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.

© 2002-2005 Lawrence Snyder, All Rights Reserved

Index1 ...

- ZPL comes with “constant arrays” of any size
- Index\(i\) means indices of the \(i\)th dimension

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 2 & 3 & 4 & 1 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 1 & 2 & 3 & 4 & 0 & 1 & 0 & 0 \\
3 & 3 & 3 & 3 & 1 & 2 & 3 & 4 & 0 & 0 & 1 & 0 \\
4 & 4 & 4 & 4 & 1 & 2 & 3 & 4 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

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

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

- The replication uses broadcast, often an efficient operation

- Matrix vector operations…flood vector to match shape: \( A [1..m,1..n] \ \text{MaxC} [1..m,1] : \)

  \[
  [1..m,1] \ \text{MaxC} := \max\llbracket [1..m,1..n] \ A \rrbracket; \quad \text{--Find max of each row} \\
  [1..m,1..n] \quad A := A / >>[1..m,1] \ \text{MaxC}; \quad \text{--Scale each row by max}
  \]

Closer Look At Scaling Each Row

\[
[1..m,1] \ \text{MaxC} := \max\llbracket [1..m,1..n] \ A \rrbracket; \quad \text{--Find max of each row} \\
[1..m,1..n] \quad A := A / >>[1..m,1] \ \text{MaxC}; \quad \text{--Scale each row by max}
\]

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

\[
\begin{array}{cccccccc}
2 & 4 & 4 & 2 & 4 & 4 & 4 & 4 \\
0 & 2 & 3 & 6 & 6 & 6 & 3 & 6 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
8 & 2 & 4 & 0 & 8 & 8 & 8 & 8 \\
\end{array}
\]

\[
A \quad \text{MaxC} \quad >>[1..m,1] \ \text{MaxC}
\]

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

```plaintext
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:

```plaintext
[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 \([1..n,\ast]\) and \([\ast,1..n]\) allocated to 4 processors

Scan

- Scan is the parallel prefix operation for associative operators: \(+, \ast, \min, \max, \&, |\)
- Scan is like reduction, but uses \(\mid\mid\)
- Exclusive prefix sum is \(+\mid\mid\)
  
  \[
  \begin{array}{c}
  A \leftrightarrow 2 4 6 8 3 \\
  +\mid A \leftrightarrow 0 2 6 12 20
  \end{array}
  \]
  - 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 … \(\text{In} \equiv \text{Ex op } A\)
  
  \[
  +\mid A + A \leftrightarrow 2 6 12 20 23
  \]
Scan (continued)

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

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
\end{array}
\]

- Yes, “or scan” is \[ || \] as in

\[
\begin{array}{cccc}
B & \Leftarrow& 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
\end{array}
\begin{array}{cccc}
\Leftarrow& 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
[2..n] \text{ Run } := (\text{Run } != \text{ Run@w}) \ast \text{Index1;}
\]

\[
pos := \text{max}<< \text{Run;}
\]

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

\[
+||[2] \text{ A is a partial scan in the 2nd dimension}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
\end{array}
\]
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)

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

\[
\begin{array}{c}
\text{C} \\
\text{r} \\
\end{array}
= 
\begin{array}{c}
\text{A} \\
\text{B} \\
\end{array}
\]

MM Illustrates Computing With Flood

• The SUMMA Algorithm

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

```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;
--- 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
   b11  b12  b13  

   c21  c22  c23  a21  a22  a23  a24
   b21  b22  b23  

   c31  c32  c33  a31  a32  a33  a34
   b31  b32  b33  

   c41  c42  c43  a41  a42  a43  a44
   b41  b42  b43  
```

**SUMMA is the easiest MM algorithm to program in ZPL**

---

**Remap**

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

- **Two cases:**
  
  Gather, $A := B[#(C_1, C_2)]$;
  
  Scatter, $A[#(C_1, C_2)] := A$

- **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

\[ \text{AT} := \text{A}[\text{Index2}, \text{Index1}]; \quad \text{-- Idiom for transpose} \]

Gather: For a position, where does value come from

\[ \text{acebdf} \Leftrightarrow \text{abcdef}#[1\ 3\ 5\ 2\ 4\ 6] \]

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

\[ \text{AT} := \text{A}[\text{Index2}, \text{Index1}]; \quad \text{-- Idiom for transpose} \]

Scatter: For a value, where does it go?

\[ \text{adbecf} \#[1\ 3\ 5\ 2\ 4\ 6] \Leftrightarrow \text{abcdef} \]
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 \( N_{\text{fact}} \) be defined \([1..n]\)

  \[
  \begin{align*}
  N_{\text{fact}} & := 1; \\
  \text{for } i & := 2 \text{ to } N \text{ do} \\
  & \quad N_{\text{fact}} := N_{\text{fact}} \times i; \quad -- \text{Compute } N! \\
  \text{end;}
  \end{align*}
  \]

Programming Homework

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

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

Assume all edge weights are less than 1000