Today's Plan

Higher level languages

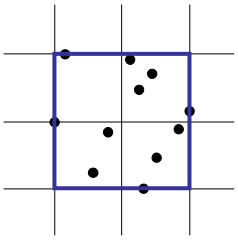
- Using ZPL
- Performance portability

Finding the Bounding Box

Given

- X and Y are 1D arrays of coordinates such that (X_i, Y_i) is a position in the 2D plane
- How do you compute the bounding box in ZPL?

```
[R] begin
      rightedge  := max<< X;
      topedge    := max<< Y;
      leftedge   := min<< X;
      bottomedge := min<< Y;
    end;
```



Bounding Box Using Records

Using a Point Type

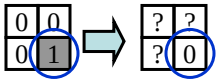
```
type point = record
        x : integer;       -- x coordinate
        y : integer;       -- y coordinate
      end;
var Points : [1..n] point;  -- points in a plane
. . .
[R] begin
      rightedge  := max<< Points.x;
      topedge    := max<< Points.y;
      leftedge   := min<< Points.x;
      bottomedge := min<< Points.y;
    end;
```


Recall the Connected Components Algorithm

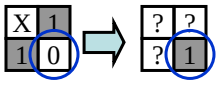
The Amazing Levialdi Shrinking Operator (1972)

- Each pixel simultaneously changes state according to the following rules

(1) A 1 bit becomes a 0 if there are 0's to its West, NW, and North



(2) A 0 bit becomes a 1 if there are 1's to its West and North



(3) All other bits remain unchanged

8-way Connected Components

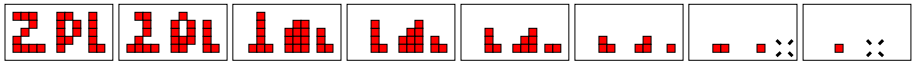
ZPL Solution

```
. . .
Count := 0;
repeat
  Next  := Image & (Image@north | Image@nw | Image@west);
  Next  := Next  | (Image@west  & Image@north & !Image);
  Conn  := Next@east | Next@se | Next@south;
  Conn  := Image & !Next & !Conn;
  Count += Conn;
  Image := Next;
  smore := |<< Next;
until !smore;
. . .
```

Rule 1

Rule 2

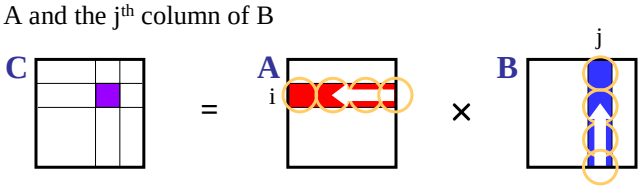
Test for Poof



Matrix Multiplication

Observation

– To compute the (i,j) element of C, we compute the dot product of the i^{th} row of A and the j^{th} column of B

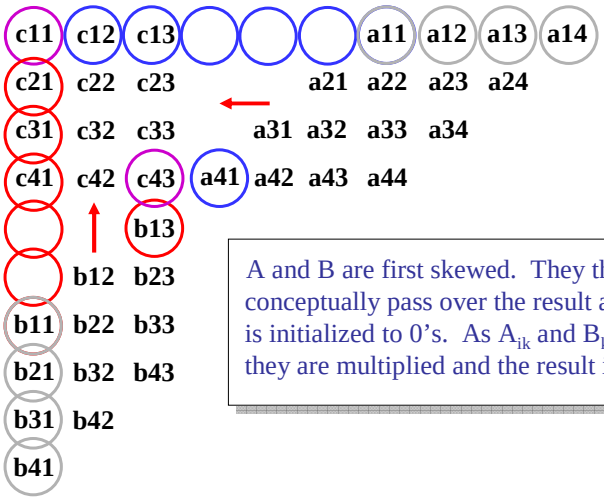


Cannon's algorithm [Cannon '69]

– For n × n matrices, each dot product consists of n multiplications

– If we skew the rows of A and the columns of B appropriately, we can align these n multiplications by shifting the A matrix to the left and the B matrix to the top

Cannon's Algorithm [1969]

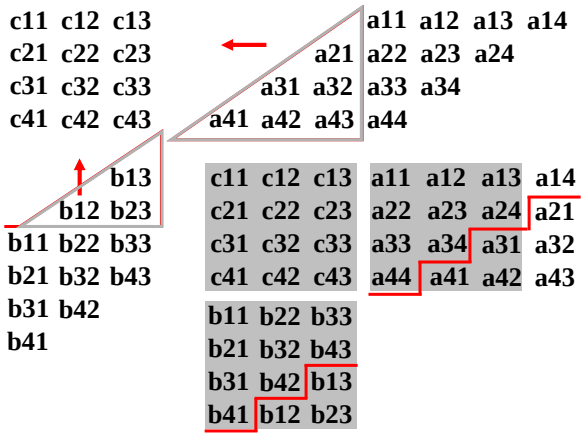


A and B are first skewed. They then conceptually pass over the result array C, which is initialized to 0's. As A_{ik} and B_{kj} pass over C_{ij}, they are multiplied and the result is added to C_{ij}

Skewed Arrays

ZPL supports only dense arrays

– How do we represent skewed arrays?

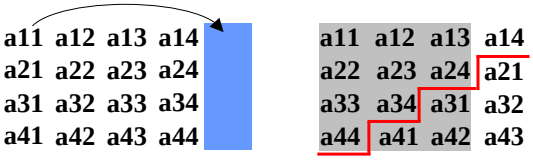


Skewing the Arrays

Use the wrap operator

– Wrap the first column to the right border, then shift left

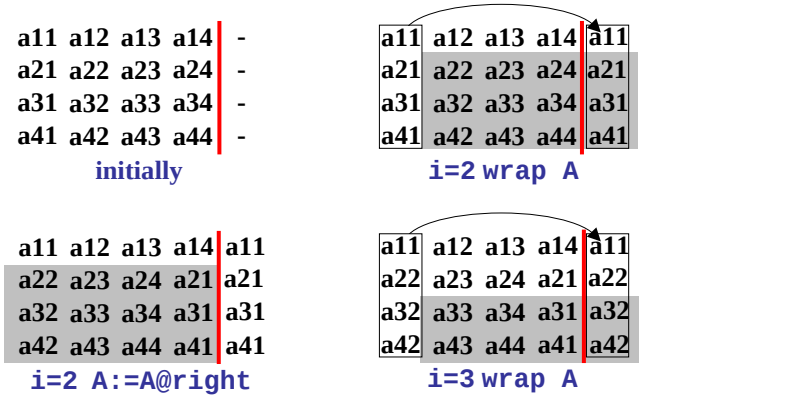
```
region Lop = [1..m, 1..n];
direction right = [0,1];
```




```
for i := 2 to m do
[right of Lop]  wrap A;       -- Move col 1 to right border
  [i..m, 1..n]  A := A@right; -- Shift last i rows left
end;
```

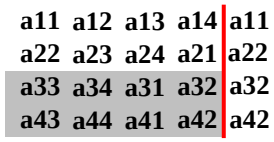

The Four Steps of Skewing A

```
for i := 2 to m do
[right of Lop]  wrap A;       -- Move col 1 to right border
  [i..m, 1..n]  A := A@right; -- Shift last i rows left
end;
```

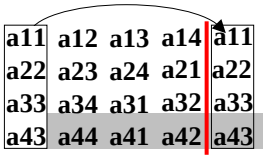


The Four Steps of Skewing A

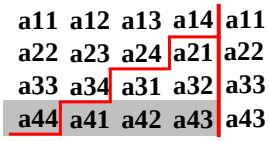
```
for i := 2 to m do
[right of Lop]  wrap A;       -- Move col 1 to right border
  [i..m, 1..n]  A := A@right; -- Shift last i rows left
end;
```



i=3 A:=A@right



i=4 wrap A



i=4 A:=A@right

Cannon's Algorithm

Skew A, Skew B, Multiply, Accumulate, Rotate

```
           for i := 2 to m do  -- Skew A
[right of Lop]  wrap A;        -- Move col 1 to right
  [i..m, 1..n]  A := A@right;  -- Shift last i rows left
           end;
           for i := 2 to p do  -- Skew B
[right of Rop]  wrap B;        -- Move row 1 to below last
  [1..n, i..p]  B := B@below;  -- Shift last i columns up
           end;
         [Res]  C := 0.0       -- Initialize C
           for i := 1 to n do  -- For A & B's common dim
         [Res]  C := C + A * B; -- Accumulate product
[right of Lop]  wrap A;        -- Send first col right
         [Lop]  A := A@right;  -- Shift array left
[below of Rop]  wrap B;        -- Send top row down
         [Rop]  B := B@below;  -- Shift array up
           end;
```

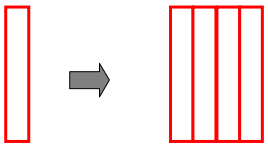
Flood

Flooding

- The Flood operation is the inverse of a reduce operation
- It replicates data from lower dimensions to higher dimensions

Example

- Can use flood to perform matrix-vector operations
  - Flood the vector to be conformal with the array



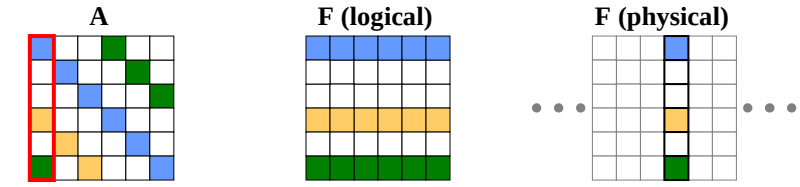
The Flood Operator

Flooding

- In ZPL, the flood operator (>>) looks like the inverse of a reduction
- It takes two regions, one applied to the statement, and the other applied to the operator

Replicate A's 1st column

```
[1..n, 1..m]  F := >>[1..n, 1] A;
```



Collapsed regions

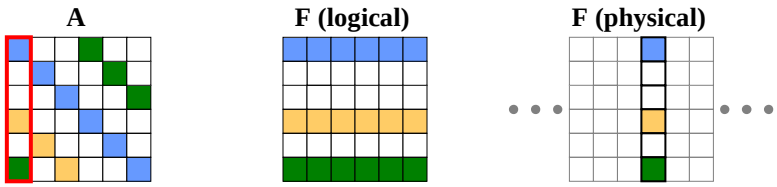
- One or more of the operator's region must be collapsed to a singleton
- Replication occurs across collapsed regions

Language Support for Flooding

Flooding

- ZPL has the flood operator (**>>**)
- Fortran 90 has **spread**
- MATLAB has "Tony's Trick"

```
ZPL:        [1..n,1..m] F := >>[1..n, 1] A;
Fortran90:              F  = SPREAD (A[:,1], DIM=2,N)
MATLAB:                 F  = A(:,ones(1,size(A,2)))
```



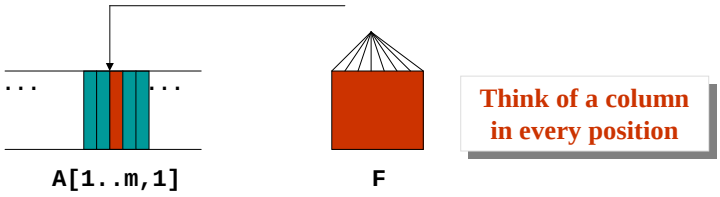
Flood Regions and Flood Arrays

Over-specification

- The collapsed singleton dimension over-specifies the situation
- To specify **any** column, use *

```
region   FloodCol = [1..m, *];      -- Flood region

[FloodCol] F := >> [1..m, 1] A;
```

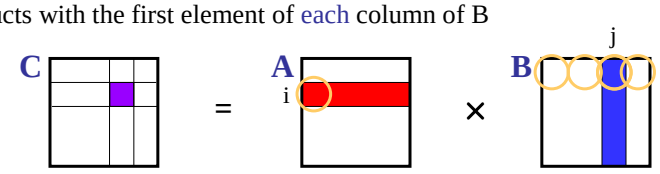


Think of a column in every position

Matrix Multiplication Revisited

Observation

- The first element of the i^{th} row of A will be used to compute partial dot products with the first element of each column of B



SUMMA algorithm [Watts and van de Geijn 95]

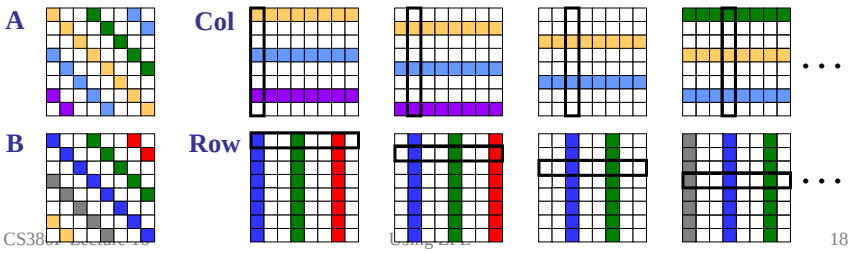
- Each element in each column of A contributes to n dot products
- Each element in each row of B contributes to n dot products
- Iteratively
  - Broadcast columns of A to all processors
  - Broadcast rows of B to all processors
  - Compute $1/n^{th}$ of the dot product for each element of C

Matrix Multiplication Revisited

SUMMA algorithm

- Iteratively flood a column of A and a row of B into temporary matrices
- Multiply and accumulate these results into C

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

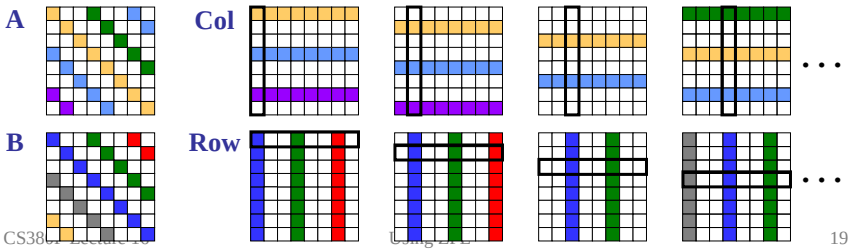


SUMMA Algorithm

Invariant

– On the k[th] iteration, the k[th] term in the dot-product of row i and column j is accumulated into position i,j

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



Comparing the Algorithms

Which matrix multiplication algorithm is better?

Cannon

```
Skew A
Skew B
loop through n
   C += A*B
   rotate A, B
```

SUMMA

```
loop through n
   flood A[,k]
   flood B[k,]
   C += A*B
```

Other ZPL Constructs

Permutation

- ZPL supports non-local data movement with the permutation operators
  - Gather: `# on rhs`
  - Scatter: `# on lhs`

Hierarchical arrays

Sparse arrays

Distribution change