Using Regions for Computation

Regions control data parallel execution by listing the indices to compute over. More complex operators require more complex uses of regions.

Index1 ...

- ZPL comes with “constant arrays” of any size
- Index[i] means indices of the jth dimension

\[
\begin{align*}
[1..n,1..n] & \quad \text{begin} \\
\quad & \quad Z := \text{Index1}; \quad \text{-- fill with first index} \\
\quad & \quad P := \text{Index2}; \quad \text{-- fill with second index} \\
\quad & \quad L := Z = P; \quad \text{-- define identity array} \\
\quad & \quad \text{end;}
\end{align*}
\]

- These arrays -- of arbitrary dimension -- are compiler created using no space

\[
\begin{array}{cccc}
\text{Index1} & 1 & 1 & 1 & 1 \\
& 2 & 2 & 2 & 2 \\
& 3 & 3 & 3 & 3 \\
& 4 & 4 & 4 & 4 \\
\text{Index2} & 1 & 2 & 3 & 4 \\
& 1 & 2 & 3 & 4 \\
& 1 & 2 & 3 & 4 \\
& 1 & 2 & 3 & 4 \\
\end{array}
\]
Recall Reduction

- The reduction operation (\(<\)) “reduces” an array to a single value using an associative operator: +\(<\), *\(<\), max\(<\), min\(<\), &\(<\), |\(<\)

- For example, +\(<\) is summation (\(\Sigma\)) and max\(<\) is global maximum as used in Jacobi

\[[1..n, 1..n]\] err := max\(<\) abs(Temp - A);

- Think of the \(n^2\) elements reduced to a scalar

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
  - [n,1..n] is a 2D array (but a single row) allocated with the last row
  - This allocation property is important; more later
Partial Reductions

- Partial reductions reduce dimensions without reducing the entire array, e.g. adding up rows.
- Partial reductions require two regions, one on the operator and one on the statement.
  \[
  \begin{align*}
  \text{Let } & A \iff [1..n,1..n], \text{ Col1 } \iff [1..n,1] \text{ Rown } \iff [n.1..n] \\
  [1..n, 1] \text{ Col1 } & := +<[1..n,1..n]; \quad \text{-- Add up rows} \\
  [n,1..n] \text{ Rown } & := \max[1..n,1..n]; \quad \text{-- Max down cols}
  \end{align*}
  \]
- The compiler compares the two regions and figures out which one(s) reduce.

Scan

- Scan is the parallel prefix operation for associative operators: +, *, min, max, &, |.
- Scan is like reduction, but uses ||.
- Prefix sum is + ||
  \[
  \begin{align*}
  A & \iff 2 \ 4 \ 6 \ 8 \ 0 \\
  +|| & A \iff 0 \ 2 \ 6 \ 12 \ 20
  \end{align*}
  \]
- First position gets identity, and each position gets preceding elements combined.
- For higher dimensional arrays, full scan operates in row major order.
  \[
  \begin{align*}
  +|| & 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 2 \ 3 \\
  & 1 \ 1 \ 1 \ 1 \ \Rightarrow 4 \ 5 \ 6 \ 7 \\
  & 1 \ 1 \ 1 \ 1 \ \Rightarrow 8 \ 9 \ 10 \ 11 \\
  & 1 \ 1 \ 1 \ 1 \ 12 \ 13 \ 14 \ 15
  \end{align*}
  \]
Scan continued

- Yes, “or scan” is ||| as in
  \[ B \leftrightarrow 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \]
  \[ \text{Run} := ||| B \leftrightarrow 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \]
  \[ [2..n] \text{ Run} := (\text{Run} != \text{Run@w}) * \text{Index1}; \]
  \[ \text{pos} := \text{max} \ll \text{Run}; \]

- Partial scans are possible too, but 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

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
  \[ [1..n,1..n] A := >>[1..n,k] B; -- \text{Replicate B’s kth column} \]

- The replication uses broadcast, often an efficient operation

- Multiply a matrix times a vector...flood vector and then multiply: \[ A\leftrightarrow[1..n,1..n] V\leftrightarrow[1,1..n]: \]
  \[ [1..n,1..n] A := A * >>[1,1..n] V; -- \text{Replicate V} \]
**Flood Regions and Arrays**

Flood regions 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.

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

Think of column in every position

---

**Flood arrays (continued)**

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

• Scaling a value by max of row:

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

Flood makes combining different ranks “element-wise”

The promotion of flooded arrays is only logical
### Remembering Reduce, Scan & Flood

- The operators for reduce, scan and flood are suggestive ...
  - Reduce $\ll$ produces a result of smaller size
  - Scan $||$ produces a result of the same size
  - Flood $\gg$ 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) Definition

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

![Diagram of matrix multiplication](image)

MM Illustrates Computing With Flood

• The SUMMA Algorithm

![Diagram of SUMMA algorithm](image)

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 for row broadcast is a row flood
- Define variables

```plaintext
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

```plaintext
[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;
```

SUMMA is the easiest MM algorithm to program in ZPL
SUMMA, The First Step

\[
\begin{array}{cccc}
\text{c11} & \text{c12} & \text{c13} & \text{a11} \\
\text{c21} & \text{c22} & \text{c23} & \text{a21} \\
\text{c31} & \text{c32} & \text{c33} & \text{a31} \\
\text{c41} & \text{c42} & \text{c43} & \text{a41} \\
\text{b11} & \text{b12} & \text{b13} \\
\text{b21} & \text{b22} & \text{b23} \\
\text{b31} & \text{b32} & \text{b33} \\
\text{b41} & \text{b42} & \text{b43} \\
\end{array}
\]

Still Another MM Algorithm

If flooding is so good for columns/rows, why not use it for whole planes?

region IK = [1..n,*,1..n]  
    JK = [*,,1..n,1..n];  
    IJ = [1..n,1..n,*];  
    IJK = [1..n,1..n,1..n];

[IK] A2 := A#INDEX1, INDEX2;  
[JK] B2 := B#INDEX2,INDEX1;  
[IJ] C := +<<[IJK]>>(IK)A2)*>>(JK)B2;
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
  
  \[
  \text{if } A < 0 \text{ then } A := -A; \quad \text{-- define absolute}
  \]

- Convenient for iterations

  Let \( N \) and \( N\)fact be defined \([1..n]\)
  
  \[
  \text{Nfact := 1;}
  \text{for } i := 2 \text{ to } N \text{ do}
  \quad \text{Nfact := Nfact * } i; \quad \text{-- Compute } N!
  \text{end;}
  \]

Programming Homework

- Given an \( nxn \) (symmetric) array \( E \) of pos edge weights with 0 on diagonals, \( \infty \) for no edge

- Compute the all pairs shortest path:
  \[
  \text{min}(d[i,j],d[i,k]+d[k,j])
  \]

\[\begin{array}{ccccccc}
   & a & b & c & d & e & f \\
 a & 0 & 5 & 6 & \infty & \infty & 7 \\
b & 5 & 0 & 3 & \infty & \infty & \infty \\
c & 6 & 3 & 0 & 2 & 3 & \infty \\
d & \infty & \infty & 2 & 0 & 2 & 4 \\
e & \infty & \infty & 3 & 2 & 0 & 1 \\
f & 7 & \infty & \infty & 4 & 1 & 0 \\
\end{array}\]

Assume all edge weights are less than 1000