Distributed and Parallel Computing

Lecture 07

Alan P. Sexton

University of Birmingham

Spring 2018

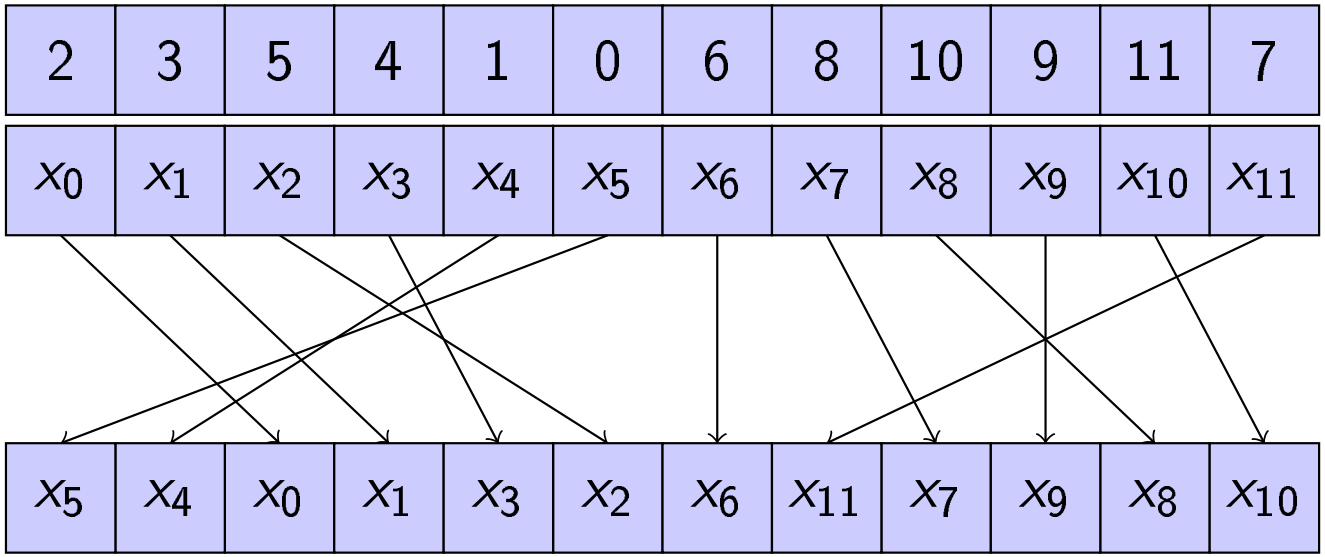
Basic Operations

Operations:

- `map(X, Y, op())`
  - Map memory access pattern
  - e.g. square every element of an array
  - e.g. vectorAdd
- `reduce(X, y, bin_assoc_op())`
  - Many-to-one memory pattern
  - e.g. sum the elements of an array
- `scan(X, Y, bin_assoc_op())`
  - Many-to-many memory pattern
  - e.g. calculate the cumulative sum of an array

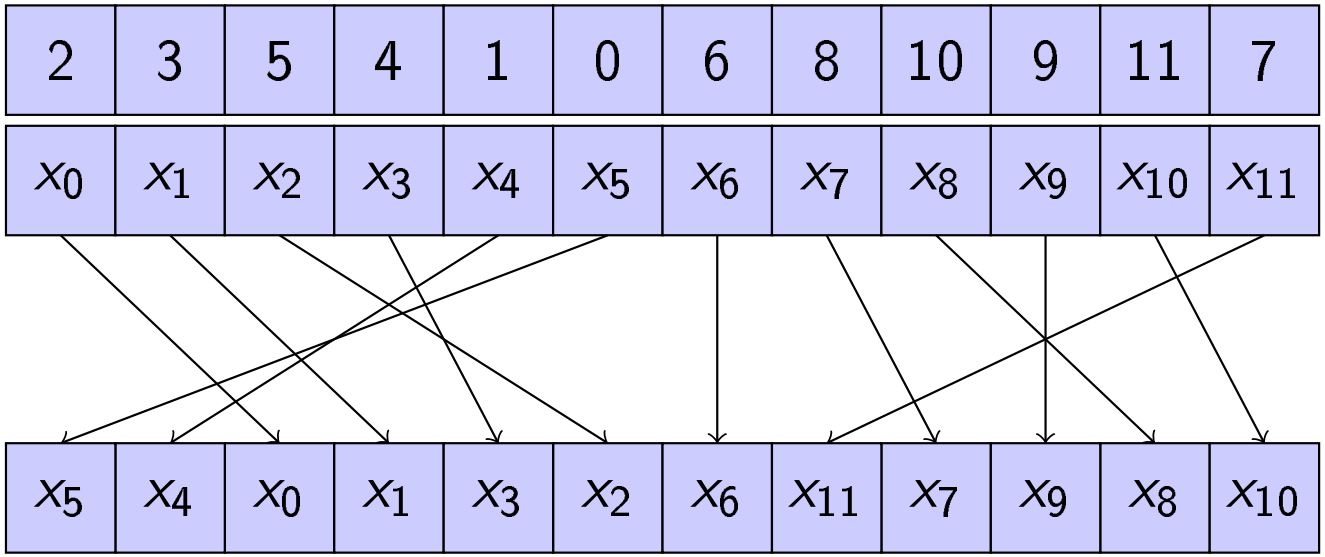
Scatter

The Scatter operation takes an input array I and a target address array T and sends each input element to the target address location in the output array: $O[T[i]] = I[i]$



Scatter

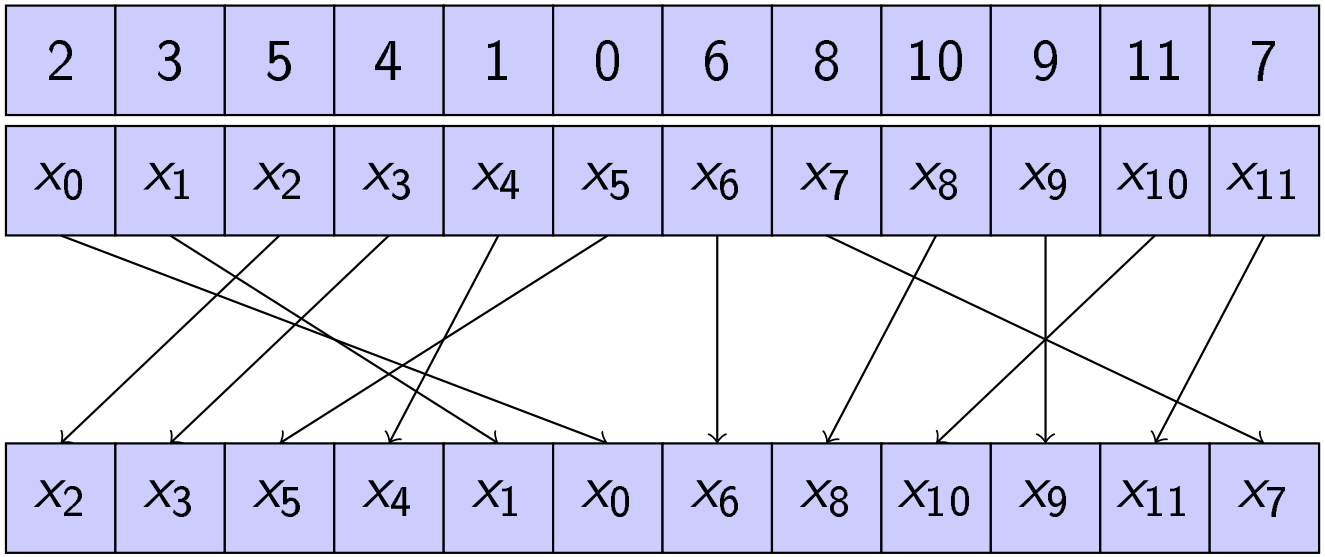
The Scatter operation takes an input array I and a target address array T and sends each input element to the target address location in the output array: $O[T[i]] = I[i]$



- There can be filtered variants where the assignment of an input element to the target is not carried out if either:
  - the target address is negative, or
  - a supplied extra predicate function applied to the input element evaluates to `TRUE`, or
  - a supplied extra array of the same size as the input array has 0 in the corresponding position

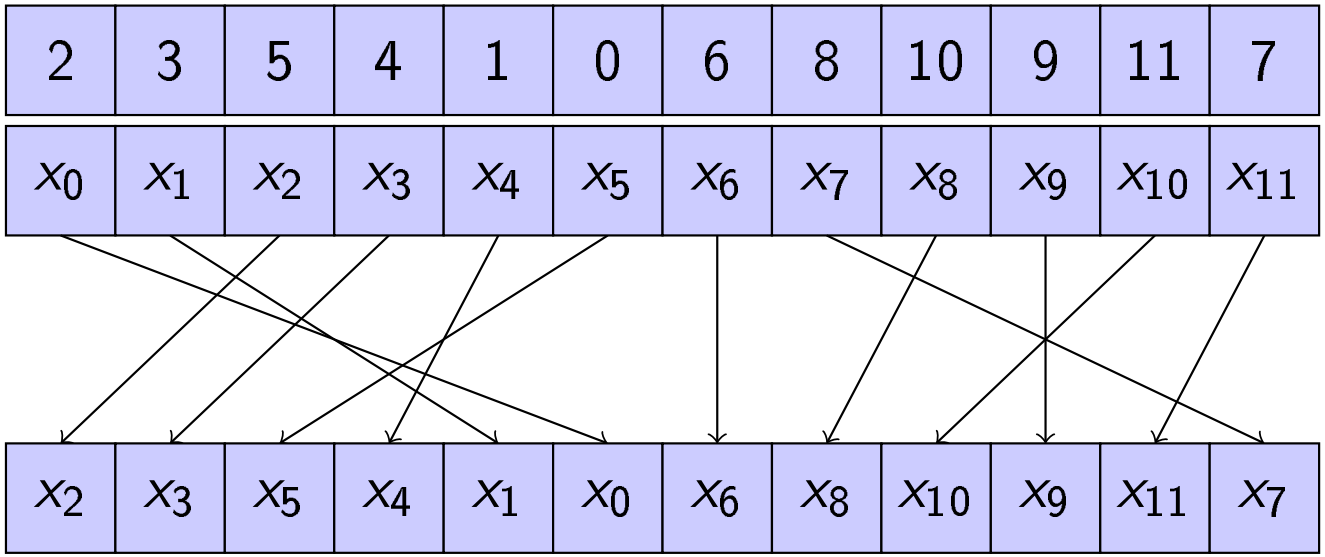
Gather

The Gather operation takes an input array I and a source address array S and writes to each output array (O) cell the value read from the location in the input array described by the corresponding element of the source address array: $O[i] = I[S[i]]$



Gather

The Gather operation takes an input array I and a source address array S and writes to each output array (O) cell the value read from the location in the input array described by the corresponding element of the source address array: $O[i] = I[S[i]]$



- As in the case of Scatter, there are similar filtered variants

Exclusive Scan

The scan we have studied is *inclusive*: it includes the current input element in its current cumulative result across the full input array. There is an exclusive variant:

Input	1	2	1	3	1	1	3	3	2	1	2	2
Inclusive Scan	1	3	4	7	8	9	12	15	17	18	20	22
Exclusive Scan	0	1	3	4	7	8	9	12	15	17	18	20

Exclusive Scan is a small modification to the scan algorithms (Hills-Steele-Horn and Blelloch) we have already studied. It does **NOT** just do an inclusive scan and shift the words on by one.

Segmented Scan

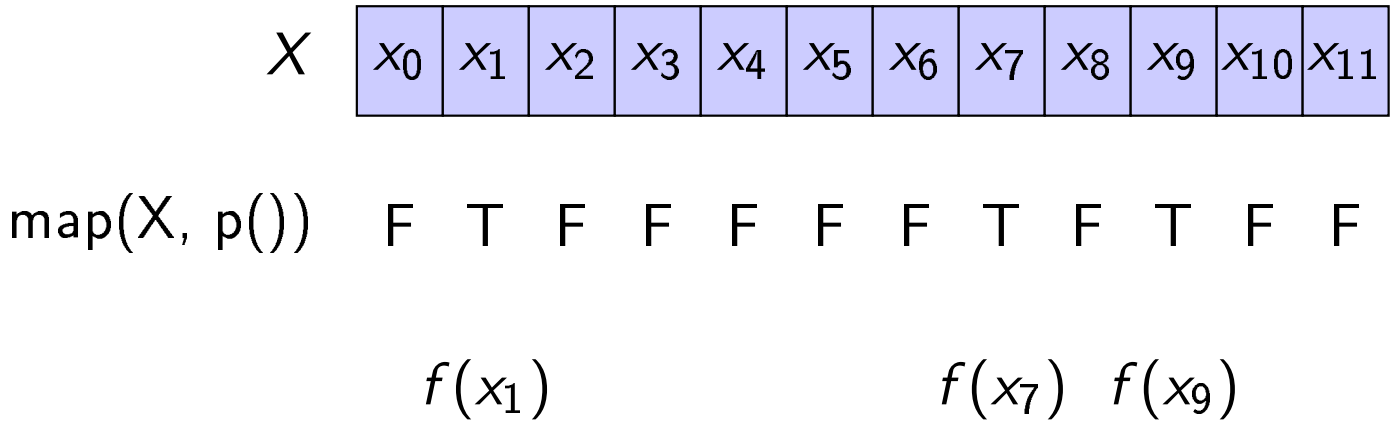
Often we want to do different scans on separate parts of the input array. It would be inefficient to do so by running multiple scan kernels one after the other. Instead we use a **segmented scan**:

Input	1	2	1	3	1	1	3	3	2	1	2	2
Headers	1	0	0	1	0	0	0	0	0	1	0	0
Alternative Headers	0	3	9									
Segmented Inclusive Scan	1	3	4	3	4	5	8	11	13	1	3	5
Segmented Exclusive Scan	0	1	3	0	3	4	5	8	11	0	1	3

This is again a small modification to our previous algorithms and, in spite of the extra work of dealing with the header array, has the same step and work complexity, though is a bit slower.

Filtering an Array: 1

Let's say we want to apply a function, $f()$, to all elements of an array that satisfy predicate $p()$:

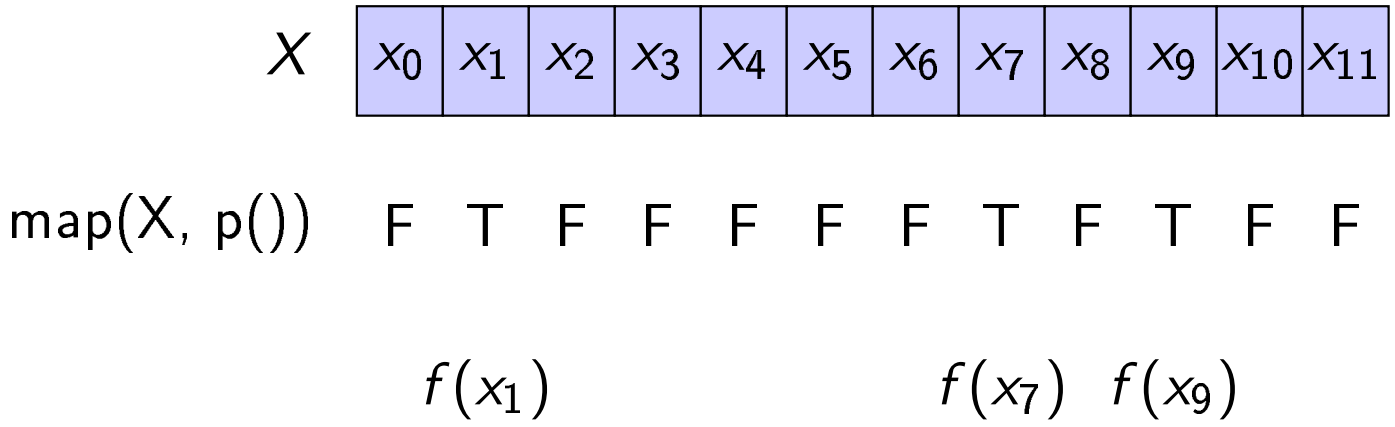


- We could do it by having each thread execute the following:

```
if (p(X(i))
    f(X(i));
```

Filtering an Array: 1

Let's say we want to apply a function, $f()$, to all elements of an array that satisfy predicate $p()$:



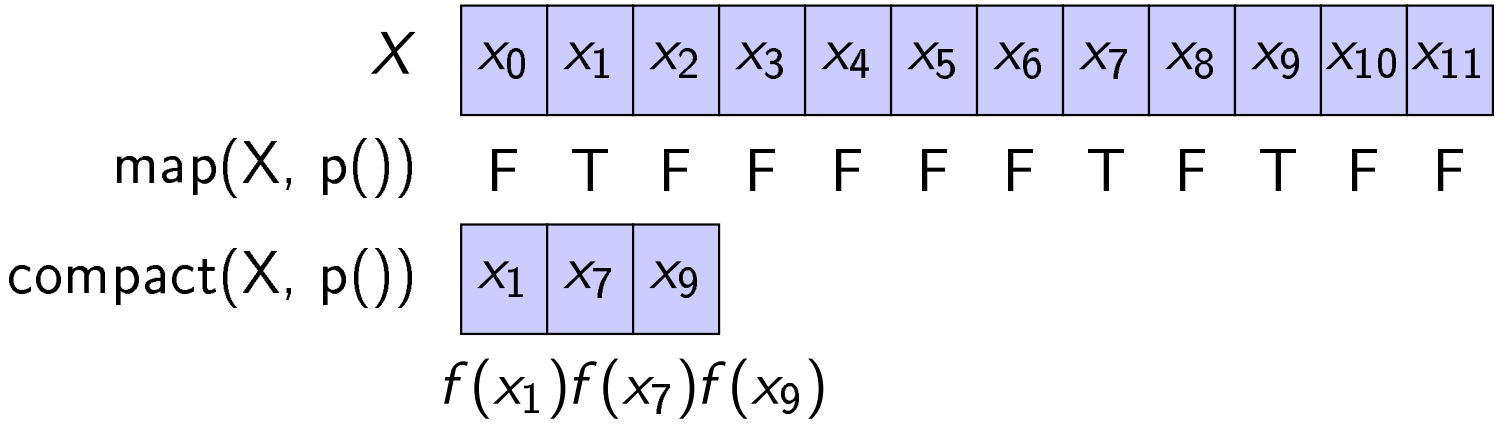
- We could do it by having each thread execute the following:

```
if (p(X(i))
    f(X(i));
```

- But: all threads execute $p()$, only some threads execute $f()$, and those threads are not compacted, so lots of divergence
  - e.g. Array of 1,000,000 entries, 1 in 20 satisfy the predicate, so 19 in 20 execute for the full length of $f()$ doing nothing while the remaining threads in the warp do something useful

Filtering an Array: 2

An alternative approach:

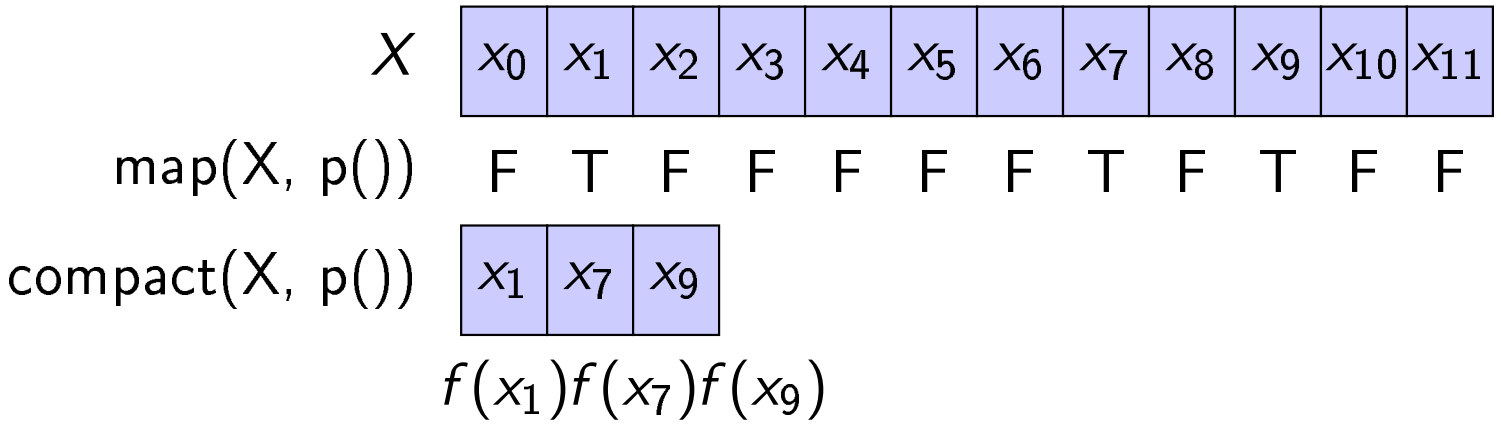


- The code for this would look like (each line in a different kernel)

```
compact(X, Y, p());
map(Y, f());
```

Filtering an Array: 2

An alternative approach:

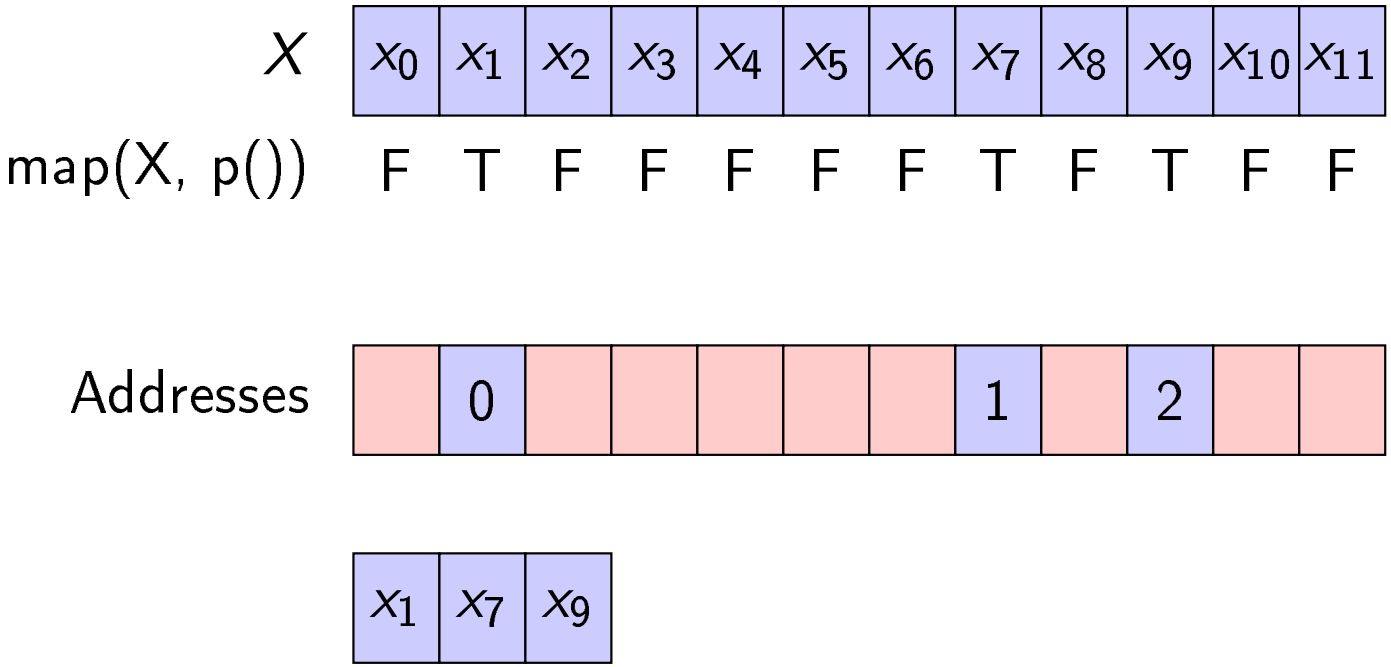


- The code for this would look like (each line in a different kernel)

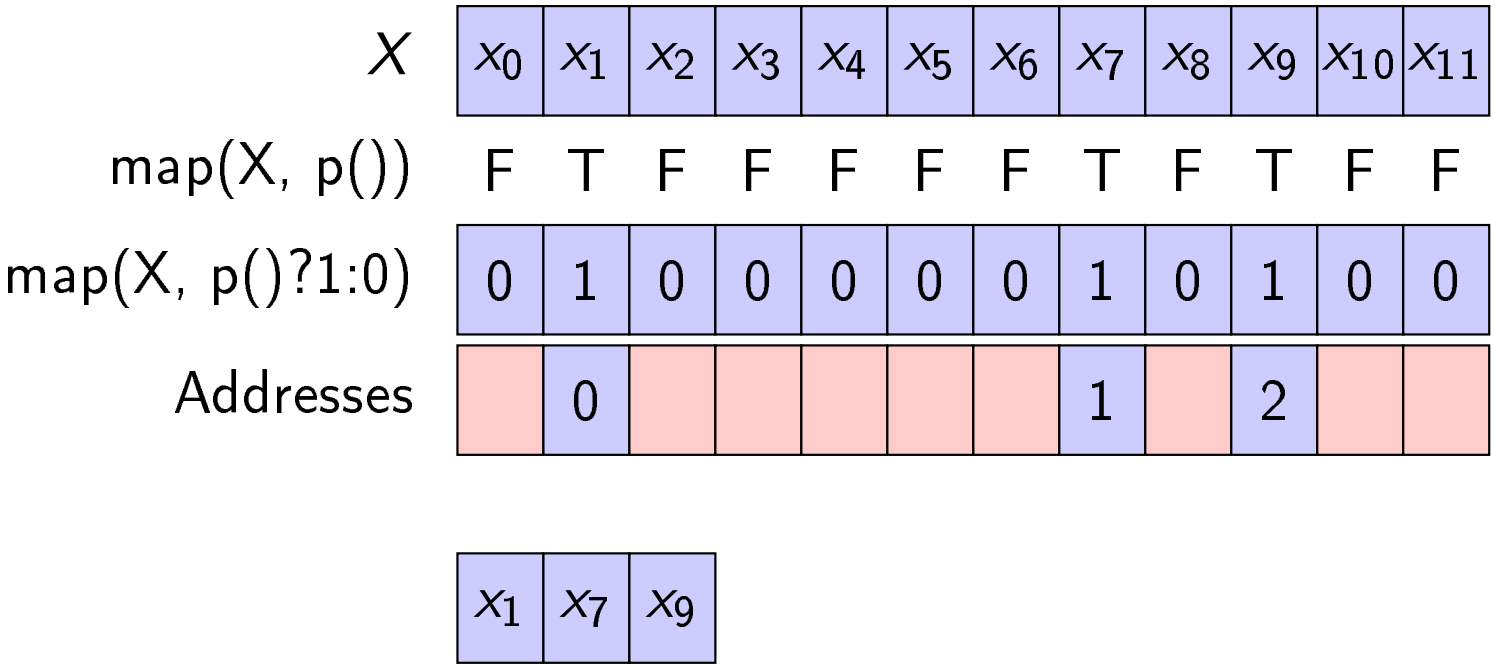
```
compact(X, Y, p());
map(Y, f());
```

- all threads are used to execute the first line, but only as many threads as the length of the filtered array execute $f()$, and those threads are compact, so no divergence
  - e.g. Array of 1,000,000 entries, 1,000,000 threads executing the compact, 1 in 20 satisfy the predicate, so only 50,000 threads execute $f()$

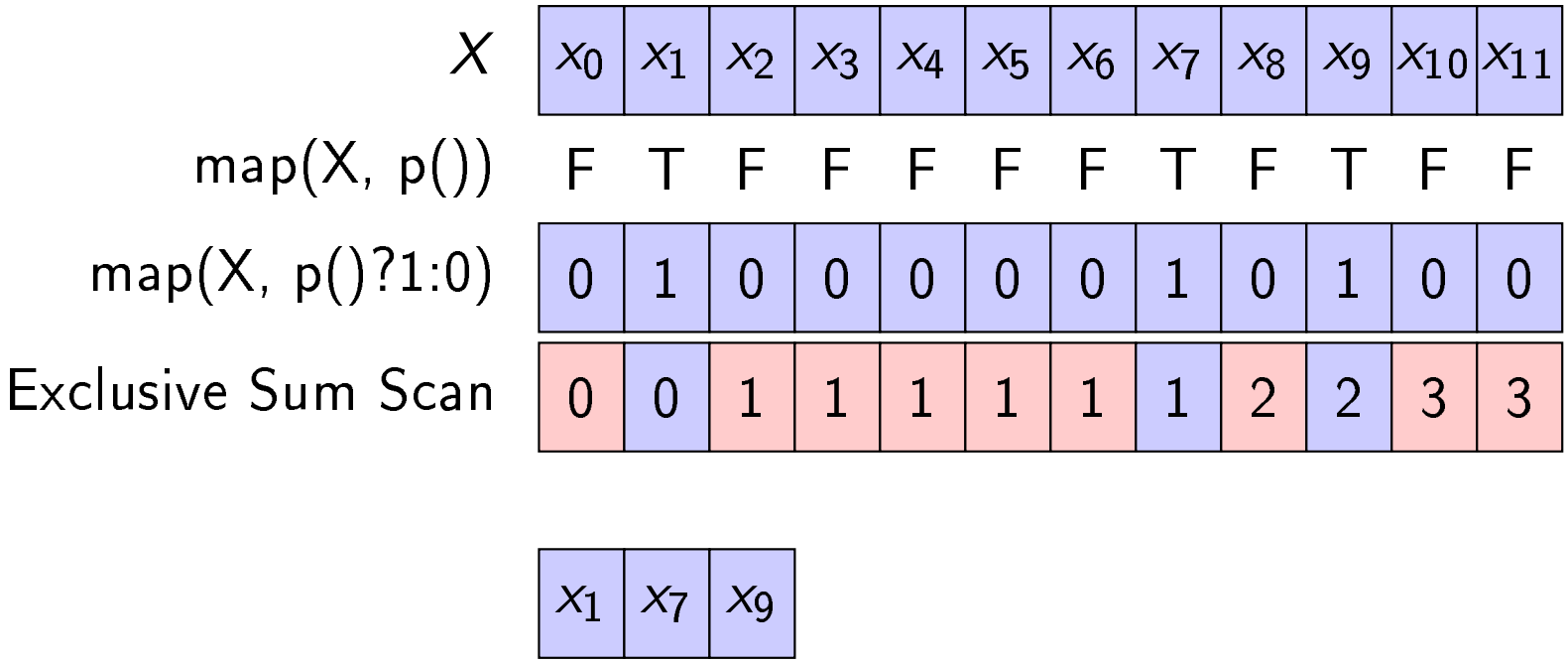
Compact


X
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
map(X, p())
F T F F F F F T F T F F
Addresses
0 1 2
x_1 x_7 x_9

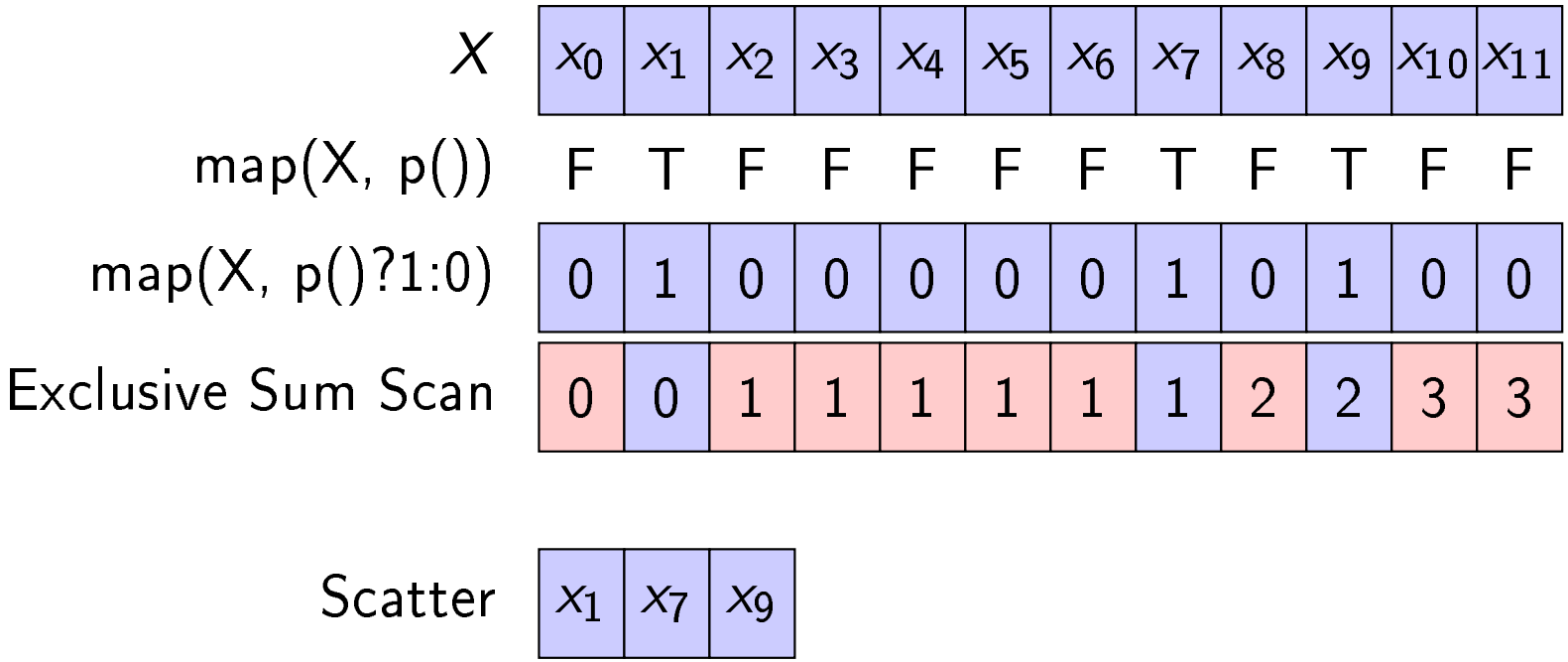
Compact


X
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11
map(X, p())
F T F F F F F T F T F F
map(X, p()?1:0)
0 1 0 0 0 0 0 1 0 1 0 0
Addresses
0 1 2
x1 x7 x9

Compact

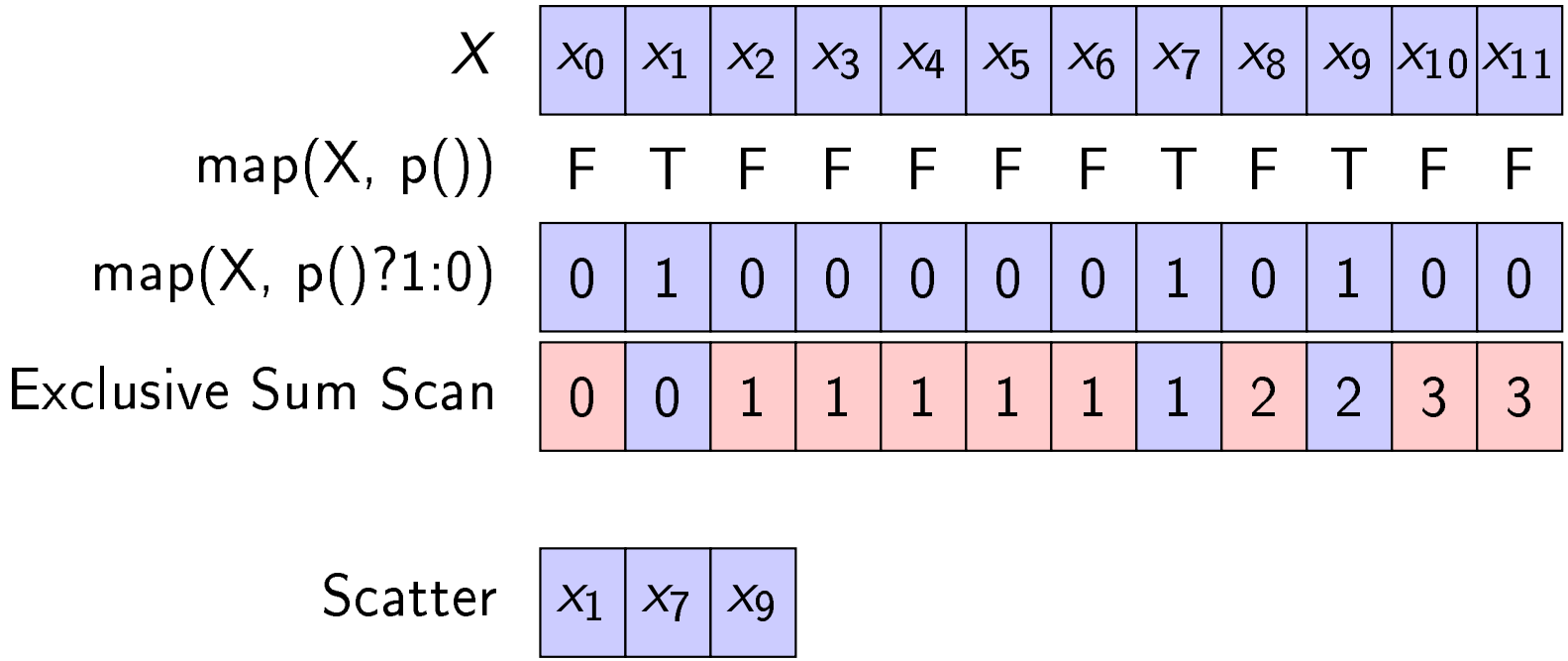

X
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11
map(X, p())
F T F F F F F T F T F F
map(X, p()?1:0)
0 1 0 0 0 0 0 1 0 1 0 0
Exclusive Sum Scan
0 0 1 1 1 1 1 1 2 2 3 3
x1 x7 x9

Compact


X
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
map(X, p())
F T F F F F F T F T F F
map(X, p()?1:0)
0 1 0 0 0 0 0 1 0 1 0 0
Exclusive Sum Scan
0 0 1 1 1 1 1 1 2 2 3 3

Scatter
x_1 x_7 x_9

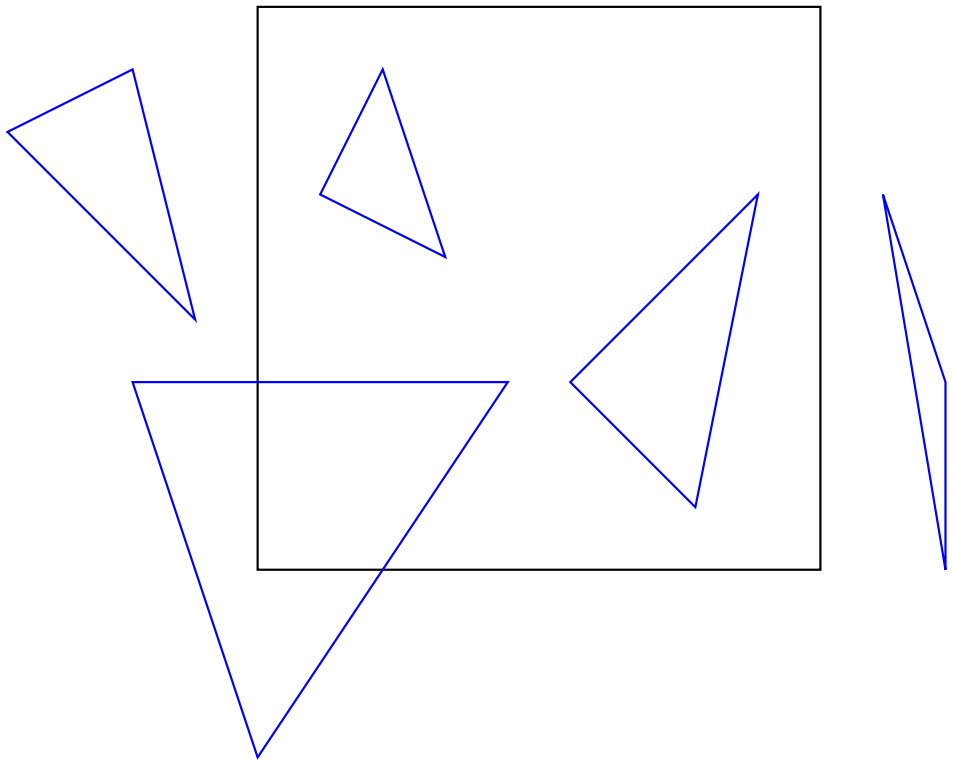
Compact



1. Map Predicate (1 for True, 0 for False) of Input into P
2. Exclusive Sum Scan of P into S
3. Scatter input into output using addresses from S if corresponding P value is greater than 0

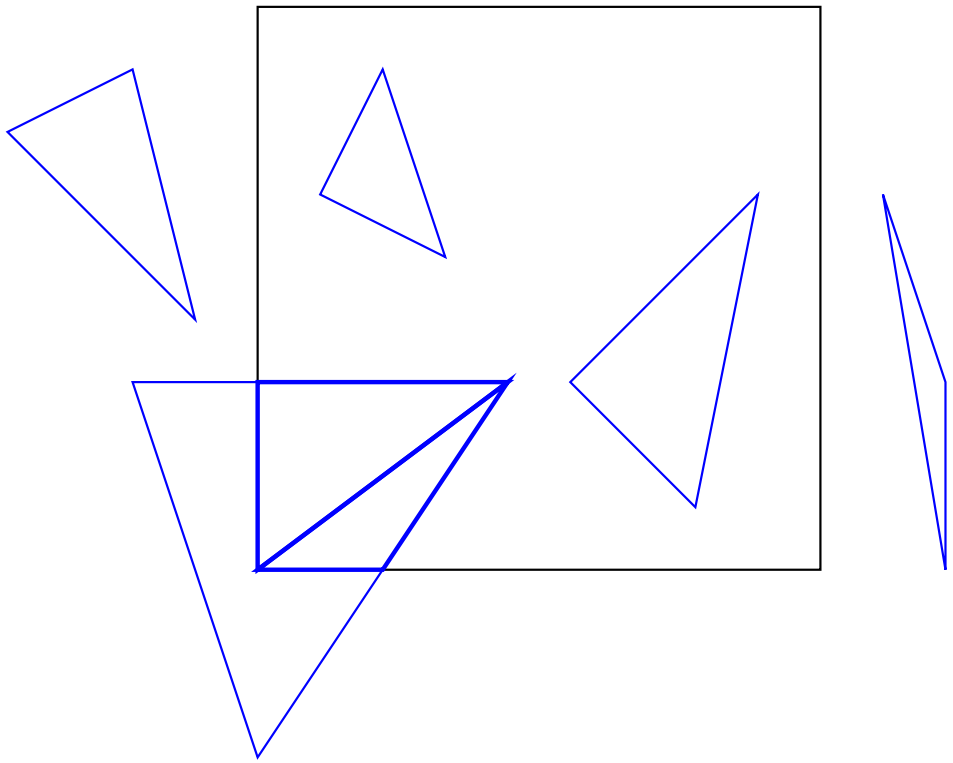
Generalised Compact

We can generalize compact: Compact reserves 1 output cell for each input cell that matches the predicate. Sometimes you may want more than one. Consider clipping triangles to a view port:



Generalised Compact

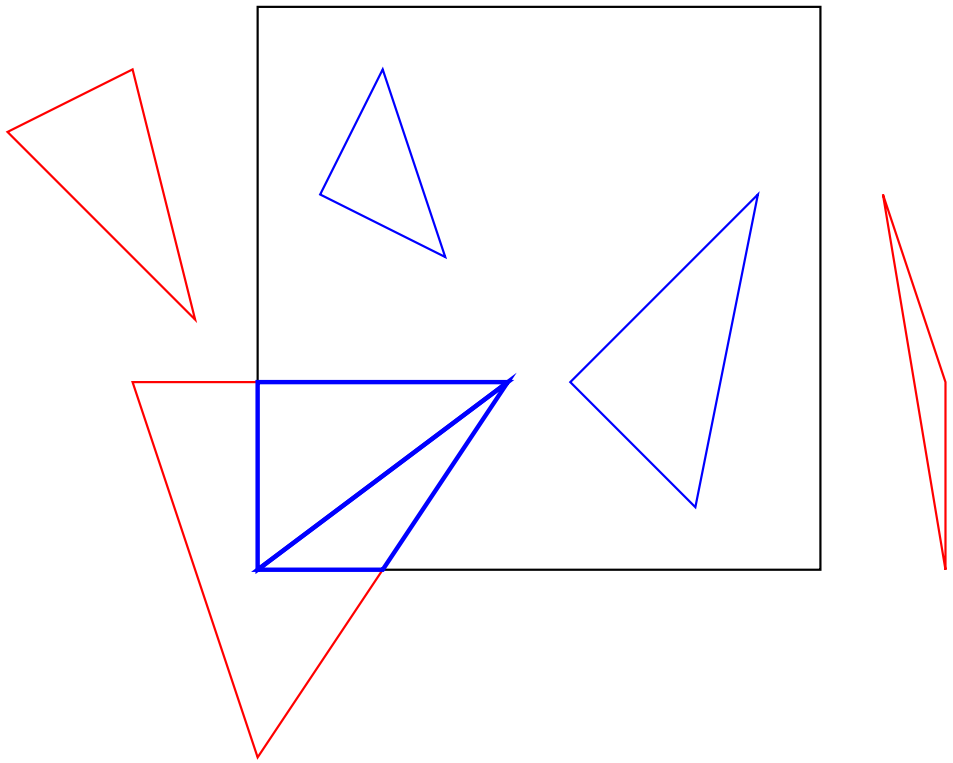
We can generalize compact: Compact reserves 1 output cell for each input cell that matches the predicate. Sometimes you may want more than one. Consider clipping triangles to a view port:



- Can have up to 5 triangles from clipping a single triangle

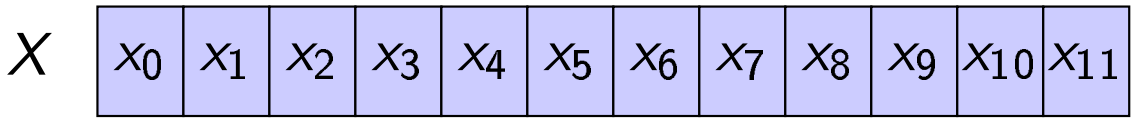
Generalised Compact

We can generalize compact: Compact reserves 1 output cell for each input cell that matches the predicate. Sometimes you may want more than one. Consider clipping triangles to a view port:



- Can have up to 5 triangles from clipping a single triangle
- This is a generalised compact operation

Generalised Compact



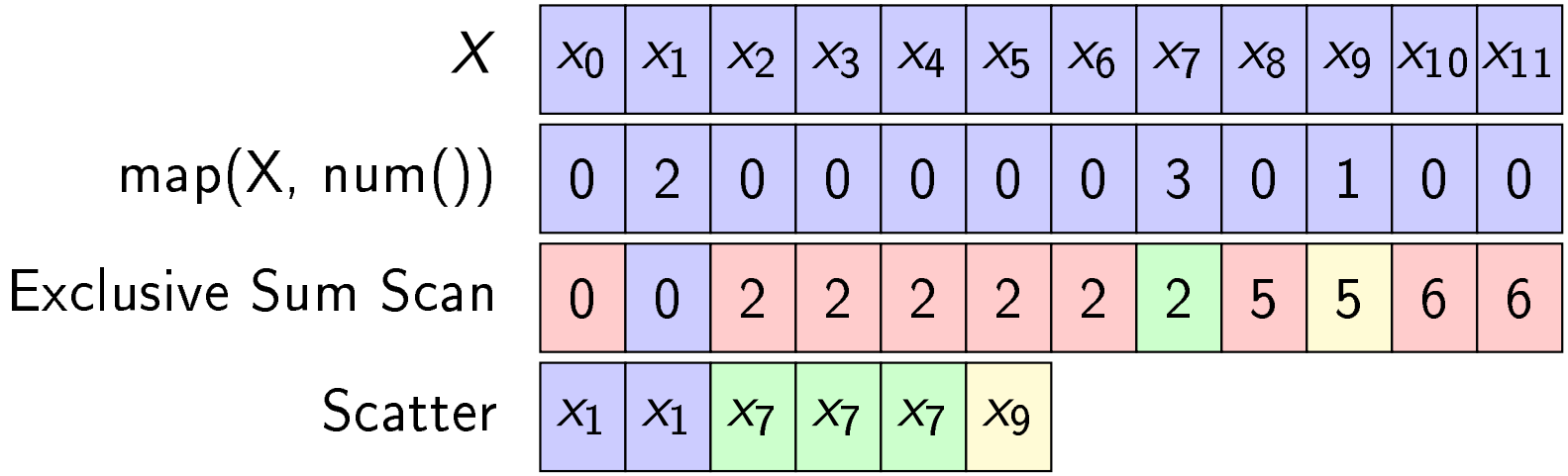
Generalised Compact

X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
map(X, num())	0	2	0	0	0	0	0	3	0	1	0	0

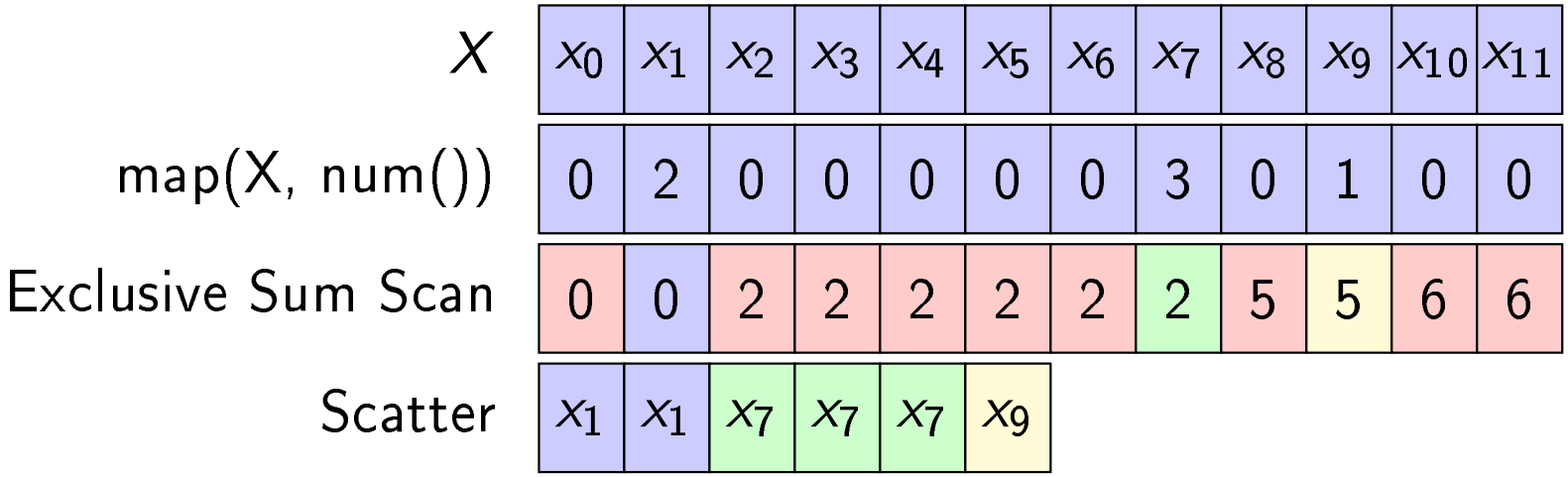
Generalised Compact

X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
map(X, num())	0	2	0	0	0	0	0	3	0	1	0	0
Exclusive Sum Scan	0	0	2	2	2	2	2	2	5	5	6	6

Generalised Compact



Generalised Compact



- The same Exclusive Sum Scan and Scatter as before, but now
  - the num() function returns the number of output cells needed for each input element and
  - each thread i now generates all num(i) output elements corresponding to a single input element x_i

Application: Sparse Matrix Dense Vector Mult (SpMv)

Recall matrix vector multiplication:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Sparse matrices have many zeros

- Occur *VERY* frequently
- Traditional 2-dimensional array represention:
  - Very wasteful of space
  - Lots of processing power wasted on multiplications by zero

Representation: Compressed Sparse Row

$$\begin{bmatrix} 0 & b & c \\ d & e & f \\ 0 & 0 & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} by + cz \\ dx + ey + fz \\ iz \end{bmatrix}$$

- V Value array: non-zero values in T to B, L to R order:

$$\begin{bmatrix} b & c & d & e & f & i \end{bmatrix}$$

- C Column array: the column of the corresponding value:

$$\begin{bmatrix} 1 & 2 & 0 & 1 & 2 & 2 \end{bmatrix}$$

- R Row pointer array: the index in V of each element that starts a row:

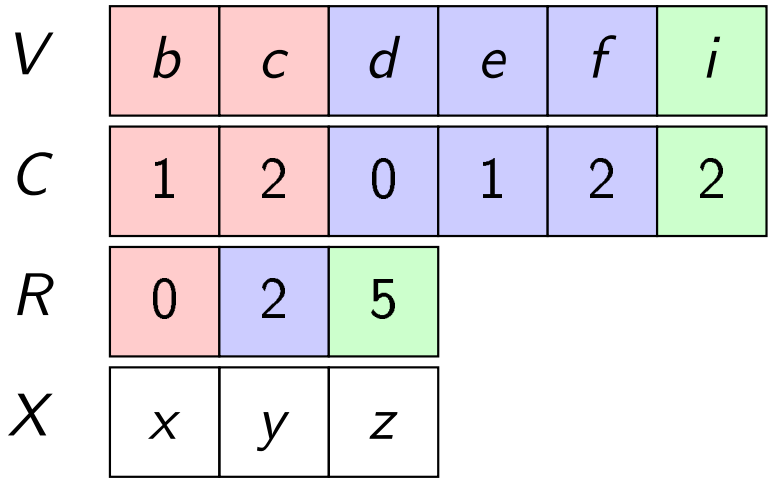
$$\begin{bmatrix} 0 & 2 & 5 \end{bmatrix}$$

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

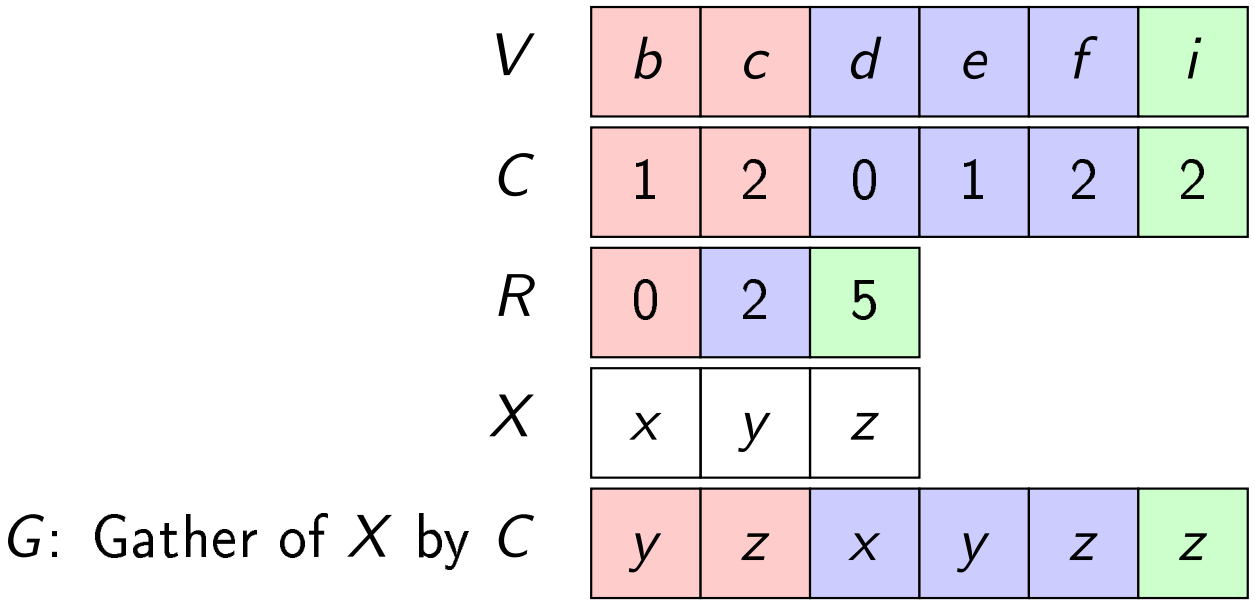
```
V: [b c | d e f | i]
C: [1 2 | 0 1 2 | 2]
R: [0 2 5]
```

- Note that the segments in V and C indicated, are defined by the header indices in R
- A gather using C to choose elements from the vector, will line up the elements of the vector with the correct elements of V that they have to be multiplied with
- The multiplication can be carried out with a map
- Now the segments of the results can be added with a segmented inclusive sum scan

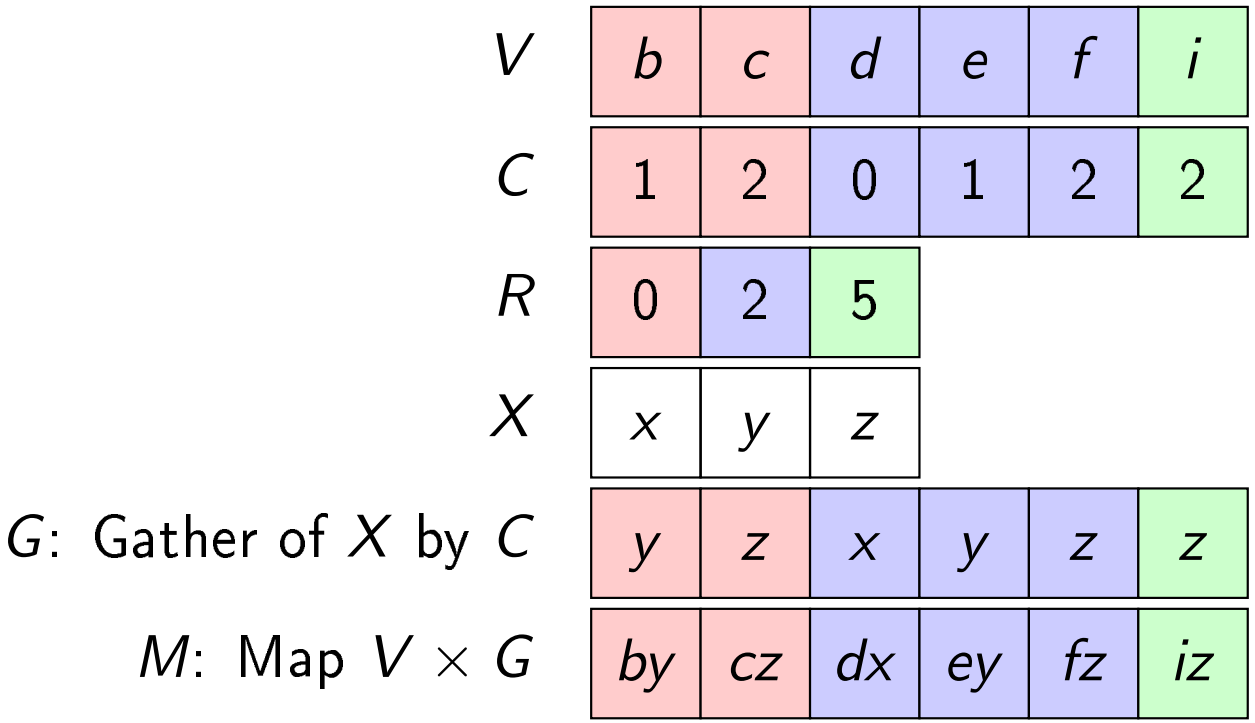
SpMv Execution



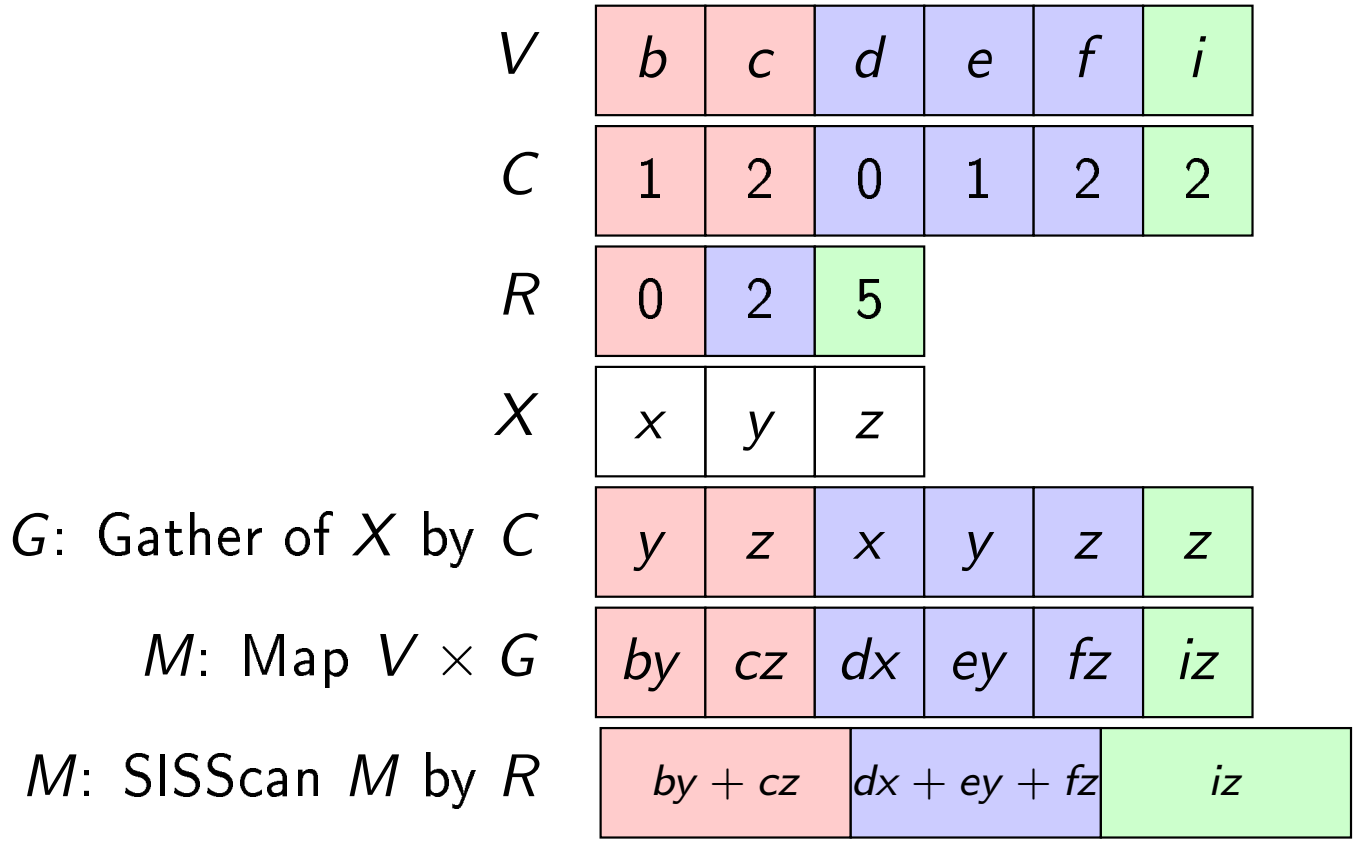
SpMv Execution



SpMv Execution

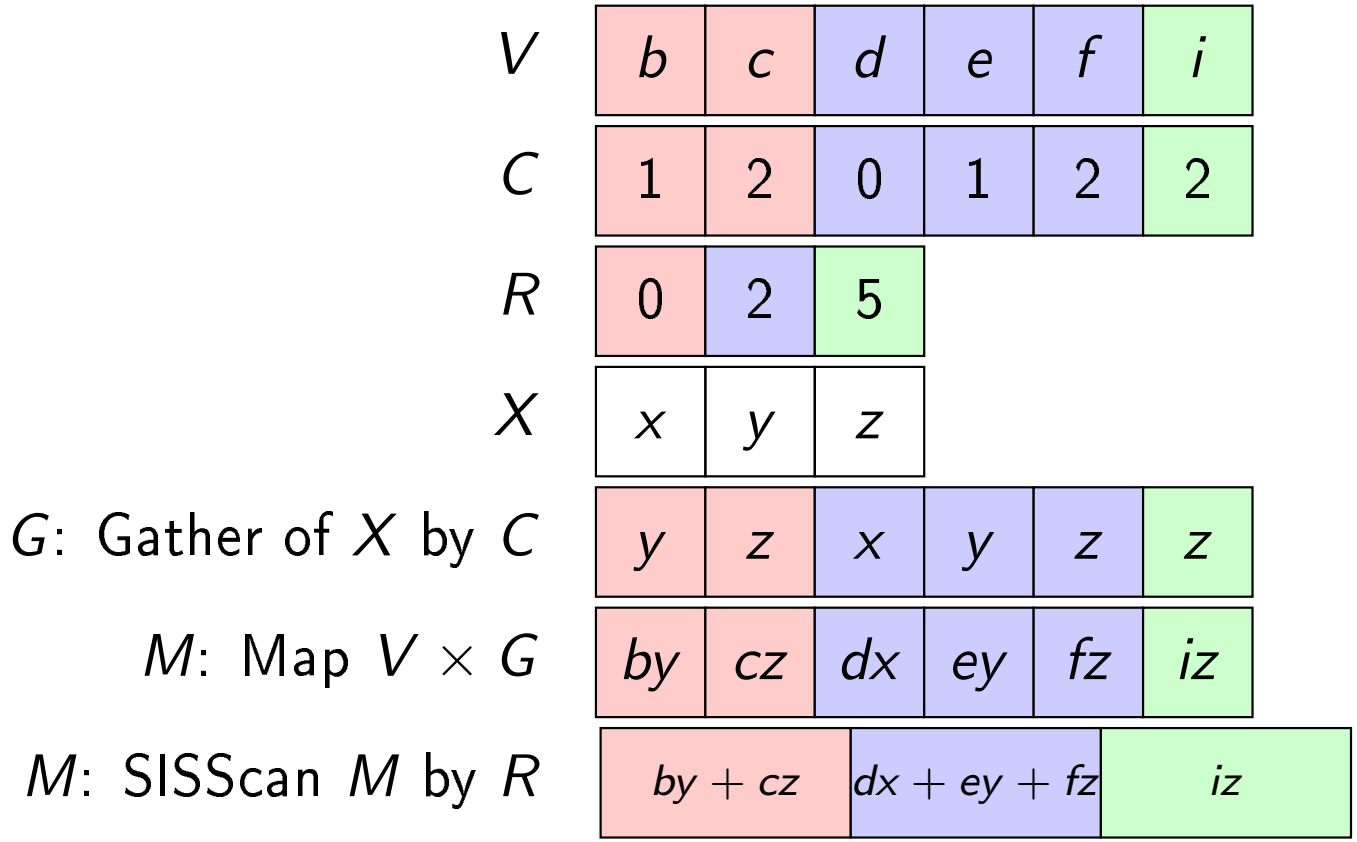


SpMv Execution



- Where SISScan is the Segmented Inclusive Sum Scan

SpMv Execution



- Where SISScan is the Segmented Inclusive Sum Scan
  - No zeros: lots of space saved
  - No zero multiplications: lots of processing saved