COMP 322: Fundamentals of Parallel Programming

Lecture 37: Algorithms based on Parallel Prefix (Scan) operations

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@rice.edu

http://comp322.rice.edu

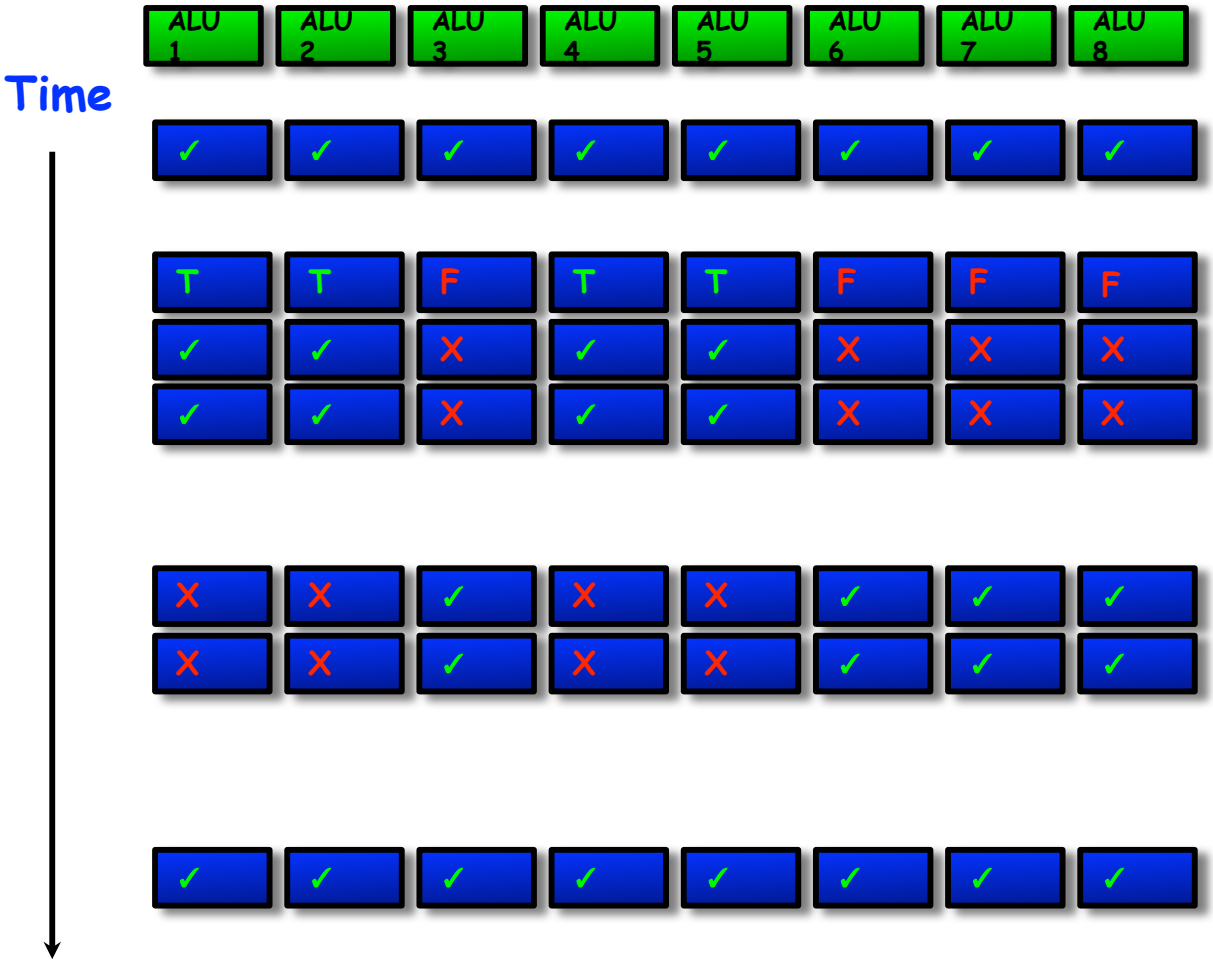
Worksheet #36: Branching in SIMD code

Consider SIMD execution of the following pseudocode with 8 threads. Assume that each call to doWork(x) takes x units of time, and ignore all other costs. How long will this program take when executed on 8 GPU cores, taking into consideration the branching issues discussed in Slide 9?

```
1. int tx = threadIdx.x; // ranges from 0 to 7
2. if (tx % 2 = 0) {
3.   S1: doWork(1); // Computation S1 takes 1 unit of time
4. }
5. else {
6.   S2: doWork(2); // Computation S2 takes 2 units of time
7. }
```

Solution: 3 units of time

GPU Design Idea #2: lock stepping w/ branching




```
Non branching code;

if(flag > 0){ /* branch */
  x = exp(y);
  y = 2.3*x;
}
else{
  x = sin(y);
  y = 2.1*x;
}

Non branching code;
```

The cheap branching approach means that some ALUs are idle as all ALUs traverse all branches [executing NOPs if necessary]

In the worst possible case we could see 1/8 of maximum performance.

Beyond Sum/Reduce Operations — Prefix Sum (Scan) Problem Statement

Given input array A, compute output array X as follows

$$X[i] = \sum_{0 \leq j \leq i} A[j]$$

- The above is an inclusive prefix sum since X[i] includes A[i]
- For an exclusive prefix sum, perform the summation for 0 <=j <i
- It is easy to see that inclusive prefix sums can be computed sequentially in O(n) time …

```
// Copy input array A into output array X

X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);

// Update array X with prefix sums

for (int i=1 ; i < X.length ; i++ ) X[i] += X[i-1];
```

- … and so can exclusive prefix sums

Summary of Parallel Prefix Sum Algorithm (Recap from Lecture 13)

- Critical path length, CPL = O(log n)
- Total number of add operations, WORK = O(n)
- Optimal algorithm for P = O(n/log n) processors
  - — Adding more processors does not help
- Parallel Prefix Sum has several applications that go beyond computing the sum of array elements
  - Parallel Prefix Sum can be used for any operation that is associative (need not be commutative)
    - — In contrast, finish accumulators required the operator to be both associative and commutative

Parallel Filter Operation (Recap)

[Credits: David Walker and Andrew W. Appel (Princeton), Dan Grossman (U. Washington)]

Given an array `input`, produce an array `output` containing only elements such that `f(elt)` is `true`, i.e., output = input.parallelStream.filter(f).toArray

Example: `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`

```
f: is elt > 10
output [17, 11, 13, 19, 24]
```

Parallelizable?

- —Finding elements for the output is easy
- —But getting them in the right place seems hard

Parallel prefix to the rescue (Recap)

1. Parallel map to compute a bit-vector for true elements (can use Java streams)

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector (not available in Java streams)

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output (can use Java streams)

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```


Examples of Problems that can be solved using Parallel Prefix Sum Operations

- Lexical comparisons of two strings of length O(n), to see which should appear first in a dictionary
- To implement radix sort
- To implement quicksort
- To perform lexical analysis. For example, to parse a program into tokens.
- To search for regular expressions. For example, to implement the UNIX grep program.
- . . .

Example Applications of Parallel Prefix Algorithm

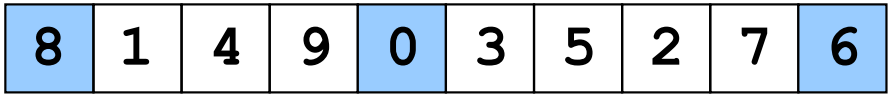
- **Prefix Max with Index of First Occurrence: given an input array A, output an array X of objects such that X[i].max is the maximum of elements A[0…i] and X[i].index contains the index of the first occurrence of X[i].max in A[0…i]**

- **Filter and Packing of Strings: given an input array A identify elements that satisfy some desired property (e.g., uppercase), and pack them in a new output array. (First create a 0/1 array for elements that satisfy the property, and then compute prefix sums to identify locations of elements to be packed.)**

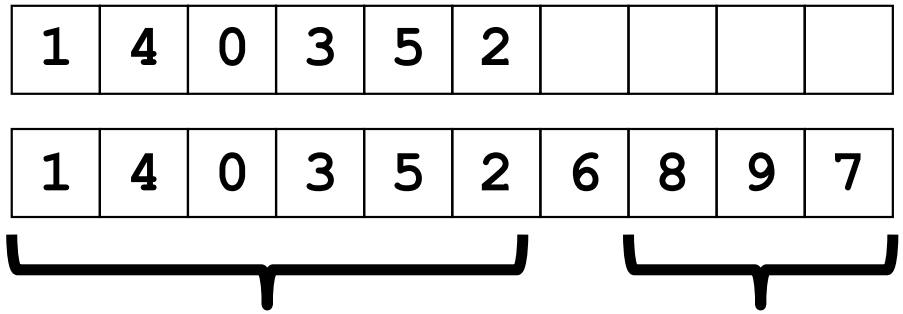
  —Useful for parallelizing partitioning step in Parallel Quicksort algorithm

Parallelizing Quicksort Example

- Step 1: pick pivot as median of three



- Steps 2: implement partition step as two filter/pack operations that store result in a second array



- Step 3: Two recursive sorts in parallel

Use of Prefix Sums to parallelize partition() in Quicksort

```
1. partition(int[] A, int M, int N) {
2.  pivot = … ; // choose pivot from M..N
3.  Allocate temporary buffer[] with size N-M+1 elements
4.  forall (point [k] : [0:N-M]) { // parallel loop
5.   lt[k] = (A[M+k] < A[pivot] ? 1 : 0);  // bit vector with < comparisons
6.   eq[k] = (A[M+k] == A[pivot] ? 1 : 0); // bit vector with = comparisons
7.   gt[k] = (A[M+k] > A[pivot] ? 1 : 0);  // bit vector with > comparisons
8.   buffer[k] = A[M+k];                   // Copy A[M..N] into buffer
9.  }
10.  // computePrefixSums() returns the prefix sum array and the total count of 1's in the input array
11. ltPs, ltCount = computePrefixSums(lt);
12. eqPs, eqCount = computePrefixSums(eq);
13. fgtPs, gtCount = computePrefixSums(gt);
14. // Parallel move from buffer into A
15. forall (point [k] : [0:N-M]) {
16.   if(lt[k]==1) A[M+ltPS[k]-1] = buffer[k];
17.   else if(eq[k]==1) A[M+ltCount+eqPS[k]-1] = buffer[k];
18.   else A[M+ltCount+eqCount+gtPS[k]-1] = buffer[k];
19.  }
20. }  // partition
```

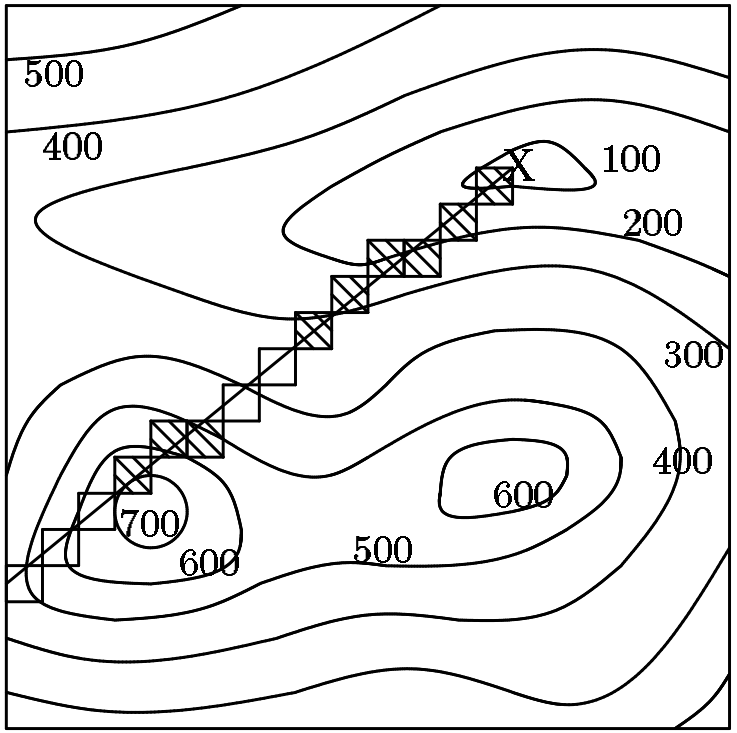

Formalizing Parallel Prefix: Scan and Pre-scan operations

- The *scan* operation is an inclusive parallel prefix sum operation.
- The *prescan* operation is an exclusive parallel prefix sum operation. It takes a binary associative operator $\oplus$ with identity I, and a vector of n elements, [a0, a1, ..., an−1], and returns the vector [I,a0,(a0 $\oplus$a1),...,(a0 $\oplus$a1 $\oplus$…$\oplus$an−2)].
- A prescan can be generated from a scan by shifting the vector right by one and inserting the identity. Similarly, the scan can be generated from the prescan by shifting left, and inserting at the end the sum of the last element of the prescan and the last element of the original vector.
- The scan operator was introduced in APL in the 1960's, and has been popularized recently in more modern languages, most notably the NESL project in CMU

Line-of-Sight Problem

- **Problem Statement: given a terrain map in the form of a grid of altitudes and an observation point, X, on the grid, find which points are visible along a ray originating at the observation point. Note that a point on a ray is visible if and only if no other point between it and the observation point has a greater vertical angle.**

  - **Define angle[i] = angle of point i on ray relative to observation point, X (can be computed from altitudes of X and i)**
  - **A max-prescan on angle[*] returns to each point the maximum previous angle.**
  - **Each point can compare its angle with its max-prescan value to determine if it will be visible or not**



Segmented Scan

- **Goal: Given a data vector and a flag vector as inputs, compute independent scans on segments of the data vector specified by the flag vector.**

$$x_i = \begin{cases} a_0 & i = 0 \\ \begin{cases} a_i & f_i = 1 \\ (x_{i-1} \oplus a_i) & f_i = 0 \end{cases} & 0 < i < n \end{cases}$$

a	=	[5	1	3	4	3	9	2	6]
f	=	[1	0	1	0	0	0	1	0]
segmented +-scan	=	[5	6	3	7	10	19	2	8]
segmented max-scan	=	[5	5	3	4	4	9	2	6]

Using Segmented Scan for Quicksort

```
procedure quicksort(keys)
  seg-flags[0] ← 1
  while not-sorted(keys)
    pivots    ← seg-copy(keys, seg-flags)
    f         ← pivots <=> keys
    keys      ← seg-split(keys, f, seg-flags)
    seg-flags ← new-seg-flags(keys, pivots, seg-flags)
```

Key	=	[6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4]
Seg-Flags	=	[1	0	0	0	0	0	0	0]
Pivots	=	[6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4]
F	=	[=	>	<	<	>	<	>	<]
Key ← split(Key, F)	=	[3.4	1.6	4.1	3.4	6.4	9.2	8.7	9.2]
Seg-Flags	=	[1	0	0	0	1	1	0	0]
Pivots	=	[3.4	3.4	3.4	3.4	6.4	9.2	9.2	9.2]
F	=	[=	<	>	=	=	=	<	=]
Key ← split(Key, F)	=	[1.6	3.4	3.4	4.1	6.4	8.7	9.2	9.2]
Seg-Flags	=	[1	1	0	1	1	1	1	0]