- Step Complexity
- Optimizations

HW#4 Remove Red Eye Effect

Stencil – normalized cross correlation scoring (done!)

Focus of HW4:
Sort all pixels using ncc scores

Map Operation: to remove red from highest scoring pixels (done!)