Emphasis on Regions, Dimensions

Much of ZPL programming involves paying attention to regions, the arrays defined from them, and how their dimensions match other region/array dimensions.

Index1 ...

- ZPL comes with “constant arrays” of any size
- Index*i* means indices of the ith dimension

```
[1..n,1..n]  begin
               Z := Index1; -- fill with first index
               P := Index2; -- fill with second index
               L := Z=P;    -- define identity array
             end;
```

- Index*i* arrays: compiler created using no space

```
1  1  1  1     1  2  3  4     1  0  0  0
2  2  2  2     1  2  3  4     0  1  0  0
3  3  3  3     1  2  3  4     0  0  1  0
4  4  4  4     1  2  3  4     0  0  0  1
  Index1         Index2            L
```

Scalars and Arrays

- Scalars have no dimensionality, and are replicated across the processors
- Arrays have dimensionality and are partitioned across the processors
- ZPL's "lower" dimension arrays are expressed with singleton dimensions
  - [1..n, 1] is a 2D array (but a single column) that is allocated with the first column or arrays
  - [n,1..n] is a 2D array (but a single row) allocated with the last row of arrays
  - This allocation property is important; more later

Partial Reductions

- Partial reductions reduce dimensions without reducing to a scalar, e.g. adding up rows
- Partial reductions require two regions, one on the operator and one on the statement

  Let A ⇔ [1..n,1..n], Col1 ⇔ [1..n,1] Rown ⇔[n.1..n]

  ```
  [1..n,1] Col1   := +<<[1..n,1..n] A;  -- Add up rows
  [n,1..n] Rown := max<<[1..n,1..n] A; -- Max down cols
  ```
- The compiler compares the two regions and figures out which one(s) to reduce

Flood

Flood (>>) is the inverse of reduce: it replicates data from lower dimensions to higher

- Like reduce it takes two regions, one on the operator and one on the statement
  - [1..m,1..n] A := >>[1..m,k] B; -- Replicate B's kth column
- The replication uses broadcast, often an efficient operation
- Matrix vector operations…flood vector to match shape: A [1..m,1..n] MaxC [1..m,1]:
  - [1..m,1] MaxC := max<<[1..m,1..n] A; --Find max of each row
  - [1..m,1..n]   A := A / >>[1..m,1] MaxC;--Scale each row by max

Closer Look At Scaling Each Row

[1..m,1] MaxC := max<<[1..m,1..n] A; --Find max of each row

[1..m,1..n]   A := A / >>[1..m,1] MaxC;--Scale each row by max

- Flooding distributes values (efficiently) so that the computation is element-wise … lowers communication

2	4	4	2	4	4	4	4	4
0	2	3	6	6	6	6	3	6
3	3	3	3	3	3	3	3	3
8	2	4	0	8	8	8	8	8
A				MaxC	>>[1..m,1] MaxC			

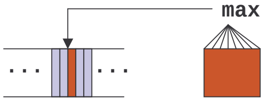
The purpose of keeping MaxC a 2D array is control how it is allocated

Flood Regions and Arrays

Flood dimensions recognize that specifying a particular column *over specifies* the situation

Need a *generic* column -- or a column that does not have a specific position … use ‘*’ as value

```
region   FlCol = [1..m, *];      -- Flood regions
         FlRow = [*, 1..n];
var       MaxC : [FlCol] double; --An m length col
           Row : [FlRow] double; -- An n length row
[1..m,*] MaxC := max<< [1..m,1..n] A; -- Better
```



Think of column in every position

Flood arrays (continued)

Since flood arrays have some unspecified dimensions, they can be “promoted” in those dimensions, i.e logically replicated

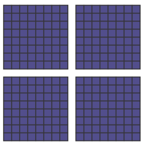
- Scaling a value by max of row w/o flooding:

```
   [1..m,*]   MaxC := max<< [1..m,1..n] A;
[1..m,1..n]      A := A / MaxC;     --Scale A;
```

The promotion of flooded arrays is only logical

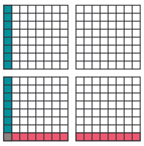
Key Difference Between Flood & Singleton

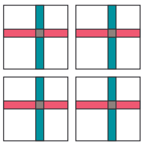
- Lower dimensional arrays can specify a singleton or a flood … it affects allocation



Region [1..n,1..n] allocated to 4 processors

Regions [1..n,1] and [n,1..n] allocated to 4 processors





Regions [1..n,*] and [*,1..n] allocated to 4 processors

Scan

- Scan is the parallel prefix operation for associative operators: `+, *, min, max, &, |`
- Scan is like reduction, but uses **||**
- Exclusive prefix sum is **+||**

```
  A ⇔  2  4   6   8   3
+||A ⇔  0  2   6  12  20
```

  - First position gets identity, and each position gets preceding elements combined
- Scan has two forms
  - Inclusive starts with first element
  - Exclusive starts with the identity … In ≡ Ex *op* A

```
+||A + A ⇔ 2  6  12  20 23
```

Scan (continued)

- For higher dimensional arrays, full scan operates in row major order

```
     1  1  1  1    0   1   2   3
+||  1  1  1  1    4   5   6   7
     1  1  1  1 ⇒  8   9  10  11
     1  1  1  1   12  13  14  15
```

- Yes, "or scan" is `|||` as in

```
           B  ⇔ 0  0  0  1  1  0  1  1
Run:=|||B  ⇔ 0  0  0  0  1  1  1  1
[2..n] Run := (Run != Run@w)*Index1;
       pos := max<< Run;
```

Partial Scan

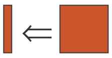
- Partial scans are possible too, but unlike reduction they do not reduce dimensionality, so the compiler cannot tell which dimension to reduce … so specify

`+||[2]`A is a partial scan in the 2nd dimension

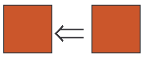
```
        1  1  1  1    0   1   2   3
+||[2]  1  1  1  1    0   1   2   3
        1  1  1  1 ⇒  0   1   2   3
        1  1  1  1    0   1   2   3
```

Remembering Reduce, Scan & Flood

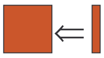
- The operators for reduce, scan and flood are suggestive …
  - Reduce **<<** produces a result of smaller size



  - Scan **||** produces a result of the same size



  - Flood **>>** produces a result of greater size



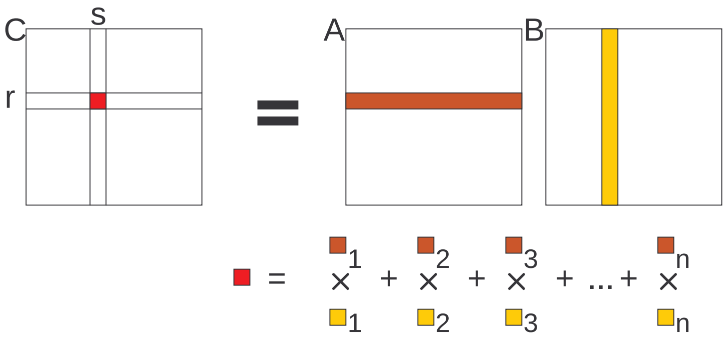
Summarizing: Don’t Change Rank

- Rather than change rank, use “singleton” values to collapse dimensions for lower rank
- For a region **`R = [1..m, 1..n]`**, the rank 2 arrays **`R1 = [1..m, 1]`** and **`R2 = [1, 1..n]`** are regions corresponding to the first column and row

- ZPL is designed to exploit the similarity between an array with collapsed dimensions and a corresponding array of lower rank

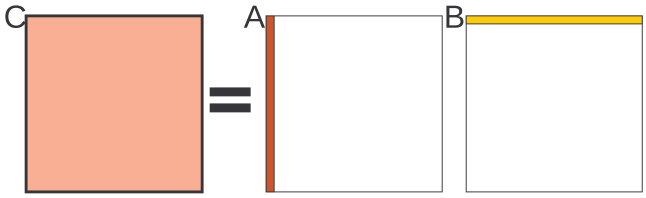
Recall Matrix Multiplication (MM)

- For n × n arrays A and B, compute C = AB where $c_{rs} = \sum_{1 \le k \le n} a_{rk} b_{ks}$



MM Illustrates Computing With Flood

- The SUMMA Algorithm



	b_{11}	b_{12}
a_{11}	$a_{11}b_{11}$	$a_{11}b_{12}$
a_{21}	$a_{21}b_{11}$	$a_{21}b_{12}$

Switch Orientation -- By using a *column* of A and a *row* of B broadcast to all, compute the "next" terms of the dot product

SUMMA Algorithm

- A column broadcast is simply a column flood and similarly a row broadcast is a row flood
- Define variables

```
var   Col : [1..m,*] double; -- Col flood array
      Row : [*,1..p] double; -- Row flood array
        A : [1..m,1..n] double;
        B : [1..n,1..p] double;
        C : [1..m,1..p] double;
```

SUMMA Algorithm (continued)

For each col-row in the common dimension, flood the item and combine it...

```
[1..m,1..p]    C := 0.0;       -- Initialize C
           for k := 1 to n do
   [1..m,*]  Col := >>[ ,k] A; -- Flood kth col of A
   [*,1..p]  Row := >>[k, ] B; -- Flood kth row of B
[1..m,1..p]    C += Col*Row;   -- Combine elements
           end;
```

--- or, more simply ---

```
            for k := 1 to n do
[1..m,1..p]  C += (>>[ ,k] A)*(>>[k, ] B);
            end;
```

SUMMA, The First Step

```
c11 c12 c13   a11 a12 a13 a14
c21 c22 c23   a21 a22 a23 a24
c31 c32 c33   a31 a32 a33 a34
c41 c42 c43   a41 a42 a43 a44
b11 b12 b13
b21 b22 b23
b31 b32 b33
b41 b42 b43
```

```
Col             Row
 a11 a11 a11     b11 b12 b13
 a21 a21 a21  ×  b11 b12 b13
 a31 a31 a31     b11 b12 b13
 a41 a41 a41     b11 b12 b13
```

```
C
 a11b11 a11b12 a11b13
 a21b11 a21b12 a21b13
 a31b11 a31b12 a31b13
 a41b11 a41b12 a41b13
```

SUMMA is the easiest MM algorithm to program in ZPL

Remap

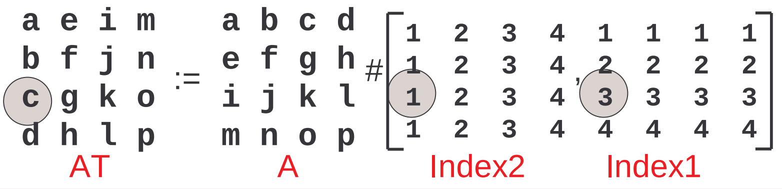
The remap operator (#) implements general data motion, including rank change

- Two cases:
  Gather, A := B#[C1,C2];
  Scatter, A#[C1,C2] := A
- For r rank array, provide r rank r arrays giving indices to be referenced
- Transpose:
  [R] AT := A#[Index2,Index1]; -- Idiom for transpose
  into position AT[i,j] goes A[j,i]

Remap (Gather)

The index array in the ith position gives the indices for the ith dimension

[R] AT := A#[Index2,Index1]; -- Idiom for transpose



$$\underset{AT}{\begin{matrix} a & e & i & m \\ b & f & j & n \\ \boxed{c} & g & k & o \\ d & h & l & p \end{matrix}} := \underset{A}{\begin{matrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{matrix}} \# \left[\underset{Index2}{\begin{matrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ \boxed{1} & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix}}, \underset{Index1}{\begin{matrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ \boxed{3} & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{matrix}} \right]$$

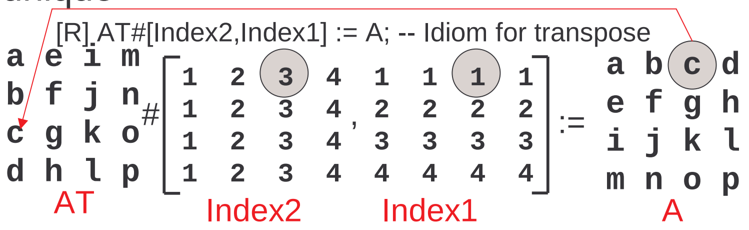
Gather: For a position, where does value come from

a c e b d f ⇔ a b c d e f#[1 3 5 2 4 6]

Remap (Scatter)

- Scatter Remap has a potential problem in that values can map to the same place … order is unspecified … use +=, etc. if not unique

[R] AT#[Index2,Index1] := A; -- Idiom for transpose



$$\underset{AT}{\begin{matrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{matrix}} \# \left[\underset{Index2}{\begin{matrix} 1 & 2 & \boxed{3} & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix}}, \underset{Index1}{\begin{matrix} 1 & 1 & \boxed{1} & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{matrix}} \right] := \underset{A}{\begin{matrix} a & b & \boxed{c} & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{matrix}}$$

Scatter: For a value, where does it go?

a d b e c f #[1 3 5 2 4 6] ⇔ a b c d e f

Shattered Control Flow

ZPL logically executes one instruction at a time

- There is a natural generalization in which statements are controlled by arrays rather than scalars

```
if A < 0 then A := -A;  -- define absolute
```

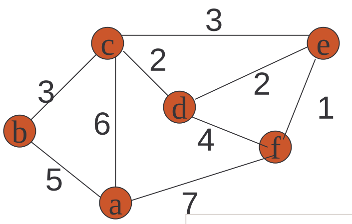
- Convenient for iterations

Let N and Nfact be defined [1..n]

```
Nfact := 1;
for i := 2 to N do
   Nfact := Nfact * i;  -- Compute N!
end;
```

Programming Homework

- Given an nxn (symmetric) array **E** of + edge weights with 0 on diagonals, ∞ for no edge
- Compute the all pairs shortest path: min(d[i,j],d[i,k]+d[k,j])



E	a	b	c	d	e	f
a	0	5	6	∞	∞	7
b	5	0	3	∞	∞	∞
c	6	3	0	2	3	∞
d	∞	∞	2	0	2	4
e	∞	∞	3	2	0	1
f	7	∞	∞	4	1	0

Assume all edge weights are less than 1000